PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers

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[‡], Jiawang Tong[‡]
mingsong.cw@alibaba-inc.com
[†]Zhejiang University and [‡]Alibaba Group

ABSTRACT

The trend in the DBMS market is to migrate to the cloud for elasticity, high availability, and lower costs. The traditional, monolithic database architecture is difficult to meet these requirements. With the development of high-speed network and new memory technologies, disaggregated data center has become a reality: it decouples various components from monolithic servers into separated resource pools (e.g., compute, memory, and storage) and connects them through a high-speed network. The next generation cloud native databases should be designed for disaggregated data centers.

In this paper, we describe the novel architecture of *PolarDB Serverless*, which follows the *disaggregation* design paradigm: the CPU resource on compute nodes is decoupled from remote memory pool and storage pool. Each resource pool grows or shrinks independently, providing on-demand provisioning at multiple dimensions while improving reliability. We also design our system to mitigate the inherent penalty brought by resource disaggregation, and introduce optimizations such as optimistic locking and index awared prefetching. Compared to the architecture that uses local resources, *PolarDB Serverless* achieves better dynamic resource provisioning capabilities and 5.3 times faster failure recovery speed, while achieving comparable performance.

CCS CONCEPTS

• **Information systems** → **Data management systems**; • **Networks** → *Cloud computing*.

KEYWORDS

cloud database; disaggregated data center; shared remote memory; shared storage

ACM Reference Format:
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[‡], and Yusong Gao[‡], Songlu Cai[‡], Yunyang Zhang[‡], Jiawang Tong[‡]. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457560

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SIGMOD '21, June 20–25, 2021, Virtual Event, China
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457560

1 INTRODUCTION

As enterprises move their applications to the cloud, they are also migrating their databases to the cloud. The driving force of this trend is threefold. First, cloud vendors provide the "pay-as-you-go" model that allows customers to avoid paying for over-provisioned resources, resulting in significant *cost reduction*. Second, marketing activities such as Black Friday and Singles' Day often demand rapid but transient resource expansion from the database systems during peak time, where cloud vendors can offer such *elasticity* to customers. Third, cloud vendors are able to quickly upgrade and evolve the database systems to maintain competitiveness and repair defects in time while sustaining *high availability*. Customers always expect that node failures, especially planned downtime and software upgrades, will have less impact on their business.

Cloud vendors such as AWS [43], Azure [2], GCP and Alibaba [9] provide relational database as a services (DBaaS). There are three typical architectures for cloud databases: *monolithic machine* (Figure 1), *virtual machine with remote disk* (Figure 2(a)), and *shared storage* (Figure 2(b)), and the last two can be referred as *separation of compute and storage*. Though these architectures have been widely used, they all suffer from challenges caused by resource coupling.

Under the *monolithic machine* architecture, all resources (such as CPU, memory and storage) are tightly coupled. The DBaaS platform needs to solve bin-packing problems when assigning database instances to machines. It is difficult to make different resources allocated on a physical machine all have a high utilization rate, which is prone to fragmentation. Moreover, it is difficult to meet the demands of customers to adjust individual resources flexibly according to the load at runtime. Finally, a system with tightly coupled resources has the problem of fate sharing, *i.e.*, the failure of one resource will cause the failure of other resources. Resources cannot be recovered independently and transparently, which leads to longer system recovery time.

With the *separation of compute and storage* architecture, DBaaS can independently improve the resource utilization of the storage pool. The *shared storage* subtype further reduces storage costs — the primary and read replicas can attach and share the same storage. Read replicas help to serve high volume read traffic and offload analytical queries from the primary. However, in all these architectures,

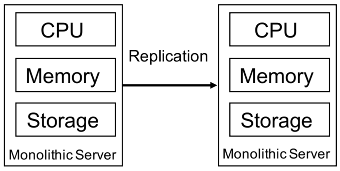
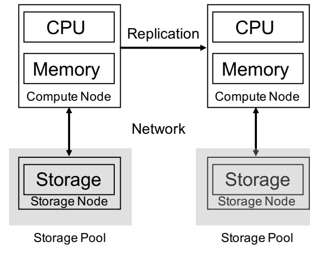
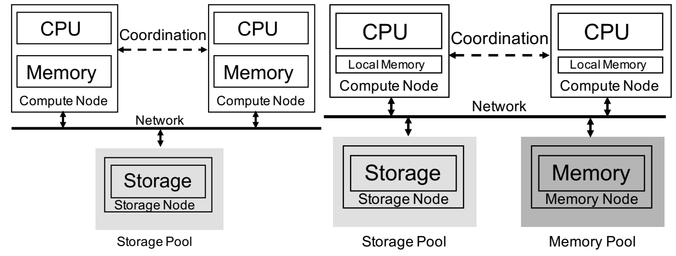


Figure 1: *monolithic machine*



(a) *virtual machine with remote disk*



(b) *shared storage*

Figure 2: *separation of compute and storage*

Figure 3: *disaggregation*

problems like bin-packing of CPU and memory, lacking of flexible and scalable memory resources, remain unsolved. Furthermore, each read replica keeps a redundant in-memory data copy, leading to high memory costs.

In this paper, we propose a novel cloud database design paradigm of the *disaggregation* architecture (Figure 3). It goes one step further than the *shared storage* architecture, to address the aforementioned problems. The disaggregation architecture runs in the disaggregated data centers (DDC), in which CPU, memory and storage resources are no longer tightly coupled as in a monolithic machine. Resources are located in different nodes connected through high-speed network. As a result, each resource type improves its utilization rate and expands its volume independently. This also eliminates fate sharing, allowing each resource be recovered from failure and upgraded independently. Moreover, data pages in the remote memory pool can be shared among multiple database processes, analogous to the storage pool being shared in *shared storage* architecture. Adding a read replica no longer increases the cost of memory resources, except for consuming a small piece of local memory.

A trend in recent years is that cloud-native database vendors are launching serverless variants [3, 4]. The main feature of serverless databases is on-demand resource provisioning (such as auto-scaling and auto-pause), which should be transparent and seamless without interrupting customer workloads. Most cloud-native databases are implemented based on the *shared storage* architecture, where CPU and memory resources are coupled and must be scaled at the same time. In addition, auto-pause has to release both resources, resulting in long resumption time. We show that *disaggregation* architecture can overcome these limitations.

PolarDB Serverless is a cloud-native database implementation that follows the *disaggregation* architecture. Similar to major cloud-native database products like Aurora, HyperScale, and PolarDB[1], it includes one primary (RW node) and multiple read replicas (RO nodes) in the database node layer. With the *disaggregation* architecture, it is possible to support multiple primaries (RW nodes), but this is not within the scope of this paper.

The design of a multi-tenant scale-out memory pool is introduced in *PolarDB Serverless*, including page allocation and life cycle management. The first challenge is to ensure that the system executes transactions **correctly** after adding remote memory to the system. For example, read after write should not miss any updates even across nodes. We realize it using cache invalidation. When RW is splitting or merging a B+Tree index, other RO nodes should not see an inconsistent B-tree structure in the middle. We protect it with global page latches. When a RO node performs read-only transactions, it must avoid reading anything written by uncommitted transactions. We achieve it through the synchronization of read views between database nodes.

The evolution of the *disaggregation* architecture could have a negative impact on the database performance. It is because the data is likely to be accessed from the remote, which introduces significant network latency. The second challenge is to execute transactions **efficiently**. We exploit RDMA optimization extensively, especially one-sided RDMA verbs, including using RDMA CAS [42] to optimize the acquisition of global latches. In order to improve concurrency, both RW and RO use optimistic locking techniques to avoid unnecessary global latches. On the storage side, page materialization offloading allows dirty pages to be evicted from remote memory without flushing them to the storage, while index-aware prefetching improve query performance.

The *disaggregation* architecture complicates the system and hence increases the variety and probability of system failures. As a cloud database service, the third challenge is to build a **reliable** system, we summarize our strategies to handle single-node crashes of different node types which guarantee that there is no single-point failure in the system. Because the states in memory and storage are decoupled from the database node, crash recovery time of the RW node becomes 5.3 times faster than that in the *monolithic machine* architecture.

We summarize our main contributions as follows:

- We propose the *disaggregation* architecture and present the design of *PolarDB Serverless*, which is the first cloud database implementation following the architecture. We demonstrate that this architecture provides new opportunities for the design of new cloud-native and serverless databases.
- We provide design details and optimizations that make the system work correctly and efficiently, overcoming the performance drawbacks brought by the *disaggregation* architecture.
- We describe our fault tolerance strategies, including the handling of single-point failures and cluster failures.

The remainder of this paper is organized as follows. In Section 2, we introduce backgrounds of PolarDB and DDC. Section 3 explains the design of *PolarDB Serverless*. Section 4 presents our performance optimizations. Section 5 discusses our fault tolerance and recovery strategies. Section 6 gives the experimental results. Section 7 reviews the related work, and Section 8 concludes the paper.

