GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

Wengang Tian
Huawei Company
tianwengang@huawei.com

Jinyu Zhang
Huawei Company
zhangjinyu.zhang@huawei.com

Ronen Grosman
Huawei Company
ronen.grosman@huawei.com

Zongchao Liu
Huawei Company
liuzongchao@huawei.com

Sihao Li
Huawei Company
sean.lisihao@huawei.com

ABSTRACT

Cloud-native databases have been widely deployed due to high elasticity, high availability and low cost. However, most existing cloud-native databases do not support multiple writers and thus have limitations on write throughput and scalability. To alleviate this limitation, there is a need for multi-primary databases which provide high write throughput and high scalability.

In this paper, we present a cloud-native multi-primary database, GaussDB, which adopts a three layer (compute-memory-storage) disaggregation framework, where the compute layer is in charge of transaction processing, the memory layer is responsible for global buffer management and global lock management, and the storage layer is used for page and log persistence. To provide multi-primary capabilities, GaussDB logically partitions the pages to different compute nodes and then assigns the ownership of each page to a compute node. For each transaction posed to a compute node, if the compute node owns all relevant pages of this query, the compute node can process the query locally; otherwise, GaussDB transfers the ownership of relevant pages to this node. To capture data affinity and reduce page transmission costs, GaussDB designs a novel page placement and query routing method. To improve recovery performance, GaussDB employs a two-tier (memory-storage) checkpoint recovery method which uses memory checkpoints combined with on-demand page recovery to significantly improve recovery performance. We have implemented and deployed GaussDB internally at Huawei and with customers, and the results show that GaussDB achieves higher throughput, lower latency, and faster recovery than state-of-the-art baselines.

PVLDB Reference Format:
Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. PVLDB, 17(12): 3786 - 3798, 2024.
doi:10.14778/3685800.3685806

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685806

1 INTRODUCTION

Cloud-native databases have attracted significant attention from both industry and academia due to their high elasticity, high availability, and low cost. Many cloud databases have been launched, e.g. Amazon Aurora [29], Google AlloyDB [12], Microsoft Azure SQL Database [2, 22], Huawei Taurus [9], and Alibaba PolarDB [5]. Many customers have moved their on-premise databases to cloud databases [1–5, 8, 9, 12, 14, 14, 15, 17–21, 23, 23, 26, 27, 29, 32, 34–36, 36–38].

Most existing cloud-native databases adopt a primary-standby architecture where only the primary node supports writes and all other standby nodes can only perform reads (i.e., single writer, multiple readers). Obviously, this primary-standby architecture has limitations on write throughput, scale-out and scale-in, and continuous availability. Thus, there is a need for multi-primary (a.k.a. multi-writer) databases, where each node can write and read, thus providing high write scalability and enhanced availability. Although Aurora [30], Taurus [10] and PolarDB[31] offer the multi-primary capabilities, they have some limitations. Aurora [30] uses the logs and optimistic concurrency control to detect write conflicts, and thus has high abort rates and lower performance. Taurus [10] adopts pessimistic concurrency control for cache coherence, but it suffers from high concurrency control overhead. PolarDB [31] uses a *stateful memory layer* for transaction/buffer/lock fusion where each compute node interacts with the memory layer for transaction processing, but it is inefficient to recover the memory layer.

Generally, there are two classes of multi-primary architectures, shared-nothing architecture and shared-storage architecture. The former physically partitions the data into different shards (e.g. hash, range, list partitions), and it uses two-phase locking to support single shard transactions and two-phase commit to support cross-shard transactions. The example systems include Spanner [6, 11, 33], CockroachDB [28], and TiDB [13]. These systems involve two-phase commit that may lead to low performance and high latency for cross-shard transactions. Moreover, they require specifying the sharding keys (i.e. partition columns and functions) to partition the data into different nodes, but it is difficult to find appropriate sharding keys especially for complex workloads (e.g. ERP or CRM), and those sharding keys may only be optimal for a subset of queries. On the other hand, shared-storage architectures logically partition pages onto different writers, relying on conflict detection and cache coherence mechanisms. The example systems include Oracle RAC [25] and IBM Db2 PureScale [24]. They do not disaggregate compute-memory-storage, and have limitations on scale-out and scale-in abilities [10]. We adopt the shared-storage architecture and extend it to build a cloud-native multi-primary database.

The big challenge for multi-primaryarchitectures is high performance transaction management from multiple nodes (detecting the transaction conflicts, guaranteeing consistency, and achieving fast failure recovery). To address this challenge, we propose a three layer (compute-memory-storage) disaggregation system GaussDB with efficient and elastic multiple writer capabilities, as shown in Figure 1. Three-layer disaggregation can make GaussDB more elastic by independently scaling compute, memory and storage resources. GaussDB logically partitions the pages into different compute nodes, and each compute node owns a subset of pages. The compute layer is in charge of SQL optimization and execution, transaction management, and recovery. For each transaction issued to a compute node, if all the relevant pages of this transaction are owned by this compute node, then the compute node can directly process the transaction; otherwise, the compute node obtains the ownership of all relevant pages and then processes the transaction. To capture data affinity and reduce page transmission costs, GaussDB designs an effective page placement and query routing method. The memory layer is in charge of page ownership management (maintaining a page ownership directory, i.e. the ownership of each page), global buffer management (i.e. warm pages that cannot be maintained in compute nodes), and global lock management (e.g. the holding and waiting on global locks). The memory layer is stateless and can be rebuilt from the compute node state. Most importantly, the memory layer allows near instant compute elasticity by separating compute growth and page ownership growth. The storage layer is responsible for page persistence, log persistence, and failure recovery. GaussDB utilizes two-tier failure recovery over both memory and storage checkpoints. If a compute node is down, GaussDB first uses a memory checkpoint to recover the node; if the memory layer fails, then GaussDB uses a storage checkpoint. Each compute node has a log stream and GaussDB only utilizes the logs of the failed compute node and does not need to access the logs of other nodes. If multiple nodes fail, GaussDB employs an efficient parallel recovery method to simultaneously recover different nodes.

In summary, GaussDB has several advantages. First, GaussDB achieves higher transaction throughput and lower latency with much fewer aborts compared to storage-layer log transaction conflict detection. Second, GaussDB achieves much faster recovery. Third, GaussDB has better scale-out and scale-in ability.

To summarize, we make the following contributions.

(1) We propose a cloud-native multi-primary database system, GaussDB, which uses a three layer (compute-memory-storage) disaggregation framework to support multiple writers.

(2) We devise a two-tier (memory checkpoint and storage checkpoint) recovery algorithm for fast recovery.

(3) We design a smart page placement method that judiciously assigns pages to different compute nodes and smartly routes queries to appropriate compute nodes in order to capture data affinity.

(4) We have deployed GaussDB internally at Huawei and with customers. The results show that GaussDB achieves higher performance and faster recovery, outperforming state-of-the-art baselines.

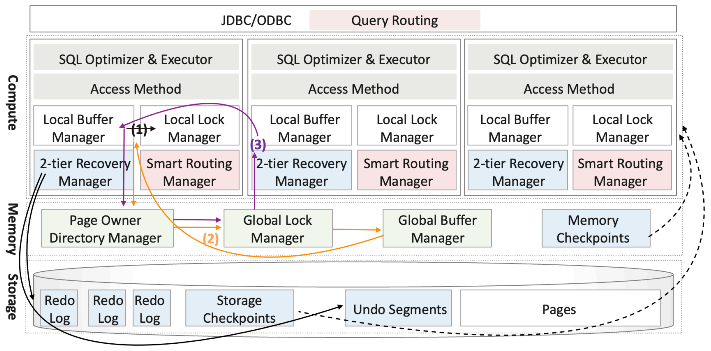


Figure 1: GaussDB Architecture

2 GAUSSDB ARCHITECTURE

GaussDB has three disaggregated layers: compute, memory and storage as shown in Figure 1. The compute layer logically and dynamically assigns page ownership to different compute nodes, and each compute node manages the pages assigned to the node; the memory layer provides global shared memory, and holds page ownership meta data; and the storage layer provides a globally shared storage. Compute nodes are in charge of SQL optimization, execution, and transaction processing. For each transaction on a compute node, the compute node gets the ownership of all the related pages and processes them on this node. Memory nodes provide unified shared memory which maintains global page ownership (i.e. which compute node owns which page), global buffers (i.e. data and index pages), global locks, and memory checkpoints. GaussDB can use memory checkpoints to accelerate failure recovery. Storage nodes are responsible for page and log persistence via a POSIX interface with the shared-storage file system. Storage nodes maintain storage checkpoints, which are used for failure recovery. The difference between a memory checkpoint and a storage checkpoint is that the former uses the pages in the shared memory and the memory checkpoint to recover while the latter uses the pages in storage nodes and the storage checkpoint to recover. Obviously, the former has faster recovery performance. If memory recovery fails, GaussDB uses storage checkpoints to continuously recover. Next we introduce the GaussDB modules as shown in Figure 2.

Compute Layer. Compute nodes are in charge of transaction processing. To support multiple writers, each compute node can modify any page once it acquires the page ownership. As with standard write-ahead logging it writes its changes to a redo log stream. To avoid page conflicts, each compute node manages a subset of pages, that is, each page has an owner node, and only the owner node has write privileges for this page. If a non-owner node wants to access a page, the node must get the write/read privilege from the owner node of this page. Thus, the compute node has a `local buffer manager` for maintaining the pages it owns in its local buffer pool and a `local lock manager` for access control to these pages. Given a transaction posed to a compute node, if the node owns all the relevant pages for this transaction (i.e. they all reside in its local buffer), GaussDB directly processes the transaction using its local buffers and local locks; if the node does not own all the pages, the compute node needs to find the pages (via the page ownership directory at the memory layer) and acquire ownership of these pages.

For recovery, the compute node has a `write-ahead log manager` and a `undo segment manager` for atomicity and durability. Note