[1] *PolarDB Serverless* is developed on a fork of PolarDB's codebase.

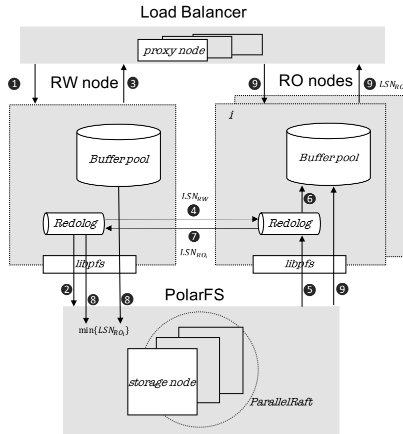


Figure 4: PolarDB Architecture

2 BACKGROUND

2.1 PolarDB

PolarDB [26] is a cloud-native database with *shared storage* architecture. It is derived from the MySQL code base and uses PolarFS [9] as the underlying storage pool. It includes one primary (RW node) and multiple read replicas (RO nodes) in the compute node layer. Like a traditional database kernel, Each RW or RO node contains a SQL processor, transaction engines (like InnoDB [35], X-Engine [18]), and a buffer pool to serve queries and transactions. In addition, there are some stateless proxy nodes for load balancing.

PolarFS is a durable, atomic and scale-out distributed storage service. It provides virtual volumes that are partitioned into chunks of 10GB size which are distributed in multiple storage nodes. A volume contains up to 10000 chunks, and can provide a maximum capacity of 100TB. Chunks are provisioned on demand so that volume space grows dynamically. Each chunk has three replicas, and linear serializable is guaranteed through *Parallel Raft*, which is a consensus protocol derived from Raft.

The RW node and RO nodes synchronize memory status through redo logs, and coordinate consistency through log sequence number(LSN), which indicates an offset of redo log files in InnoDB. As shown in Figure 4, in a transaction ①, after RW finishes flushing all redo log records to PolarFS ②, the transaction can be committed ③. RW broadcasts messages that the redo log have been updated and the latest LSN lsn_{RW} to all RO nodes asynchronously ④. After the node RO_i received the message from RW, it pulls updates of redo log from PolarFS ⑤, and applies them to the buffered page in buffer pool ⑥, so that RO_i keeps synchronization with RW. Then RO_i piggybacks the consumed redo log offset lsn_{RO_i} in the reply and send it back to RW ⑦. RW can purge the redo log before the $min\{lsn_{RO_i}\}$ location, and flush the dirty pages elder than $min\{lsn_{RO_i}\}$ to PolarFS in the background ⑧. While RO_i can serve read transactions using the snapshot isolation with version before lsn_{RO_i} ⑨. Some RO nodes may fall behind because of high CPU utilization or network congestion. Say there is a certain node RO_k, whose LSN lsn_{RO_k} is much lower than that of RW lsn_{RW} (the lag is larger than one million). Such node RO_k will be detected and kicked out of the cluster to avoid slowing down RW to flush dirty pages.

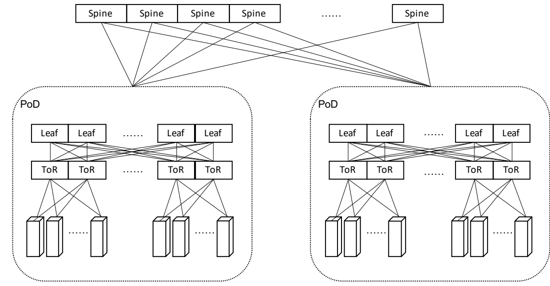


Figure 5: Network Architecture at Alibaba

Proxy nodes provide transparent load balancing service for separating read and write traffics, distribute read requests to RO nodes and forward write requests to RW.

2.2 Disaggregated Data Centers

In disaggregated data centers, compute nodes, memory nodes, and storage nodes are interconnected through a high-speed network. The recent development of RDMA network technology [17] (such as InfiniBand, RoCE/RoCEv2, and iWARP) makes it possible to build large-scale disaggregated data centers. Alibaba's data center network is an Ethernet-based multi-layer clos network, which adopts RDMA technology RoCEv2 based on Ethernet.

As shown in Figure 5, a typical data center at Alibaba has three layers: the spine layer, the leaf layer, and the ToR layer. A ToR switch connects up to 48 nodes. ToR switches connect to leaf switches. Leaf switches connect to spine switches. Each machine is equipped with a dual-port RDMA NIC, which is connected to two different ToR switches to support continuous service in case of single network link failure. A leaf switch group is composed of multiple leaf switches, which provide services at the same time and serve as backups for each other. The collection of racks and servers managed by a leaf switches group is called a PoD (Point of Delivery). A PoD contains up to 32 groups of ToR, thus more than 1500 servers. The servers in a PoD are interconnected through the RDMA network.

Under the *disaggregation* architecture, the computation, memory and storage resources required by a single database instance are allocated under the same PoD. And, the resources of different instances can be placed to different PoDs. Computation and memory resources tend to be allocated together under the same ToR for lower latency and fewer jitters.

2.3 Serverless Databases

Serverless databases are highly elastic derivatives of cloud-native databases, such as AWS Aurora Serverless [3] and Azure SQL Serverless [4]. Their main design goal is on-demand resource provisioning, which enables agile resource adjustment according to actual workloads and provides a pay-as-you-go payment model. More specifically, *auto-scaling* pops up resources during peak periods and shrinks when the surge subsides. This can help many businesses meet both their performance and budget constraints. Similarly, *auto-pause* releases computing resources during inactive periods and hence only storage consumption is billed. When traffic comes, the database can be automatically resumed with little recovery time.

The *auto-scaling* capacity of existing serverless databases have limitations, due to the underlying shared storage architecture [2, 43]. Their CPU and memory resources are tightly coupled, which must be scaled up and down simultaneously at the granularity of *resource*

unit. A resource unit is a package of fixed CPU and memory resources on the compute nodes (virtual machines or containers), e.g., an ACU (Aurora Compute Unit) contains 2 GiB of memory and associated virtual processor. This setting fixes the resource ratio and is inadequate to satisfy all scenarios. For example, analytical database customers have higher requirement for memory capacity than for CPU capacity, since large amounts of data should be cached in memory for fast access. In contrast, transactional database customers demands more CPU resources to support their business peaks, where small memory is adequate for a high cache hit rate. Under the *disaggregation* architecture, since CPU and memory are completely decoupled, the resource provision can be more cost-effective for auto-scaling.

Similarly, the *auto-pause* capacity of existing serverless databases are also limited. They have to release both CPU and memory resources, resulting in relatively long resumption time. Under the *disaggregation* architecture, CPU and memory no longer share fate. Hence, memory can be retained during auto-pause and database resumption becomes much faster, which eliminates the loading of key data (such as data dictionary and pages) from remote storage.

Moreover, *scaling transparency* is another design goal for serverless databases. Ideally, even when the database scales across compute nodes, the migration shall not interrupt customer workloads. With the *disaggregation* architecture, ephemeral states of the database (such as dirty pages, transaction status, logical locks, and intermediate query results) can be stored in the shared memory, which provides more opportunities for better transparency. *PolarDB Serverless* now stores dirty pages in shared memory and others are left for future work.

3 DESIGN

PolarDB Serverless is a cloud native database of the *disaggregation* architecture, evolved from PolarDB. Similar to the PolarDB architecture described in Section 2.1, each *PolarDB Serverless* instance consists of multiple proxy nodes, one RW node, multiple RO nodes, and uses PolarFS as the underlying storage pool. The biggest difference with PolarDB is the use of the remote memory pool.

The advantage of shared memory is that it allows pages to be shared between RW and RO nodes instead of maintaining a private copy for each node, which improves memory resource usage. The size of shared memory can be scaled horizontally, allowing the database to have a larger memory pool where data can be completely cached. This memory pool has lower latency than remote storage and is conductive to analytic workloads. However, the adoption of shared memory also brings a performance penalty. In particular, the following issues are addressed: First, the access speed of remote memory is much slower than that of local memory, where a tiering memory system and optimizations like prefetching are required. Second, the private pages previously managed by RW and RO now become shared resources, where a mechanism for cross-node mutual exclusion is required. We extensively use low-latency one-sided RDMA verbs (like RDMA CAS [42]) and optimistic protocols to avoid the use of global latches. Third, the transmission of pages (e.g. flushing dirty pages) brings burden to the network. Similar to Aurora and Socrates, we write redo logs to the storage and materialize pages from logs asynchronously.