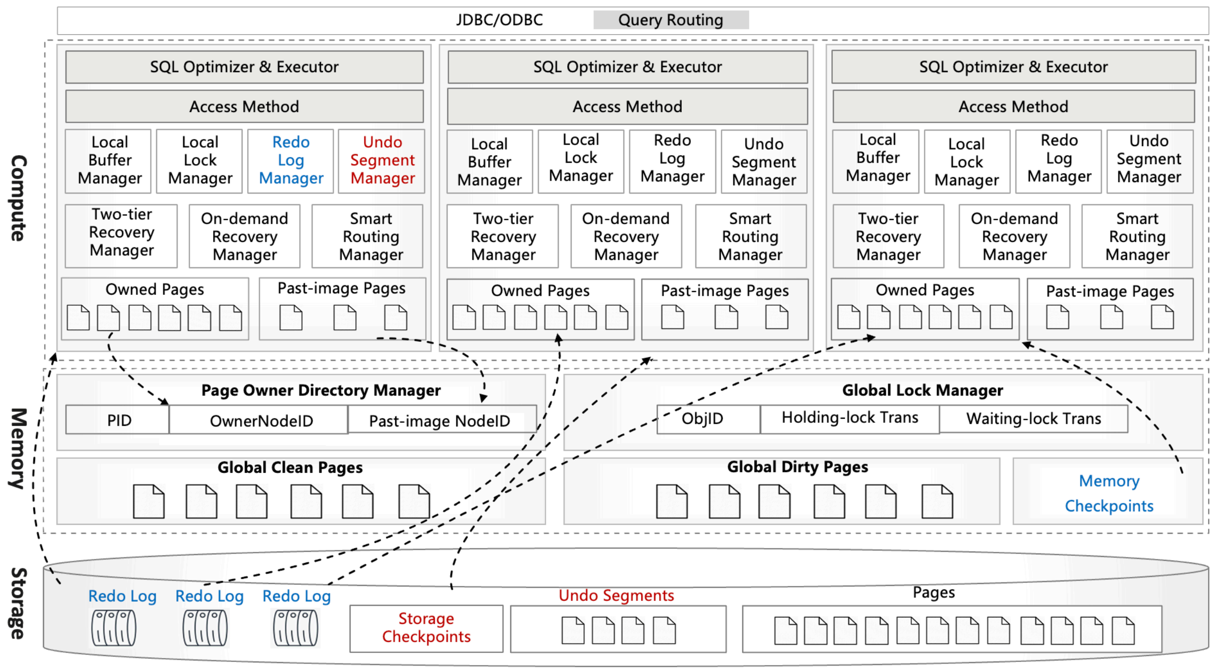


Figure 2: GaussDB Modules

that each compute node has an individual redo log stream while the undo segments are shared by all compute nodes. The redo logs are individually owned by each node for performance reasons. If the redo logs were shared, it would be prohibitively expensive to synchronize access across compute nodes, and node recovery would need to filter a significant amount of irrelevant logs. Hence individual redo logs were chosen. The undo segments are shared, but each transaction is assigned an individual undo segment for write during a transaction. This way an undo segment has only one writer, but any transaction can read and reconstruct any MVCC version by reading the appropriate shared undo segment pages. GaussDB uses an undo segment to maintain all the undo versions in the storage layer. The compute node has a two-tier recovery manager which includes a memory checkpoint manager for recovery from memory nodes and a storage checkpoint manager for recovery from storage nodes. We will discuss transaction processing in Section 3 and recovery in Section 4.

Pages have data affinity, i.e. some pages will be co-accessed by a transaction and some pages will not. It is inefficient to randomly assign an owner node for each page. Thus, the pages that are frequently co-accessed should have the same owner node. Smart query routing manager is used for routing the queries to appropriate nodes that own most of the pages of a query. The routing method is deployed at the JDBC/ODBC layer and updated at compute nodes. We will discuss the details in Section 5.

Memory Layer. The memory layer is stateless and can be fully reconstructed from compute node state, and hence does not require log based recovery. It has the following functionality. First, it maintains page ownership information. To find the ownership of a page, GaussDB uses a page owner directory (POD) to maintain the owner node of each page. It is expensive to maintain the POD of all pages because there could be a huge number of pages. To address this issue, POD only maintains the owner node for pages that are already loaded into memory. If a page is not in memory, then when a node accesses the page, GaussDB will load the page from storage and assign the page ownership to this node. The POD is maintained in the shared-memory layer. As there could be a large number of pages and the POD could be large, to avoid single-node bottlenecks and failures, GaussDB distributes the POD to all active memory nodes by using a consistent hashing strategy which partitions the POD over multiple nodes. Each local POD maintains a subset of page owners. There are several issues in maintaining the POD, including POD update and POD recovery. We will discuss POD update in Section 5 and POD recovery in Section 4.

Second, the memory layer is also used for global object/table lock management. Note that global page locking is maintained in the POD. To efficiently check whether a node can access a shared object GaussDB uses the lock manger to maintain lock information. If a node wants to lock an object it can acquire the lock at the global lock manager. As the global lock hash table could be large, GaussDB also uses a consistent hashing scheme to implement the global lock table. We will discuss lock recovery in Section 4. Note that when a transaction updates a tuple, it needs to first lock the page via the POD and then lock the tuple via the on page tuple lock. Then if the transaction updates the page, the compute node can release the page lock but still reserve the tuple lock until the transaction commits. In this way, other transactions can lock the page to update other tuples in this page that are not locked. Transactions wait on locked tuples using the lock manager to wait on the locking transaction id.

Third, the memory layer is also responsible for warm page caching. When the buffer pool of a compute node is full, GaussDB selects a page from the compute node for replacement to the memory layer. To accelerate performance, GaussDB can remove clean pages (the unmodified pages in memory which are the same as those in the storage nodes) into the memory layer. When a compute node accesses this page, GaussDB can directly return the page from the memory layer without loading from the storage layer. Note that GaussDB will not remove dirty pages to the memory layer because the memory layer is stateless and if the memory layer fails, the updates on the dirty pages will be lost. If the compute node has to select a dirty page to replace, GaussDB will first flush the page to the storage layer and then put the page in the memory layer for performance acceleration.

Fourth, the memory layer is responsible for memory elasticity. The shared buffer pool in the memory layer is shared by different compute nodes. In other words, the memory buffer pool is used on-demand by different compute nodes. If the memory buffer pool is full, GaussDB can on-the-fly expand the memory buffer pool.

Fifth, the memory layer can accelerate failure recovery. GaussDB flushes dirty pages into the memory layer and records a memory checkpoint. For a compute node failure, GaussDB uses memory checkpoints to recover. We will discuss the details in Section 4.

Storage Layer. The storage layer is used for page and log persistence. GaussDB maintains an individual redo log stream and a storage checkpoint for each compute node which are used for redo. GaussDB maintains a shared undo segment to maintain all the old versions of MVCC updates. Note that GaussDB partitions undo segments into fine-grained blocks. Each transaction applies for a block via the global lock manger and maintains the old versions in this block. When the transaction completes, it releases the lock and the block can be repeatedly used by other transactions. In our current design, we persist the pages to the storage layer because the network between the compute layer and storage layer is not a bottleneck. If the network is a bottleneck, we can write only the logs to the storage layer and replay the pages from the redo logs. However, this log-is-data idea requires additional compute resources in the storage layer to replay the pages.