3.1 Disaggregated Memory

3.1.1 Remote Memory Access Interface. Database processes access the remote memory pool via the *librmem* interface. The five most important APIs are listed below. A Page is identified by the *page_id* (*space*, *page_no*). *page_register* and *page_unregister* manage the life cycle of cached pages inside the remote memory. The database node obtains the address of a page in the remote memory pool through *page_register*, and increases the page's reference count by one. If the page does not exist, this function allocates space for the page in the remote memory pool, otherwise, the page is copied to the caller's local cache. Conversely, *page_unregister* decreases the page reference count by one. When the page reference count reaches 0, the page can be deleted from the remote memory pool. *page_read* fetches a page from the remote memory pool to the local cache using one-sided RDMA read. *page_write* writes a page from the local cache to the remote memory pool using one-sided RDMA write. *page_invalidate* is called by RW to invalidate all copies of a given page in RO nodes' local cache.

```
int  page_register(PageID page_id,
                   const Address local_addr,
                   Address& remote_addr,
                   Address& pl_addr,
                   bool& exists);
int  page_unregister(PageID page_id);
int  page_read(const Address local_addr,
               const Address remote_addr);
int  page_write(const Address local_addr,
                const Address remote_addr);
int  page_invalidate(PageID page_id);
```

3.1.2 Remote Memory Management. The memory resources required by an instance can be provisioned from multiple memory nodes. The unit of memory allocation is a *slab*, and the size of each slab is 1 GB.

Page Array (PA) Each slab is implemented by a PA data structure. A PA is a contiguous piece of memory containing an array of 16 KB pages. When a memory node boots, all memory regions where PAs are located will be registered to the RDMA NIC, so that the page stored in PAs can be accessed directly by remote nodes through one-sided RDMA verbs.

The memory node that serves slabs is also called a *slab node*. A slab node can hold multiple slabs, and the memory resources of each instance are distributed over one or more slab nodes. When each instance is created, the DBaaS will reserve all the slabs that the instance required according to its predefined buffer pool capacity. The slab node where the first slab is located is assigned as the *home node*. Compared with ordinary slab nodes, the home node contains some additional instance-wise metadata:

Page Address Table (PAT) PAT is a hash table that records the location (slab node id and physical memory address) and reference count of each page. In *page_register*, the home node determines whether a page already exists by looking up the PAT table. If it does not exist, an entry will be added to the PAT table after the page is allocated in the remote memory pool. In *page_unregister*, when a page's reference count drops to 0 and is evicted from the remote memory pool, the corresponding entry in the PAT will be deleted.

Page Invalidation Bitmap (PIB) PIB is a bitmap. For each entry in the PAT table, there is an invalidation bit in PIB. Value 0 indicates that the copy of the page in the memory pool is of the latest version, while value 1 means that the RW node has updated the page in its local cache and haven't written it back to the remote memory pool yet. There is also a local PIB on each RO node, indicating whether each page in the RO node's local cache is outdated.

Page Reference Directory (PRD) PRD is another mapping table. For each entry in the PAT table, there is a list of database nodes tracked in the PRD, which means that these nodes have obtained a reference of this page through calling *page_register*. PIB and PRD are used together to achieve cache coherency, see Section 3.1.4.

Page Latch Table (PLT) PLT manages a page latch (PL) for each entry in the PAT table. PL is a global physical lock protecting and synchronizing read and write on pages between different database nodes. Especially it is used to protect the B+ Tree's structural integrity when multiple database nodes access the same index concurrently, see Section 3.2.

Here is the process of how an instance allocates a page from the remote memory: The database process sends a *page_register* request to the home node. If the page doesn't exist, the home node scans all existing slabs to find the one which has the most free memory available. If there is no free memory in any slab, it looks for pages with a reference count of 0, which can be evicted by the LRU algorithm. Because the storage supports page materialization offloading, even dirty pages can also be evicted instantaneously without flushing back. The home node then writes page location into the PAT and returns the remote addresses of the page and corresponding PL to the caller. During the process of page allocation, there is no interaction between the home node and slab nodes for efficiency. As traditional databases, there is a background thread that periodically evicts pages and keeps slab nodes having free pages to avoid evicting pages in the foreground.

When the user expands the buffer pool size elastically, the home node will request the DBaaS to allocate new slabs, and expand the buffer pool size of the instance and the metadata like PAT, PIB and PD accordingly. On the contrary, if the buffer pool size is shrunk on-demand, the extra memory can be released through the LRU page eviction algorithm. Then pages are migrated in the background to defragment, and unused slabs are released finally.

3.1.3 *Local Cache.*

Due to the expensive delay brought by network communication, the database process can't directly read or in-place modify the page data in remote memory like accessing local memory, otherwise the CPU will spend a lot of cycles stalling and waiting for the data to be read from the network to fill in the CPU cache line. It's necessary to read data from the remote memory to the local cache in units of pages, and then directly manipulate the pages in the local cache like traditional monolithic database. The size of local cache is tunable and empirically set to $min\{\frac{1}{8} * Size_{remote_memory}, 128GB\}$, because this ratio provides a balance between performance and cost (e.g., less than 30% performance drop in benchmarks like sysbench, TPC-C, TPC-H). In the future, we plan to dynamically adjust local cache size according to custom workloads and hit ratio [41].

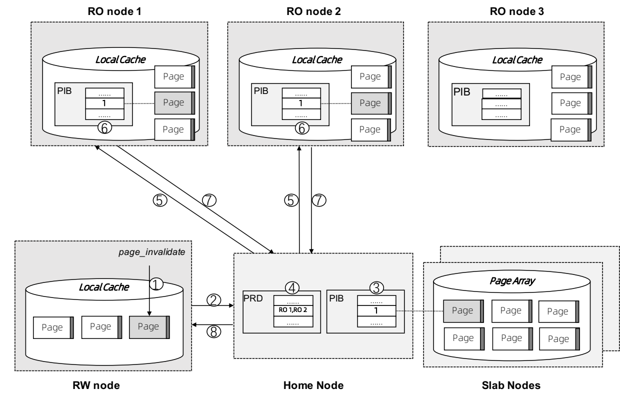


Figure 6: Cache Invalidation

If an accessed page does not reside in the remote memory, the database process reads the page from PolarFS to the local cache through *libpfs*, and then writes it to the remote memory using *librmem*. There is no direct communication between memory nodes and storage nodes. Not all pages read from PolarFS need to be written into the remote memory, for example, when the database process is performing a full table scan. Because the page load by the full table scan is unlikely to be read again in the near future, it would squeeze out other pages cached in the remote memory and pollute the cache.

When a local cache miss occurs, the database process needs to wait for pages to be fetched from remote memory or remote storage, which is much slower than local memory/storage. Therefore, enhancing the hit ratio of local cache through prefetching is the key to improve performance. Detailed description is in Section 4.2.

When the local cache is full, pages need to be evicted. Page eviction uses the LRU algorithm. Clean pages can be freed directly, and modified pages need to be written back to remote memory before they can be released. Besides, the database process should call *page_unregister* to decrease the reference count of the page.

3.1.4 *Cache Coherency.*

In PolarDB, the buffer pool of database processes are private to themselves. A RO node cannot directly access pages with the latest version in RW's buffer pool. Therefore, the RO node captures updated versions of pages by replaying redo logs to its buffer pool continuously, see Section 2.1. In *PolarDB Serverless*, RW can write back modified pages to the remote memory, which can be seen instantaneously by other RO nodes. Thus RO nodes no longer need to replay redo logs. However, in order to reduce the overhead of network communication on modifying a page, RW's changes to a page will be buffered in the local cache and will not be synchronized to the remote memory immediately. What's more, RO nodes may also have copies of pages held in the local cache. This requires a synchronization mechanism. After RW modifies a page in its local cache, the remote memory and other RO nodes that cache this page should know that their copies are outdated. This mechanism is called *cache invalidation*, which ensures cache coherency.

Figure 6 illustrates how cache invalidation works, which is implemented based on the PIB and PRD data structures described in Section 3.1.2. After RW updates a page in its local cache, it calls

page_invalidate ①. This function will set the corresponding bit in PIB on the home node ③, look up PRD to get a list of RO nodes (RO node 1 and 2 in this example) which hold copies in their local cache ④, and then set the corresponding bit in PIB on those RO nodes ⑥. *page_invalidate* is a synchronous blocking operation. Only when PIBs on all participants are set, can it return successfully ⑦ ⑧. If there is an abnormal RO node which doesn't respond after timeout, DBaaS will be involved to kick off the node out of the cluster to ensure *page_invalidate* succeed.

A transaction is divided into multiple mini-transactions (MTR), which is a group of continuous redo log records. Before redo logs in a MTR are flushed to PolarFS, all modified pages in the MTR must be invalidated using the *page_invalidate* interface. This can ensure that there is no such page in the remote memory, and its state is valid, but the version is older than the state of data on the PolarFS (pages plus redo logs). Crash recovery of the database node relies on this property, which is described in Section 5.1.

3.2 B+Tree's Structural Consistency

Concurrency control of B+Trees on a multi-processor system is widely studied, which includes two aspects of work: The first problem is how to guarantee the physical consistency when multiple threads work on the same index simultaneously [10, 15, 25, 30]. If not properly protected, some threads could access a page while another thread is modifying it, resulting in them following stale or dangling links. The second problem is how to maintain the logical consistency and satisfy various snapshot isolation levels when there are concurrent transactions working on the same set of data [29, 32–34].