Deployment. The storage layer is implemented by a block-based shared storage filesystem with a file interface. The compute layer is provided by the compute cloud. The memory layer can be provided individually or hybrid deployed with the compute layer. For example, if each node has 512GB memory, then we can deploy a compute node with 256GB local memory and the other 256GB can be organized as a shared memory block. Different layers can communicate with each other through TCP or RDMA. We use RDMA in our evaluation. We do not use local SSDs.

Lamport Clock. In distributed environments, the clocks of different nodes may not be well aligned. To address this issue, we use a *page-level transfer* based Lamport clock to synchronize the log sequence number (LSN) and a global lamport clock for the commit sequence number (CSN). GaussDB uses a background thread to synchronize the Lamport CSNs, and the compute nodes obtain CSNs from the background thread.

3 GAUSSDB TRANSACTION PROCESSING

In this section, we present how to support transaction processing as shown in Figure 3. Given a transaction routed to a compute node, GaussDB first gets the ownership of all relevant pages for this transaction, uses two-phase locking to lock the corresponding tuples, and then executes the transactions on this node[1]. Note that each page can be distributed into three types of nodes: (1) a compute node (the owner of the page); (2) a memory node (the owner of the page); or (3) the storage node (the page has no owner yet). In compute nodes and memory nodes, for each page in the node, GaussDB uses a flag to denote (1) the page is owned by the node; (2) the page is a read-only cached page for accelerating read; or (3) the page is a past image (e.g., the page owner is transferred to another compute node and the node records the page content before the transferring, which will be discussed later for accelerating recovery in Section 4). To find a page, GaussDB first finds the page in the local buffer pool of the compute node. If not in the local buffer pool, GaussDB uses the page ownership directory (POD) to find the page as the POD keeps page owner information.

[1]Note that there are different levels of locks, e.g., spinlock, latch, and lock. For ease of presentation, we do not distinguish them if there is no ambiguity.

POD Structure. The POD maintains a mapping from a page ID to its owner node ID as well as the lock information of this page (e.g., which node locks this page and which pages wait for the lock). As the POD may be rather large, we use a consistent hashing scheme to implement a distributed POD in the memory layer. For example, $(P0 : N0, T1, W)$ in Figure 3 denotes that the owner of page P0 is N0, and P0 is write-locked by transaction T1.

Global Lock Structure. The global lock structure maintains a mapping from an object (e.g., table lock) to its current locking node, locking transaction, and locking privileges (read or write), and also a waiting lock queue of waiting transactions on this object.

Next, we discuss how to process a page based on different cases – (1) read or write and (2) whether the node is the page owner.

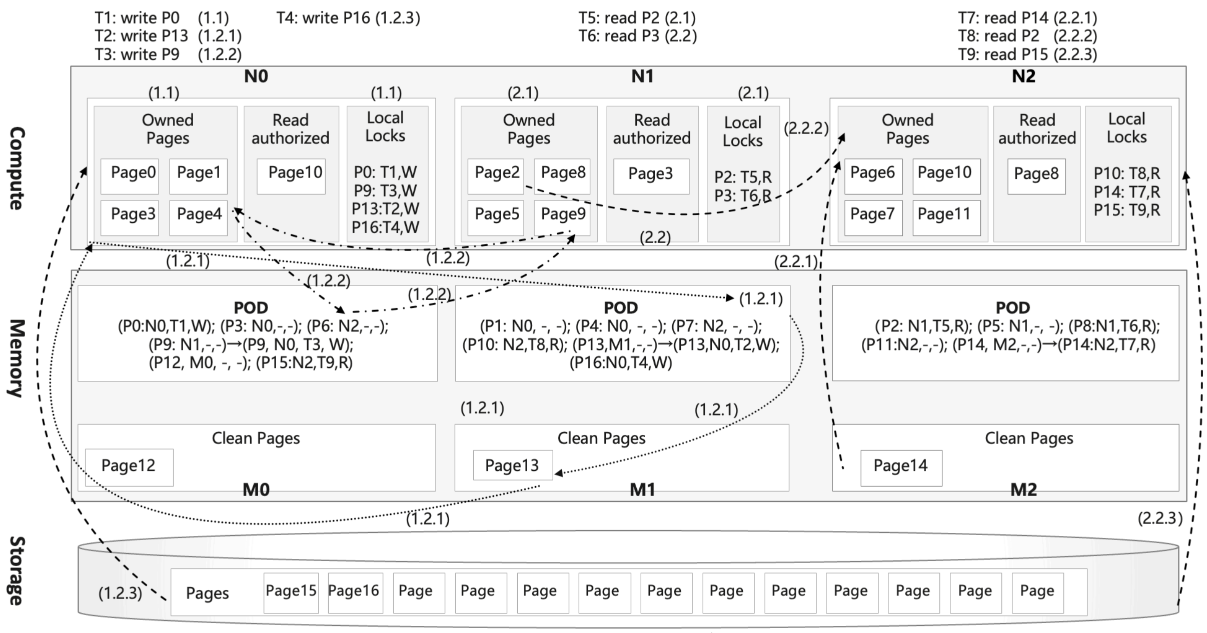
(1) Page Write. The transaction wants to write this page. The node checks the local buffer pool to verify whether the node is the page owner.

(1.1) The Node is Page Owner. If the page is in the local buffer pool and the node is the page owner (using a flag to denote whether this node is the page owner), the node gets the write lock from the local lock manager and writes the page on this node. For example, transaction T1 in node N0 can write page P0 with a local lock.

(1.2) The Node is Not Page Owner. The page is not in the local buffer pool, or the page is in the local buffer pool but the node is not the owner (based on the flag to denote the owner, in which case the page is a read-only cached page or a past image). The node uses the consistent hashing function to get the corresponding POD from the memory node. Based on the POD, it checks whether the page already has an entry in the POD. If yes, the node can obtain the page ownership from either the memory node, or another compute node; if no, then the page is in the storage node.

(1.2.1) Page Owner is a Memory Node. The page owner is a memory node. The compute node gets the ownership of the page from the memory node (via the POD) and moves the page from the memory node to its local buffer pool. It then sets itself as the page owner in the POD. If the node cannot get ownership, i.e. the page is locked by another node, then the compute node needs to wait for page access privileges in the POD. For example, transaction T2 in node N0 cannot directly write page P13 and has to get the lock.

(1.2.2) Page Owner is Some Other Compute Node. The page owner is some other compute node. The compute node gets the ownership of the page from the corresponding compute node (via the POD) and moves the page from the other compute node to its local buffer pool. Note that it also sets itself as the page owner in the POD. Similarly, if the node cannot get the lock, it needs to wait on the POD. For example, transaction T3 in node N0 cannot directly write page P9 and has to get the ownership of P9 from N1.



T1: write P0 (1.1)
T2: write P13 (1.2.1)
T3: write P9 (1.2.2)
T4: write P16 (1.2.3)
T5: read P2 (2.1)
T6: read P3 (2.2)
T7: read P14 (2.2.1)
T8: read P2 (2.2.2)
T9: read P15 (2.2.3)
Compute
Memory
Storage
N0
N1
N2
Owned Pages
Read authorized
Local Locks
Page0 Page1 Page3 Page4
Page10
P0: T1,W P9: T3,W P13:T2,W P16:T4,W
Page2 Page8 Page5 Page9
Page3
P2: T5,R P3: T6,R
Page6 Page10 Page7 Page11
Page8
P10: T8,R P14: T7,R P15: T9,R
POD
(P0:N0,T1,W); (P3: N0,-,-); (P6: N2,-,-); (P9: N1,-,-)→(P9, N0, T3, W); (P12, M0, -, -); (P15:N2,T9,R)
(P1: N0, -, -); (P4: N0, -, -); (P7: N2, -, -); (P10: N2,T8,R); (P13,M1,-,-)→(P13,N0,T2,W); (P16:N0,T4,W)
(P2: N1,T5,R); (P5: N1,-, -); (P8:N1,T6,R); (P11:N2,-,-); (P14, M2,-,-)→(P14:N2,T7,R)
Clean Pages
Page12
Page13
Page14
M0
M1
M2
Pages
Page15 Page16 Page Page Page Page Page Page Page Page Page Page Page

Figure 3: GaussDB Page Owner Directory and Transaction Processing

(1.2.3) There is No Page Owner. This page is in the storage layer and has no owner node yet. The compute node loads the page from the storage layer to its local buffer pool and then processes the page similar to case (1.1). It also sets itself as the page owner by adding a (page, this node) entry to the POD. For example, transaction T4 in node N0 cannot directly write page P16 and has to load the page from the storage layer.

(2) Page Read. The transaction wants to read a page. The node checks the local buffer pool to verify whether the node is the page owner.

(2.1) Node is Page Owner. If the page is in the local buffer pool and the node is the page owner (using a flag to denote this), the node gets the read lock and reads the page. For example, transaction T5 in N1 can read page P2 as N1 is the owner of P2.

(2.2) Node is Not Page Owner. The page is in the local buffer pool but the node is not the page owner. If the node is read authorized (i.e. the node is granted a read privilege, see below about read authorization), the node can also safely read the page. If the node is not read authorized, or the page is not in the local buffer pool, the node must read the page from the corresponding owner node. The node gets the POD from the memory node. Based on the POD, it checks whether the page is in the POD. If yes, the node can get the page ownership for this page, from either a memory node or another compute node; if no, the page is in the storage node. For example, transaction T6 in node N1 can directly read page P3 as it has a read authorization.

(2.2.1) Page Owner is a Memory Node. The page owner is a memory node. The node copies the page from the memory node to its local buffer pool. Note that in this case, the owner of this page will not be changed. The node is only read authorized, and the page owner needs to keep track of which node is granted with read authorization because when the page is updated, the owner needs to invalidate all read authorizations. If the node cannot get read authorization (i.e. locked by another node for a write operation), the node has to wait on the POD. For example, transaction T7 in node N2 can read page P14 from the memory layer.

(2.2.2) Page Owner is Some Other Compute Node. The page owner is some other compute node. The node copies the page from the corresponding compute node to its local buffer pool. Similar to case (2.2.1), the page owner will not be changed and the node is only read authorized. The node has to wait if it cannot get the read lock. For example, transaction T8 in N2 can read P2 from N1.

(2.2.3) There is No Page Owner. This page is in the storage layer and has no owner node yet. The node loads the page from the storage layer to its local buffer pool and gets the ownership of this page. GaussDB adds a (page, this node) entry to the POD to set page ownership. For example, T9 in N2 reads P15 from the storage layer.

Read Authorization. To improve the performance for read-heavy and write-light pages, we can grant read authorization for non-page owner nodes that frequently read a page. In this way, when a node reads a page in its local buffer pool, if the page is not invalidated, the node can safely read the page. When the owner node updates this page, it will invalidate all the read authorizations.

Two-Phase Locking. When processing a transaction, GaussDB gets the page access (read/write) privilege and processes the corresponding tuples. For tuple processing, GaussDB uses two-phase locking (2PL) to lock the tuples. For example, suppose a transaction updates two tuples, t_1 and t_2, on two pages, P_1 and P_2. GaussDB gets the lock of page P_1, gets the 2PL lock of tuple t_1, updates tuple t_1 on page P_1, releases the lock of page P_1, gets the lock of page P_2, gets the 2PL lock of tuple t_2, updates tuple t_2 on page P_2, releases the lock of page P_2, and implicitly releases the 2PL locks of t_1 and t_2 when committing.