This section mainly discusses how to solve the first problem in *PolarDB Serverless*, particularly how to update B+tree in RW, while allowing other RO nodes to maintain a consistent view of physical B+tree structure to traverse simultaneously. The second topic will be discussed in the next section.

In *PolarDB Serverless*, only RW can modify pages, so there is no need to protect write conflicts caused by modifications from multiple nodes at the same time. However, Structure Modification Operations (SMO) will modify multiple pages at a time, which may cause RO nodes to see an inconsistent physical B+tree structure when traversing the tree. For example, RW splits a leaf node A into two nodes, A' (previously A) and A'' (new allocated), and then insert it into the parent node, changing B to B'. But when a RO node traverses root-to-leaf concurrently, it may see the parent node B (before the split) and the node A' (after the split), therefore the node A'' and data inside it will be missed.

We solved this problem using PL. PL is the global physical latch with two locking modes: S and X. It does not replace the local page latch which is used to synchronize modification on a single node, instead it is an add on to make sure the integrity of index structure in a multi-node environment, and it works with algorithms like crabbing/lock coupling. All pages involved in SMOs will be X-locked with PL until the SMO is completed. Conversely, all reads on RO need to check with PLT if the page being read is X-locked, and place an S-lock on the page being read. For the example in the previous paragraph, during a root-to-leaf traversal, RO will first add a S-lock to the parent node B, then add a S-lock to the leaf node A, at last release the S-lock of the B node. For RW insert/delete operations, *PolarDB Serverless* always adopt two step approaches. First it will do an optimistic tree traversal for insert/delete, assuming no SMO required. In such case, only local latches are required. And if indeed no SMO is required, the data will be directly inserted or deleted on the leaf page. If the optimistic traversal finds the leaf page is relatively full or empty and a SMO is possible, then it will restart a "pessimistic" traversal from the root again, now it starts placing X latches as well as X-PL locks on all nodes that could be possibly involved in the SMO. In above example, RW needs to add a X-lock to B first, then add a X-lock to A, these locks won't be released until the split is complete. The lock ordering can ensure that RO either see the B+tree structure before SMO (A and B) or the structure after SMO (B', A', A''), because the X-locks on A and B can be regarded as a barrier.

To reduce the PL lock cost, PL has a "stickness" property, which means that it does not require to be released right after SMO as long as there is no request from other RO nodes. The benefit of this property is that the node does not need to request the lock again from the central PLT lock manager for next request if it still holds this lock.

To speed up the PL-acquisition operation, we first try to use RDMA CAS [42] operation to acquire the lock (the address of a page's PL is returned together in page_register). If the fast path fails, for example, trying X-lock a PL that has been S-locked, which may occur when splitting the root node, then get the lock through negotiation among the home node and database nodes. The caller requesting the lock needs to wait for the negotiation to complete.

3.3 Snapshot Isolation

PolarDB Serverless provides SI (snapshot isolation) [14] based on MVCC. Same as InnoDB's Implementation of MVCC, previous versions of a record are constructed with undo logs. In *PolarDB Serverless*, a transaction relies on a snapshot timestamp to control what version of records the transaction is allowed to see. It maintains a centralized timestamp (sequence) named *CTS* in RW, to allocate a monotonic increasing timestamp for all database nodes. A read-write transaction needs to acquire the timestamp from *CTS* twice, once at the beginning of the transaction (*cts_read*), and the other time at the commit time (*cts_commit*). When the transaction commits, it writes down *cts_commit* together with the records it modified. All records and undo records reserve a column to store *cts_commit* of the transaction which modified it. Reads within a transaction always return the value of the most recent version of records whose *cts_commit* is smaller than the transaction's *cts_read*. Each version of record has a field to store the *trx_id* of the transaction that modifies it, with which each transaction could recognize its own writes. A read-only transaction only needs to get a *cts_read* timestamp once at the beginning of the transaction.

However, for large transactions, *cts commit* cannot be updated immediately for all modified rows, which incurs a large number of random writes when the transaction commits. Thus, *cts_commit* columns need to be filled asynchronously in the background. This approach leads to the problem that concurrent transactions cannot determine the version of rows that are not yet filled with *cts_commit*. This problem is solved by looking up the CTS Log data structure on RW. The CTS Log is a circular array, which records

the cts_commit timestamp of the most recent read-write transactions (for example, the last 1,000,000). If the transaction has not yet been committed, the value of its commit timestamp is null. When any database node reads a record or undo record with missing cts_commit, it can look up the CTS Log to determine whether the transaction has been committed, or whether the committed transaction is visible to current transaction.

Obtaining the timestamp and determining whether the record is visible are high frequency operations in transaction processing. We use one-sided RDMA verbs extensively to optimize the timestamp acquisition and the CTS array access. The CTS timestamp counter is fetched and incremented atomically using RDMA CAS [42]. In addition, the CTS Log is placed in a contiguous memory region registered to RDMA NIC, so that RO nodes can efficiently lookup it through one-sided RDMA read. Compared with using RPC, it achieves superior performance without taking up RW's CPU resources, which prevents the RW node from becoming a bottleneck.

3.4 Page Materialization Offloading

Traditional monolithic databases periodically flush dirty pages to durable storage. However, it introduces a large amount of network communications between RW, memory nodes and PolarFS in *PolarDB Serverless*, and affects the performance of high-priority operations in the critical path, such as log writes and page reads. In order to mitigate network bottleneck, Aurora [43] proposed the concept of "log is database". It treats redo log records as a series of incremental page modifications, generates the latest version of pages by applying redo logs continuously to the pages on the storage nodes. Socrates [2] further evolved on this basis, separating log from storage. Logs are first persisted to the XLOG service, and then asynchronously sent to a group of page servers. Each page server is responsible for a database partition, independently replays the logs, generates pages and serves GetPage@LSN requests.

PolarDB adopts a similar approach, which is closer to Socrates. We extend PolarFS such that logs and pages are separately stored in two types of chunks (i.e., log chunk and page chunk). Redo logs are first persisted to log chunks and then asynchronously sent to page chunks, where logs are applied to update pages. The difference against Socrates is that, in order to reuse PolarFS components and minimize changes, logs are sent only to the leader node of the page chunk, who will materialize pages and propagate updates to other replicas through *ParallelRaft*. This method adds additional latency to the ApplyLog operation due to the replication cost. However, it is not a critical issue because ApplyLog is an asynchronous operation not in the critical path. Moreover, since the replicated state machine guarantees data consistency between page chunks, there is no need for an extra gossip protocol among storage nodes like in Aurora.

As illustrated in Figure 7, a storage node in PolarFS can host multiple log chunks and page chunks at the same time. Before a transaction commits, RW flushes changes of redo log files into log chunks. After data written is replicated to three replicas, the transaction can be committed ①. After that, RW decomposes changes of redo logs into log records. According to the pages involved in each log record, a log record is only sent to the corresponding page chunks, whose partition will be affected by the log record ②. Each

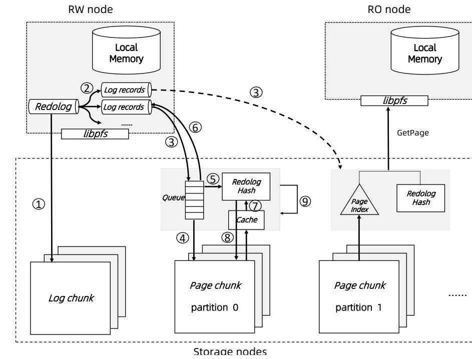


Figure 7: Page Materialization Offloading

page chunk maintains a partition (about 10 GB) of the database, and only receives relevant log records.

After the page chunk's leader node receives log records from RW ③, it immediately persists these records to replicas to ensure durability ④, and inserts them to a in-memory hash which uses the page id as the key ⑤, and then acknowledges RW ⑥. At this point, dirty pages in RW can be evicted. Unless step ⑥ completes, RW always holds references of the dirty pages in its local cache, so that the dirty pages won't be evicted on the remote memory either.

In the background, the old versions of pages are read from the cache or the disk, which are merged with the log records inside the memory hash ⑦. After that, the new version of the pages are written to a new location to replicas ⑧. The underlying storage will hold multiple versions of the page for a period of time, and support fast point-in-time recovery. GC is performed to collect redo logs and old versions in the background ⑨. When serving a *GetPage* request, the leader node merges the version in the cache/disk, and log records on the page in the memory hash, generates the latest page and returns it to the database node.

3.5 Auto-Scaling

The capability of transparent and seamless scaling without interrupting customer workloads is essential for serverless databases. Ideally, even when the database reboots or migrates across compute nodes, applications should not be aware of any exceptions, such as database disconnections, aborted transactions and unacceptably long response time. In *PolarDB Serverless*, when a switching event occurs (e.g., version upgrade or cross-node migration), the proxy node is responsible for keeping client connections alive. It pauses active transactions and stops forwarding subsequent statements to the old RW node. It then waits for a short period (e.g. 100 ms), which is long enough for the old RW node to complete most of in-progress statements. After that, the old node flushes all dirty pages to the shared memory and shuts down itself. Meanwhile, the new RW node will attach to the shared memory and initialize its local memory state (e.g., scanning the undo header to rebuild the active transaction list). At last, the proxy connects to the new RW node, restores its session states, and then forwards pending statements for execution. Note that long-running statements that have not been completed by the old RW node will be reissued after the new RW node rolls them back.

For long-running multi-statement transactions, such as bulk insertions, the proxy also tracks a savepoint for each statement.

The savepoint indicates the execution progress of the transaction, i.e., the *i*-th statement since the beginning of the transaction. As a result, when the switching event happens, the proxy can inform the new RW node to resume execution from the latest savepoint, which avoids rolling back the entire transaction and significantly improves migration transparency.

Recall that during the switching process, active transactions need to be paused until the new RW node takes over the service. It is hence critical to shorten the pause. In *PolarDB Serverless*, transferring transaction states (e.g. dirty pages and rollback segments) through shared memory is faster than the traditional way that relies on remote storage. Other optimizations such as statement-level rollback are also helpful. Overall, we can reduce the pause time to 2-3 seconds (as shown in Section 6.3). We plan to investigate more optimizations as future work, such as putting transaction locks in shared memory to support multiple RW nodes, and caching intermediate results in shared memory to speedup analytical query resumption.

4 PERFORMANCE OPTIMIZATIONS

The evolution of the disaggregated architecture could have a negative impact on the database performance, since data is likely to be accessed from the remote-memory nodes and storage nodes, involving notable network latency. To address this problem, we employ a suite of optimizations to minimize the chances of compromise.

4.1 Optimistic Locking

In Section 3.2, we describe that the RO node uses the lock coupling protocol to acquire PL locks on the pages during the traversal from root to leaf. Even if RW performs SMO at the same time, RO can still see a consistent B+ tree physical structure. Although in most cases, the acquisition of PL can be successfully completed through the fast path of RDMA CAS [42], it may still degenerate into multiple rounds of negotiations. In this section, we propose an Optimistic Locking mechanism. It is optimistic to assume that the RO node will not encounter SMO during a root-to-leaf traversal, so there is no need to obtain any PL. When SMO is detected, try again or fall back to the pessimistic strategy. SMO can be detected like this. A SMO counter is maintained on RW, namely SMO_{RW}. Whenever SMO occurs in the B-tree, the counter is incremented by one. When SMO occurs, the snapshot of SMO_{RW} (SMO_{page}) is recorded on all pages modified by SMO. At the beginning of each query, the snapshot of the SMO counter is obtained as SMO_{query}. When the RO node executes this query and traverses from root to leaf, it stops if it finds that SMO_{page} of any page on the path is greater than SMO_{query}. It means that SMO occurs during the query, so that it may read an inconsistent B+tree.

4.2 Index-Awared Prefetching

Some systems implement data prefetching strategies [31, 44] at the operating system and storage levels, which are agnostic to the database workloads. However by checking the SQL plan generated in the database kernel, we can use such information to accurately predict which blocks will be accessed next and prefetch them in advance. In *PolarDB Serverless*, we propose *Batched Key PrePare* (BKP). BKP prefetches pages containing interesting tuples from disaggregated memory and storage to hide remote I/O latency. Similar prefetching methods has also been used in hash join to avoid CPU cache misses [11].

There are a large number of daily database operations that access secondary indexes. For example:

```
select name, address, mail
from employee where age > 17
```

In this example, the field *age* has a secondary index, but the other fields *name*, *address*, and *mail* must be read from the primary index. When MySQL executes this statement, The typical process is to first scan a batch of qualified record primary keys from the secondary index and then read other fields from the base table. The first step is a sequential traversal that can be sped up by pre-reading continuous B+ tree leaf nodes. And the second step is likely to be a series of random accesses, and can be optmized by BKP.

We implement BKP in our storage engine, the interface of BKP accepts a group of keys to be prefetched. When the interface is called, the engine will start a background prefetching task, retrieves required keys from the target index and fetches corresponding data pages from the remote memory/storage when necessary.

BKP can also optimize analytical workloads. For TPC-H queries, for example, many of the joins are equal join with indexes, especially for very large tables. When dealing with these huge tables or the table data is cold, a large number of pages may need to be fetched from the remote. BKP can be used to speed up equal joins that access the inner table with index. In MySQL, there is a join buffer which accumulates the interesting columns of rows produced by the left join operand. Tables from earlier joins are read in portions into the join buffer, when the number of rows in the join buffer reaches a threshold, BKP will build and send a batch of keys to the storage engine to prefetch pages of the inner table. Concurrently, the storage engine prefetches the pages containing these keys in the background. When the join buffer is full, the rows in the buffer will be joined with the inner table, at this point, we expect that most pages needed should have already be loaded into the cache.

5 RELIABILITY AND FAILURE RECOVERY

In *PolarDB Serverless*, the *disaggregated* architecture allows nodes of each type to failover independently. Therefore, we tailor the failure recovery method for each node type. Most recovery methods follow approaches in existing *PolarDB* design. All proxy nodes in *PolarDB Serverless* are stateless. When a proxy node fails, it can be easily replaced. User connections can reconnect to other alive nodes. *PolarDB Serverless* maintains at least 2 replicas for each storage node (i.e., 3 replica in total). Each group of replicas is managed by the distributed consensus protocol, *Parallel Raft*. Overall, our design guarantees that there is no single-point failure in the system.

In the rest of this section, we focus on more complex mechanisms that deal with database node recovery, memory node recovery and cluster recovery, respectively.

5.1 Database Node Recovery

PolarDB Serverless adopts an ARIES-style recovery algorithm [33]. RW and RO nodes have different recovery procedures. A failed RO node can be easily replaced with a new one using pages in the shared memory. Based on whether the node failure is planned or not, the recovery process is different.

Unplanned Node Failure When the RW node fails, the Cluster Manager (CM) detects this event via heartbeat signals (usually working at 1Hz) and initiates a RO node promotion process. The steps are as follows:

(1) CM notifies all memory and storage nodes to refuse subsequent write requests from the original RW node.
(2) CM selects a RO node (RW') and informs it to make the promotion.
(3) RW' collects the latest version (LSN_{chunk}) from each PolarFS page chunk, and gets $min\{LSN_{chunk}\}$, which is the checkpoint version LSN_{cp}.
(4) RW' then reads the redo log records persisted on PolarFS log chunks, starting from LSN_{cp} to the end of redo log LSN_{tail}, and distributes them to page chunks, waiting for page chunks to consume these redo records and complete the recovery.
(5) RW' scans the remote memory pool, evicts pages whose invalidation bit is 1, and pages whose version (LSN_{page}) is newer than the version of redo (LSN_{tail}).
(6) RW' releases all PL locks obtained by the original RW node.
(7) RW' scans undo headers to construct the state of all active transactions at the moment the original RW node fails.
(8) RW' is ready to receive write requests, after it notifies CM the promotion completes RW' becomes the new RW node.
(9) RW' plays undo logs to rollback uncommitted transactions in the background.

Step (3) and (4) are performing the REDO phase in the ARIES-style recovery procedure [33]. The difference is that REDO is no longer executed on a single machine, but concurrently executed on many page chunk nodes, which speeds up the REDO phase. Step (5) evicts the pages from the remote memory, whose version is inconsistent with the stored versions in the remote storage, which relies on the property described in Section 3.1.4: the version of a page with the invalidation bit of 0 in the remote memory cannot be lower than that in the storage. On the other hand, because RW may already write back dirty pages to the remote memory before the redo logs are flushed to PolarFS, some pages in the remote memory may be more advanced than the persisted state in PolarFS. Therefore, these pages also need to be cleared. After step (3) (4) (5), redo logs in PolarFS, pages in PolarFS, and pages in the remote memory are completely consistent. In step (6), because RW uses PL locks to protect the SMO and all page modifications of a SMO are packed into one MTR, all modifications of a SMO are written as an atomic operation. After step (3) (4) (5), the state of the database is restored to a consistent status, so there is no SMO in progress at this time, thus all PL locks can be released safely. Since most active data pages still reside in the remote memory pool, the *cold cache* issue in traditional master-slave replication architecture is prevented. Note that parallel redo recovery is a well known technique [8, 22, 43]. However, integrating it with disaggregated memory and storage requires nontrivial engineering effort.