On Page Tuple Locking. GaussDB performs tuple locking by marking a tuple with the transaction id that is currently modifying the tuple. As such if a transaction updates a tuple t on page P, it locks page P, and then updates tuple t with its transaction id and then releases the lock on page P. When other transactions want to also lock tuple *t* they can use the lock manager to check if the transaction id is still active, (and consequently wait on it), or determine that it's committed and then lock the row themselves. Each transaction id is composed of two parts ($undoSegment, SequenceNumber$). Each node at a node level owns undo segments (using the global lock manager), and then assigns an undo segment to a transaction locally during its first update; incrementing the associated sequence number to form a unique transaction id. The transaction then obtains a local transaction id lock (in the local lock manager only to avoid communications with the global lock manager). When another transaction wants to wait on a transaction id it first determines the correct node holding the lock by querying the associated global undo segment lock (in the global lock manager) and then sending the transaction id lock request to the appropriate node. On page tuple locks are cleaned up in an opportunistic fashion when accessing a page. This can be done cheaply using the minimum active transaction per undo segment which is synchronized periodically across nodes.

Two Layer Lock Management. GaussDB employs a two layer locking mechanism: the local lock at each compute node and the global lock at the memory layer. If a compute node accesses a page owned by itself, the node can get the read/write privilege via its local lock and without needing to check the global lock. If a node accesses a page owned by another node, it needs to get the privilege from both the global lock and its local lock. Specifically, it first checks the global lock, and if the page is not locked in the global lock, then it checks the local lock of the corresponding owner node; if the page is not locked by the local lock of the page owner, the node can get the privilege, otherwise it must wait for the lock to be released. Note that the global lock is also distributed and maintained via consistent hashing. When a page access is complete, the corresponding local locks must also be released.

Past-Image Page Caching for Fast Failure Recovery. When we transfer the page ownership from node N1 to N2, we also keep the page content in N1 and mark it as a past image. This is because if N2 fails, we can get the page information from N1 and do not need to recover from the logs in N1. We will discuss the details in Section 4. For example, P9 is transferred from N1 to N0, and the version at N1 is used to accelerate the recovery.

Buffer Replacement. (1) For a compute node, if its buffer pool is full, it needs to select a page to replace. If it selects a *clean page* (the page in the buffer pool is the same as that in the storage layer), it can safely move this page to a memory node (it also needs to update the page owner information in the POD); if it selects a dirty page, it has to write the page to both a memory node and a storage node. It also needs to update the corresponding page owner entry in the POD. This is because if we move a dirty page to a memory node but not to the storage layer, then if the memory node crashes, the dirty page will be lost. (2) For a memory node, if its buffer pool is full, it needs to select a page from the buffer pool to replace. If it selects a *clean page*, it can safely remove this page as the storage layer also has the same page. Note that it cannot replace a dirty page, because the compute node is in charge of flushing dirty pages and the memory layer is stateless. Thus, the clean pages in the memory layer are used for maintaining warm pages that are replaced from compute nodes; the dirty pages are used for accelerating compute node recovery based on memory checkpoints. Existing systems, e.g., IBM Db2 and SQL Server, also use the idea of buffer pool extensions that move pages from compute nodes to memory nodes in order to avoid moving them to storage nodes. We extend the idea to support fast failure recovery.

Page Ownership Optimization. For read-heavy pages, we can utilize read authorization to improve the performance. For pages whose owners are frequently updated (i.e. frequently accessed by different compute nodes), we can put their page ownership in memory nodes and utilize single-side RDMA to update such pages. For other pages, we put their ownership in the compute nodes.

Discussion on Log-is-Data. Log-is-data is widely used in cloud databases if the network between the compute and storage layer is the bottleneck, and we can address this issue by only writing logs and without writing dirty pages. In Huawei, the network is not a bottleneck, especially in the current high-speed network environment. So, we do not use log-is-data. GaussDB can be extended to support log-is-data: GaussDB will not flush dirty pages to the storage layer and the dirty pages will be replayed from the logs. We leave this as future work.

API and Compatibility. GaussDB has its own APIs and also supports some MySQL or PostgreSQL compliance.

4 GAUSSDB RECOVERY

We present GaussDB recovery as shown in Figure 4. We first discuss the stateless data structure recovery, and then discuss page recovery.

POD Recovery at the Memory Layer. The POD records the owners of each page and is stored in the memory layer. If a memory node fails and the POD is lost, GaussDB reconstructs the POD by scanning the compute nodes as each compute node keeps the page owned by the node. In other words, each compute node contains the (node, page) information. We use this information to generate the (page, owner node) information, like inverted indexes. If a compute node fails, we scan the POD to get pages owned by this node. If both the POD and compute nodes are down, we can reconstruct the POD when accessing (non-ownership) pages from the storage layer during log recovery.

Global Lock Recovery at the Memory Layer. The global lock manager keeps lock information for all objects, and similar to the POD, it is maintained in the memory layer. If the global lock manager fails, we can reconstruct it by scanning the local locks of compute nodes.

Page Recovery at the Memory Layer. When a memory node fails, we do not need to recover the pages in the memory layer as the memory node is stateless and all the pages in the memory layer are also kept in the storage node with the same content.

Lock Recovery at the Compute Layer. If the local lock is down: (1) if the lock is also maintained in the global lock, we can scan the global lock to get the locks of objects owned by the node; (2) if the lock is not in the global lock, then when the compute node restarts, the local lock is lost and we can re-apply the lock when required.

Page Recovery at the Compute Layer. When a compute node fails, we need to recover the pages in the compute node. We first select a new node, fetch the log, and replay the log as follows. To recover the pages in the compute node, GaussDB uses redo/undo logs to achieve fast failure recovery. Different from the Aries recovery algorithm [16], GaussDB records checkpoints on both the memory layer and the storage layer. If a compute node fails, GaussDB first

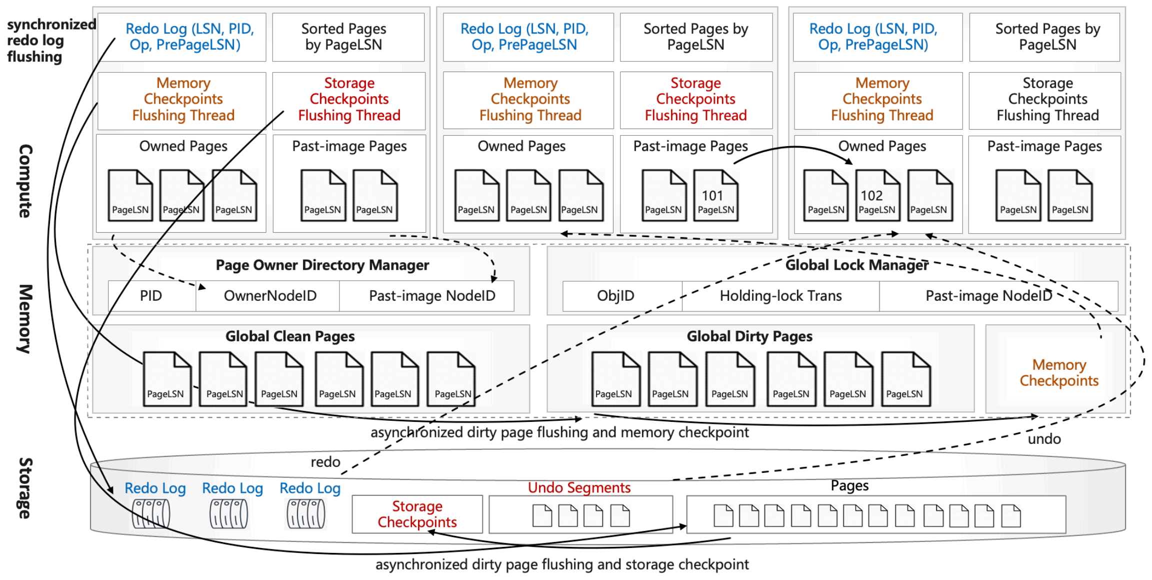

synchronized redo log flushing
Compute
Memory
Storage
Redo Log (LSN, PID, Op, PrePageLSN)
Sorted Pages by PageLSN
Memory Checkpoints Flushing Thread
Storage Checkpoints Flushing Thread
Owned Pages
Past-image Pages
PageLSN
101
102
Page Owner Directory Manager
PID
OwnerNodeID
Past-image NodeID
Global Lock Manager
ObjID
Holding-lock Trans
Global Clean Pages
Global Dirty Pages
Memory Checkpoints
asynchronized dirty page flushing and memory checkpoint
undo
redo
Redo Log
Storage Checkpoints
Undo Segments
Pages
asynchronized dirty page flushing and storage checkpoint

Figure 4: GaussDB Recovery

uses the memory checkpoint to recover. Only when a memory node fails does GaussDB use the storage checkpoint to continue the recovery. As the memory checkpoint is newer and the shared memory is faster than shared storage, recovering from the memory checkpoint is much faster. Even if memory recovery fails, recovering from the storage checkpoint guarantees correctness. This is because the redo log is idempotent, recovering redo logs can override the redo replay; the undo log has the compensation log and will not undo again.

Logging. Each compute node has an individual log stream to record the page insert/delete/update operations. (Note that for multiple tenants, each tenant has a log for tenant recovery and migration.) Each log record has a log sequence number (LSN). Ideally, GaussDB can recover a compute node page using its own log. However, the page may be transferred across different nodes, and page updates from different nodes can be distributed in different logs. To recover a page, we need to replay its related logs in a sequential order. To get a sequential order of related transactions, the LSNs of these related transactions should follow a partial order. However, the clocks of different nodes may not be aligned and some clocks may have drift, and thus to synchronize LSNs of different compute nodes, we use Lamport clocks to generate LSNs. When a transaction at node A transfers the page ownership of a page from node B and gets an LSN to generate a log entry, we set the LSN as max(A.LSN, B.LSN+1). In this way, we can guarantee a partial order of related transactions.

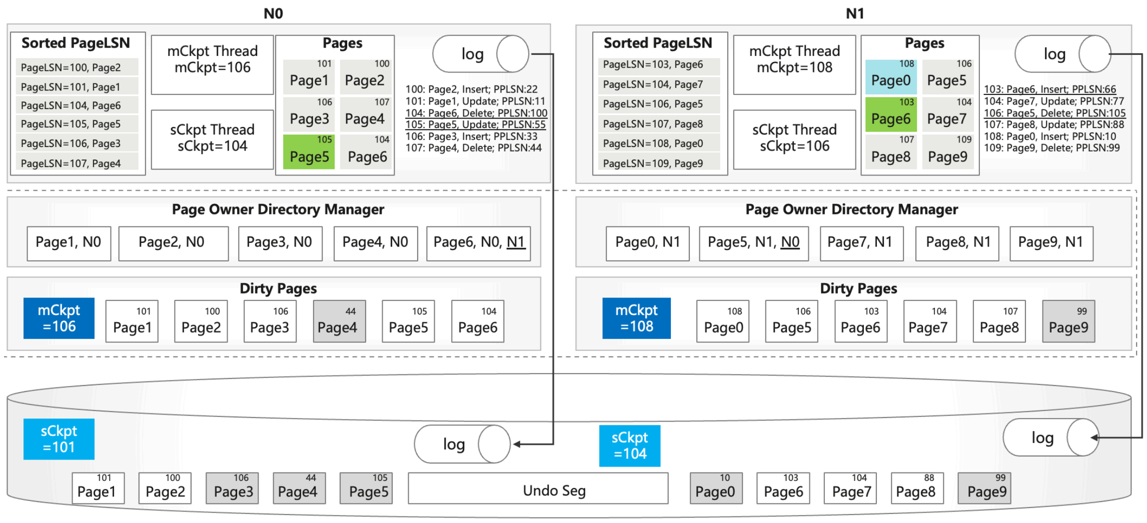
Undo Segment. GaussDB uses a separate undo segment to record the old versions for data update as well as all the active transactions. For each update, GaussDB records the redo logs at the write-ahead logging stream and the undo logs at the undo segment. In this way, for failure recovery, we can get all the active (incomplete) transactions by scanning the undo segment and then undo the operators of these active transactions by scanning the undo log entries from the end to the beginning. Besides, for undo operations, the undo segment also keeps multi-versions to support MVCC. Note that the undo segment can be reused, and old versions that will not be used by any transactions can be recycled by garbage collection.