Planned Node Down When the RW node is planned to be down, it will perform a series of cleanup work to reduce the workload of the new RW node to take over. For example, synchronize redo logs to the page chunk, actively release all PL locks, and write dirty pages back to remote memory, finally flush redo logs to PolarFS. In this way, the new RW node can save step (4) (5) (6) when recovering. In addition, the new node can choose to delay its promotion until the moment when the number of active transactions is low, which will further reduce the cost of step (7) and (9). Together with techniques discussed in Section 3.5, operations such as upgrades and migrations will have less impact on the application.

5.2 Memory Node Recovery

Memory nodes caches buffered pages used by databases. Logs are always flushed to the storage before dirty pages are written to memory nodes. Therefore when a memory node is down, the buffered pages can always be recovered from the storage.

Recall that the memory node containing the first slab is called home node (Section 3.1). This node contains the critical control metadata such as PAT, PIB, PRD and PLT. The control metadata is essential to the database normal activities, since it contains critical information to ensure cross-node consistency. Under this consideration, home node's metadata is also backed up in another "slave replica" in a synchronized manner. The home node is responsible for detecting slab node failures. In the case of slab node failures, the home node will go over the PAT and process all buffered pages originally registered under those slab nodes. These buffer pages will be re-registered by fetching them from RW's local cache or just calling page_unregister to remove them.

5.3 Cluster Recovery

In rare cases, when all replicas of the home node are unavailable, the service needs to be recovered through cluster recovery. All database nodes and memory nodes will be restarted from the cleared state, and all memory states are rebuilt from storage. After initialization (attach to remote memory and storage, etc.), the RW node performs the parallel REDO recovery as described in Section 5.1, and then scans the undo header to find all uncompleted transactions. After that, the RW node starts services and rollbacks uncommitted transactions in the background. In the cluster recovery, the pages cached in the remote memory is cleared, so it will endure the *cold cache* problem.

6 EVALUATION

6.1 Experiment Setup

In this section, we evaluate the elasticity, availability, and performance of *PolarDB Serverless*. For performance evaluation, we use standard benchmarks including sysbench [23], TPC-C [12] and TPC-H [37] for both transactional and analytical workloads. We deploy database instances in dockers of version 17.06.2 and use cgroup to vary the configuration from 8 CPUs to 32 CPUs, and 8 GB to 500 GB memories. These dockers are deployed in a cluster containing 32 machines, connected by a high-speed RDMA network using Mellanox ConnectX-4 network adaptor. The OS we use is Linux 3.10.

6.2 Elasticity of the Disaggregated Memory

Elasticity is one of key advantages of *PolarDB Serverless*. In Figure 8, we report the overall throughput (sysbench oltp_read_only with range select) of a *PolarDB Serverless* instance. In 200, 350, 500 seconds, the remote memory size of the instance increased from

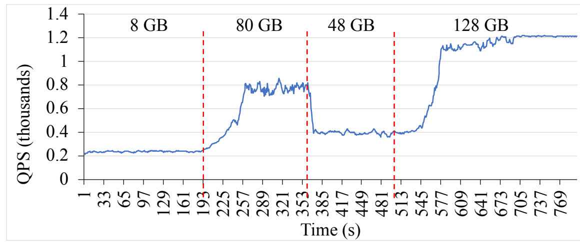


Figure 8: Throughput of *PolarDB Serverless* while scaling out-/in the remote memory (i.e., 8GB, 80GB, 48GB, 128GB) while processing range queries.

8GB to 80GB, 48GB and 128GB respectively. This test simulates the scenario where customers use *PolarDB Serverless* on a pay-as-you-go basis. It can be seen from the measured throughput that after each memory expansion operation, the performance gradually improves, because the newly added slabs take some time to be warmed. After each memory shrink operation, the performance will drop immediately, as slabs and pages are removed from the remote buffer pool at once. With such elasticity, customers can expand remote memory to almost unlimited capacity, and only pay for the necessary resources for achieving performance targets.

6.3 Fast Failover

Figure 9 shows the trend of throughput during a RW node is recovering from either a planned switch (switching RW during auto-scaling) or an unplanned failure. It takes only 6 seconds (pause for 2s and another 4s to resume throughput) for the RW in *PolarDB Serverless* to fully recover from a planned switch. For unplanned failures, we compare the recovery of *PolarDB Serverless* in different cases: (1) with the remote memory and all optimizations (blue line), (2) without the remote memory but with page materialization (yellow line), and (3) without page materialization (orange line). The remote memory is able to keep hot pages despite the switch/crash, which avoids costly page reloading during recovery. Therefore, it only takes 16 seconds for the blue line to resume the service and another 7 seconds to reach its previous performance (i.e. 90% of the peak performance before the crash). Without the remote memory, although the RW can read materialized pages from the storage, it takes extra time (12 seconds longer) to warm up the buffer pool. When materialized pages are not available, the RW has to apply the redo log and reconstruct pages, taking 5.3 times longer time (or 85 seconds) to resume the service. Finally, it spends a total of 105 seconds to reach its previous performance.

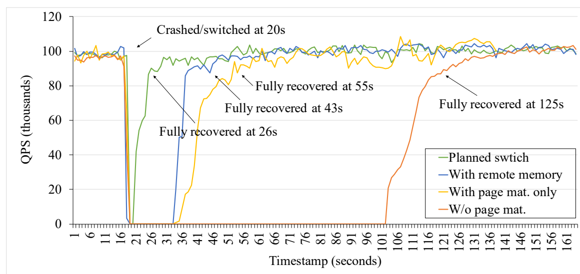
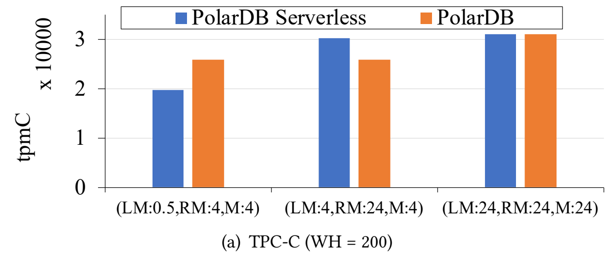
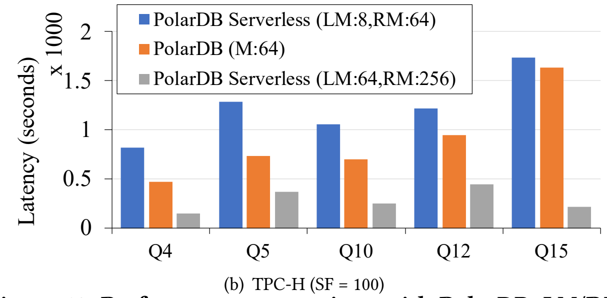


Figure 9: Recovery time for the RW with shared memory or local memory



(a) TPC-C (WH = 200)



(b) TPC-H (SF = 100)

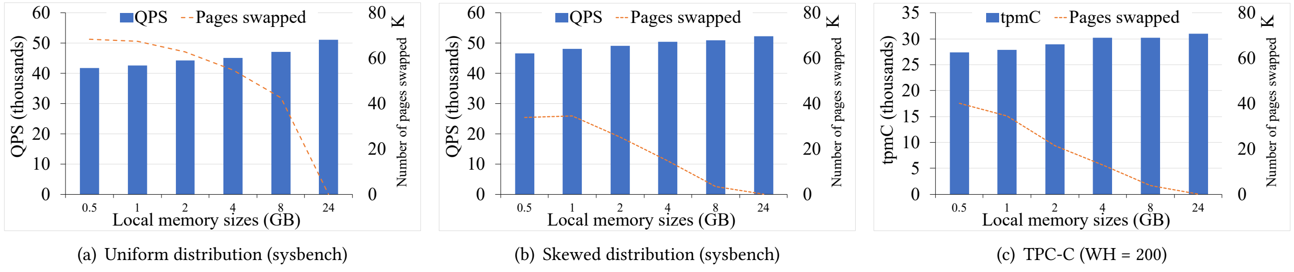
Figure 10: Performance comparison with PolarDB. LM/RM: local/remote memory size of PolarDB Serverless (GB), M: memory size of PolarDB (GB).

6.4 Performance Evaluation

Performance comparison with PolarDB. In Figure 10(a) and 10(b), we compare the performance of *PolarDB Serverless* and PolarDB using TPC-C and TPC-H, respectively. In these figures, *M* represents the memory size for PolarDB, *LM* and *RM* are local and remote memory size for *PolarDB Serverless*, *DS* is the data size of TPC-C/TPC-H workloads. The unit of memory size is GB.

We first compare the performance of TPC-C under three different pairs of configurations. (1) When PolarDB's memory size and *PolarDB Serverless*'s remote memory size are configured with the same size (4GB), which are smaller than the entire working set size (20GB). In this case, we observe that the performance of PolarDB is better because accessing local memory is faster than remote memory. (2) When we increase *PolarDB Serverless*'s local memory size to 4GB, and PolarDB's memory size and *PolarDB Serverless*'s local memory size are equal, the performance of *PolarDB Serverless* is superior to PolarDB since accessing data from shared memory is faster than from shared storage. (3) Finally, when both PolarDB's buffer pool and *PolarDB Serverless*'s local memory are large enough (24GB) to hold the entire dataset, their performance is close. Next, we test the performance of TPC-H under two similar scenarios where PolarDB's memory is configured as same as *PolarDB Serverless*'s local or remote memory size. We collect similar observations as above test. In summary, *PolarDB Serverless* offers cloud customers a more elastic (scalable and tunable) way of using databases at a reasonable performance loss, compared with the conventional PolarDB. In the following, we show that the performance of *PolarDB Serverless* scales well when more resources are provisioned.

Effect of local memory size. We evaluate the performance effect of the database node's local memory capacity using sysbench, TPC-C and TPC-H respectively. These results demonstrate that the performance penalty of memory disaggregation is acceptable and tunable.



(a) Uniform distribution (sysbench) (b) Skewed distribution (sysbench) (c) TPC-C (WH = 200)

Figure 11: Performance of mixed reads and writes with varying local memory sizes.

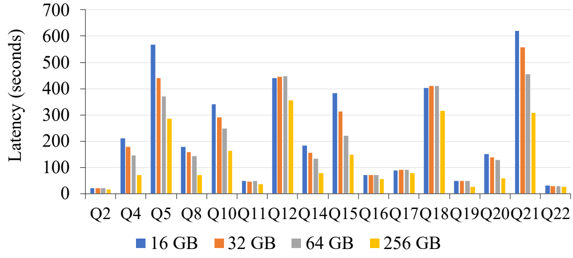


Figure 12: Latency of TPC-H queries (SF = 100) with varying capacities of the local cache (16 GB to 256 GB)

Figure 11(a), 11(b) and 11(c) show the throughput of mixed reads and writes in addition to the number of swapped pages. We test on sysbench *oltp_read_write* benchmark using uniform and skewed[2] distributions, as well as TPC-C queries. In all cases, we vary the local memory size from 0.5GB to 24GB. From Figure 11, we make three observations: (1) The performance losses caused by page swapping between local and remote memory are at most 18.5%, 10.7% and 13.4% in three cases when the local memory is as small as 0.5GB. (2) Since hot data pages are cached on the database node as the local memory capacity increases, the total amount of data transferred from memory nodes to database nodes decreases significantly, and meanwhile the system performance greatly increases. The throughput increases gradually until the local memory is large enough to hold the entire working set, in which case there is no page swapping between local memory and remote memory. (3) As the local memory size increases, the performance curves of skewed distribution and TPC-C workload are more smooth than that of uniform distribution due to higher hit ratio of local memory for hot pages.

Figure 12 shows the measured latency of TPC-H queries with different local cache size varied from 16 GB to 256 GB. The data size of TPC-H workloads is configured to SF=100 (200GB in storage). We observe that the performance curve with the increase of local memory size is steeper than that of sysbench. The reason is that the data size of TPC-H is larger than that of sysbench, which introduces more remote memory accesses with the same local memory size. Compared to 10 GB local memory, the latency could be reduced by 40.9% when using 256 GB local memory to hold the entire working set.

Effect of remote memory size. Next, we analyze the effects of shared memory size on the overall system performance. Figure 13 evaluates the latency of TPC-H queries with different capacities of the shared remote memory varied from 32 GB to 256 GB. The local cache size is fixed to 8 GB. We observe that the capacity of shared memory can have a significant effect on performance. As the memory size grows from 32 GB to 128 GB, about two-thirds of TPC-H queries achieve speedup by 3.06x on average. This indicates that *PolarDB Serverless* can provide higher performance by expanding shared memory capacity. Note that the rest (Q2, Q11, Q16, Q17, Q19) are not sensitive to the memory capacity, since their execution time is relatively short (all less than 100 seconds) and 32GB memory seems large enough for them.

Effect of optimistic locking. In Figure 14, we run sysbench-read-write (both uniform and skew distributions) on a *PolarDB Serverless* instance with one RW node and another RO node. Proxy will redirect all writes to the RW node and balance the read requests between RW and RO. We compare the total read throughput using either the optimisitic lock (Olock) or not (Plock) when increasing the number of client threads. When the number of concurrent threads increases from 32 to 128, Plock loses almost 50% of its total QPS, while Olock only loses up to 10%. It shows that under high concurrency, optimistic locking significantly improves read performance by reducing the cost of acquiring PL locks on RO nodes.

Effect of prefetching. Figure 15(a) and 15(b) evaluate the effects of prefetching on the measured response times of TPC-H queries, when initially placing all the data in the remote memory or the storage, respectively. In Figure 15(b), we turn off the remote memory to force cache-missed pages to be loaded from the storage, so that we can evaluate the effect of prefetching for remote storage. We choose these queries because joins in them involve retrieving a lot of records from large tables (i.e., lineitem and orders) after the corresponding equi-join conditions are evaluated, where BKP is supposed to be effective. We make two observations: (1) The latency gradually decreases at 25.4% and 52.3% on average in two cases, because remote access overhead can be reduced by data prefetching. (2) The benefit from prefetching on remote storage is higher than that on remote memory, because the I/O latency of remote storage is higher than remote memory.

7 RELATED WORK

Databases with decoupled compute and storage. Cloud-native databases like Aurora [43], Socrates [2] and PolarDB build multi-tenant database services on top of a shared storage pool. Aurora offloads page materialization downwards to the shared storage. Socrates transforms the on-premise SQL Server into a DBaaS and

[2]The skewed distribution is generated by the sysbench oltp_read_write benchmark with setting “rand-type=default”, which uses about 5% of all key IDs as hot data in generated queries.

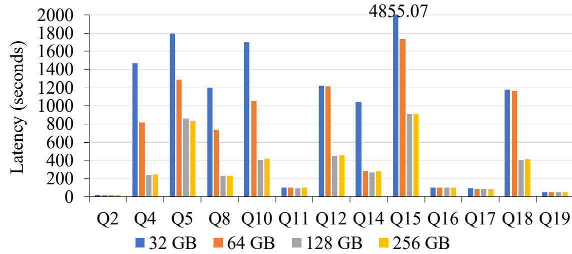


Figure 13: Latency of TPC-H queries (SF = 100) with varying capacities of the remote memory (32 GB to 256 GB)

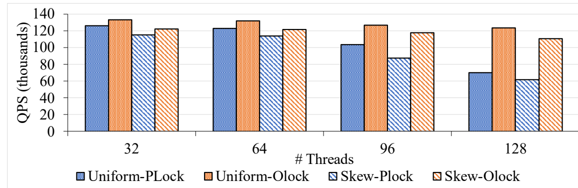
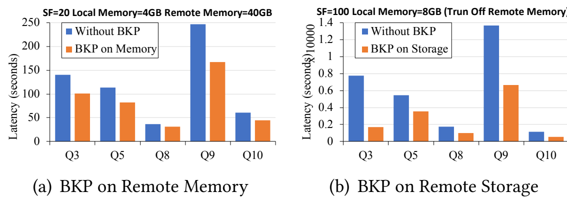


Figure 14: Optimistic Locking vs Pessimistic Locking



(a) BKP on Remote Memory (b) BKP on Remote Storage

Figure 15: Performance Effect of Index-Awared Prefetching for Remote Memory & Storage.

further separates logging and storage from database kernel to dedicated services. However, tightly coupled CPU and memory resources still suffer from low resource utilization, limited elasticity and fate sharing.

Databases with remote memory. Substantial advances in high-speed network push forward the development of shared-memory (or shared-cache) database cluster. Oracle RAC proposes the Cache Fusion technique [24] that allows nodes to share the local cache with data coherence through high-speed inter-node messaging. However, since RAC is built on distributed cache and lock management, the complexity of inter-node coordination grows exponentially as the number of nodes increases. IBM's DB2 pureScale [6, 21] utilizes centralized lock management to directly access global cache using RDMA, which effectively simplifies read-sharing and write-sharing policy. It further adopts MVCC [29, 34] to permit that queries can read a slightly old but still consistent version of a record to avoid locking. However, the maximum global cache capacity of pureScale is limited by the total memory of dedicated cluster caching facility, lacking of elasticity for on-demand cloud applications. NAM-DB [46] redesigns distribuetd database based on the separated architecture of compute and memory via RDMA. It adopts a optimistic native SI-Protocol, which may results in high abort rates especially when there are hot spots or slow workers.

Databases in disaggregated data centers. Cloud infrastructure providers now offer multi-socket servers with hundreds of cores and tens of TB memory. Prior work [1] focuses on fine-grained elasticity for shared-everything, scale-up OLTP systems deployed in such a multi-core server. [5] analyses in-memory DBMSs on latest multi-socket hardware and further optimizes concurrency control schemes. However, the maximum global memory capacity and CPUs are still limited by a single standalone machine.

LegoOS [38] proposes an intriguing idea of replacing all hardware components in a server with network-linked counterparts in a data center. Prior work [47] evaluates popular databases on LegoOS and concludes that simply placing databases on disaggregated OS suffers significant performance degradation from massive remote memory accesses. To eliminate the limited memory size in a single machine, FaRM [13] and INFINISWAP [16] propose memory disaggregation schemes to provision a global memory pool to all machines. They exploit the modern RDMA network to reduce remote memory access latency by an order of magnitude, compared to that of those systems using TCP/IP. Prior work [27] leverages remote memory via RDMA to accelerate memory-intensive workloads for existing RDBMSs when local memory is insufficient. However, all schemes do not address the inter-node data consistency problem, disallowing memory write-sharing among multiple machines.

GAM [7] is a distributed in-memory computing platform to expose a unified global memory interface. RAMCloud [36] is a log-structured DRAM-based key-value store that keeps all data in DRAM at all times. It is nontrivial to build high-performance databases using GAM or RAMCloud directly. Tailored database designs and optimizations are still open to explore.

Computation pushdown. Numerous studies have proposed methods for near-data processing [19, 20, 40, 45]. The key idea is that data should be processed at the place close to where it is stored to eliminate unnecessary data access and movement. Our past work [8] pushes table scan down from LSM-tree to disaggregated storage, which significantly reduces latency and storage-to-memory data movement volume for TPC-H. This inspires us to integrate similar techniques to the disaggregated memory, which is close to the area of Near-Memory Computing [28, 39].

8 CONCLUSION

This paper presents *PolarDB Serverless*, the first cloud-native database implementation of the *disaggregation* architecture. It is a fork of the PolarDB MySQL/InnoDB codebase. The buffer pool management has been completely rewritten to take advantage of the scalable remote memory pool with a local caching tier. We have introduced the detailed design and optimizations adopted in *PolarDB Serverless*. For future work, we plan to explore offloading more tasks from compute nodes to remote memory to reduce cross-node communication cost, and store transactions state (such as logical locks and intermediate query results) in disaggregated memory to improve auto-scaling transparency.

ACKNOWLEDGMENTS

PolarDB Serverless benefits greatly from our clients. Their expectations for cloud-native databases have driven us to design this disaggregated architecture in this work. The making of PolarDB Serverless is a large project that is not possible without mutual efforts from multiple roles at Alibaba Cloud. We would like to thank Jing Cui, Ming Zhao, Guangqing Dong, Changtong Liu, Zewei Chen, Guangbao Ni, Chao Bi and Jingze Huo for their professional contributions. We also deeply appreciate anonymous reviewers' valuable comments on this paper.

REFERENCES

[1] A.-C. Anadiotis, R. Appuswamy, A. Ailamaki, I. Bronshtein, H. Avni, D. Dominguez-Sal, S. Goikhman, and E. Levy. A system design for elastically scaling transaction processing engines in virtualized servers. Proc. VLDB Endow., 13(12):3085–3098, Aug. 2020.

[2] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, V. Purohit, H. Qu, C. S. Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade. Socrates: The new sql server in the cloud. In Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, page 1743–1756, New York, NY, USA, 2019. Association for Computing Machinery.

[3] AWS. Aurora serverless. https://aws.amazon.com/cn/rds/aurora/serverless/.

[4] Azure. Azure sql serverless. https://docs.microsoft.com/en-us/azure/azure-sql/database/serverless-tier-overview.

[5] T. Bang, N. May, I. Petrov, and C. Binnig. The tale of 1000 cores: An evaluation of concurrency control on real(ly) large multi-socket hardware. In Proceedings of the 16th International Workshop on Data Management on New Hardware, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery.

[6] V. Barshai, Y. Chan, H. Lu, S. Sohal, et al. Delivering continuity and extreme capacity with the IBM DB2 pureScale feature. IBM Redbooks, 2012.

[7] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang. Efficient distributed memory management with rdma and caching. Proc. VLDB Endow., 11(11):1604–1617, July 2018.

[8] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, Z. Liu, F. Zhu, and T. Zhang. POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In 18th USENIX Conference on File and Storage Technologies (FAST 20), pages 29–41, Santa Clara, CA, Feb. 2020. USENIX Association.

[9] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma. Polarfs: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. Proc. VLDB Endow., 11(12):1849–1862, Aug. 2018.

[10] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01, page 181–190, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[11] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. ACM Trans. Database Syst., 32(3):17–es, Aug. 2007.

[12] T. P. P. Council. Tpc benchmark. http://tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf.

[13] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pages 401–414, Seattle, WA, Apr. 2014. USENIX Association.

[14] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. ACM Trans. Database Syst., 30(2):492–528, June 2005.

[15] G. Graefe. A survey of b-tree locking techniques. ACM Trans. Database Syst., 35(3), July 2010.

[16] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 649–667, Boston, MA, Mar. 2017. USENIX Association.

[17] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[18] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, page 651–665, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Z. István, D. Sidler, and G. Alonso. Caribou: Intelligent distributed storage. Proc. VLDB Endow., 10(11):1202–1213, Aug. 2017.

[20] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. G. Lee, and J. Jeong. Yoursql: A high-performance database system leveraging in-storage computing. Proc. VLDB Endow., 9(12):924–935, Aug. 2016.

[21] J. W. Josten, C. Mohan, I. Narang, and J. Z. Teng. Db2's use of the coupling facility for data sharing. IBM Systems Journal, 36(2):327–351, 1997.

[22] Juchang Lee, Kihong Kim, and S. K. Cha. Differential logging: a commutative and associative logging scheme for highly parallel main memory database. In Proceedings 17th International Conference on Data Engineering, pages 173–182, 2001.

[23] A. Kopytov. Sysbench: a system performance benchmark. http://sysbench.sourceforge.net/, 2004.

[24] T. Lahiri, V. Srihari, W. Chan, N. MacNaughton, and S. Chandrasekaran. Cache fusion: Extending shared-disk clusters with shared caches. In Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01, page 683–686, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[25] P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on b-trees. ACM Trans. Database Syst., 6(4):650–670, Dec. 1981.

[26] F. Li. Cloud-native database systems at alibaba: Opportunities and challenges. Proc. VLDB Endow., 12(12):2263–2272, Aug. 2019.

[27] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and rdma. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, page 355–370, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Z. Liu, I. Calciu, M. Herlihy, and O. Mutlu. Concurrent data structures for near-memory computing. In Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '17, page 235–245, New York, NY, USA, 2017. Association for Computing Machinery.

[29] R. A. Lorie, C. Mohan, and M. H. Pirahesh. Multiple version database concurrency control system, 1994.

[30] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

[31] H. A. Maruf and M. Chowdhury. Effectively prefetching remote memory with leap. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 843–857. USENIX Association, July 2020.

[32] C. Mohan. Aries/kvl: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In Proceedings of the 16th International Conference on Very Large Data Bases, VLDB '90, page 392–405, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[33] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst., 17(1):94–162, Mar. 1992.

[34] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, SIGMOD '92, page 124–133, New York, NY, USA, 1992. Association for Computing Machinery.

[35] Oracle. Innodb. https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html.

[36] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. ACM Trans. Comput. Syst., 33(3), Aug. 2015.

[37] M. Poess and C. Floyd. New tpc benchmarks for decision support and web commerce. SIGMOD Rec., 29(4):64–71, Dec. 2000.

[38] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 69–87, Carlsbad, CA, Oct. 2018. USENIX Association.

[39] G. Singh, L. Chelini, S. Corda, A. Javed Awan, S. Stuijk, R. Jordans, H. Corporaal, and A. Boonstra. A review of near-memory computing architectures: Opportunities and challenges. In 2018 21st Euromicro Conference on Digital System Design (DSD), pages 608–617, 2018.

[40] M. Singh and B. Leonhardi. Introduction to the IBM Netezza warehouse appliance. In Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON), pages 385–386, 2011.

[41] J. Tan, T. Zhang, F. Li, J. Chen, Q. Zheng, P. Zhang, H. Qiao, Y. Shi, W. Cao, and R. Zhang. Ibtune: Individualized buffer tuning for large-scale cloud databases. Proc. VLDB Endow., 12(10):1221–1234, June 2019.

[42] M. TECHNOLOGIES. Rdma aware networks programming user manual. http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[43] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17, page 1041–1052, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Y. Wiseman and S. Jiang. Advanced Operating Systems and Kernel Applications: Techniques and Technologies: Techniques and Technologies. IGI Global, 2009.

[45] L. Woods, Z. István, and G. Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. Proc. VLDB Endow., 7(11):963–974, July 2014.

[46] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. Proc. VLDB Endow., 10(6):685–696, Feb. 2017.

[47] Q. Zhang, Y. Cai, X. Chen, S. Angel, A. Chen, V. Liu, and B. T. Loo. Understanding the effect of data center resource disaggregation on production dbmss. Proc. VLDB Endow., 13(9):1568–1581, May 2020.