Memory Checkpoint Recovery. GaussDB uses a separate thread to flush dirty pages to the memory layer. Note that it requires flushing the pages based on their LSNs in an ascending order, i.e. for a log entry with $LSN = 100$, if its corresponding updated page is flushed, all the page updates for $LSN < 100$ must also be flushed. After we keep the latest LSN in the memory checkpoint, for failure recovery, we can analyze the log from the flushed latest LSN and skip the log entries occurring before this LSN. Note that the flushed dirty pages (for failure recovery) are different from the replaced clean pages (for buffer replacement). The former have no owner node and are used for improving the performance and accelerating the failure recovery by caching them on shared memory. The latter have owner nodes and are used to accelerate transaction processing.

Storage Checkpoint Recovery. GaussDB uses a separate thread to flush dirty pages to the storage layer. Note that it also requires flushing the pages based on LSN in an ascending order. After we keep the latest LSN in the storage checkpoint, for failure recovery, we can analyze the log from the flushed latest LSN and then skip the log entries occurring before this LSN.

Two-Tier Failure Recovery. When a compute node fails, if the compute node can restart quickly, GaussDB recovers the node by recovering the POD, lock, and pages (using a memory checkpoint). If the compute node cannot be restarted quickly, GaussDB selects another compute node (with the smallest workload) to recover. Note that the selected node uses a separate log stream to recover the failed node. GaussDB also contains analysis-redo-undo phrases. The log analysis phase analyzes the logs from the memory (or storage) checkpoints and detects the redo log entries (completed transactions after the checkpoint) and undo entries (incomplete transactions). Next we discuss the redo and undo phases.

Past-Image Based Redo. For page recovery, if there is a single log, we can recover the log in LSN order (i.e. ascending order for redo and descending order for undo). We first consider the redo operation. A log entry $(LSN, PID, Operation)$ records the operation of updating a page with identifier PID, where LSN is a log sequence number, and the operation field records the before image and after



```
100: Page2, Insert; PPLSN:22
101: Page1, Update; PPLSN:11
104: Page6, Delete; PPLSN:100
105: Page5, Update; PPLSN:55
106: Page3, Insert; PPLSN:33
107: Page4, Delete; PPLSN:44
```

```
103: Page6, Insert; PPLSN:66
104: Page7, Update; PPLSN:77
106: Page5, Delete; PPLSN:105
107: Page8, Update; PPLSN:88
108: Page0, Insert; PPLSN:10
109: Page9, Delete; PPLSN:99
```

Figure 5: GaussDB Page Recovery Example

image of the operation. In traditional page recovery, if the page with PID is in-memory, then it is the latest version, and page recovery applies the operation on this page; otherwise, page recovery loads the page from storage. However, if a page is updated in different compute nodes and the update log entries are distributed in different logs, then there are two challenges. Firstly, the newest page may not be in the current node and possibly not even in the storage layer. Instead it may be on another compute node, and thus it is challenging to get the newest page for recovering a log entry. Secondly, a page may be updated on different compute nodes, and it is challenging to replay the logs that update the same page in order from different log streams.

First, given a log entry $(LSN, PID, operation)$, to replay the log, we need to first get the latest page identified by PID. The page may be in this node, another compute node, a memory node, or a storage node. We need to get the newest page, i.e. the page with the largest PageLSN (when updating a page via a log record with LSN, we update the PageLSN with the log entry LSN), because the newest page can minimize the number of newest redo operations and avoid replaying them from multiple log streams. To get the newest page, a straightforward method first scans all the buffer pools of each compute node. If it cannot find the newest page in the buffer pools, it gets the page from the storage node. Moreover, given the newest page and log entry, if there are some updates for this page on other compute nodes before this log entry, we cannot replay this log entry on the newest page, because we need to first replay all other logs before this log entry. However it is expensive to find such logs. To this end, for each compute node, we do not remove the previous versions of a page; instead, we maintain them in the buffer pools of compute nodes, i.e. when we transfer a page owner from node N1 to node N2, we keep the page at N1 (called *the past image*, PI). In this way, if a single compute node is down, we can prove that the newest page contains all the updates in the logs of other compute nodes, i.e. we do not need to replay other logs. To efficiently identify the newest page, in the POD, we also keep a list of past-image pages, which include the page LSN and compute node IDs for this page LSN. In this way, when we find a page, we efficiently identify its past image based on the POD. Note that when a page is flushed to the storage layer, we remove all past images of this page (including any past images at compute nodes and the past-image list at POD).

Second, if multiple compute nodes fail, we recover the page following the LSN order that may be in different log streams. Specifically, when we replay a log entry that updates a page, we need to know whether there is another log entry that updates this page before the LSN. If yes, we wait until all log entries before this LSN that have updated the pages have been replayed. Here, a challenge is how to know whether there is a log entry that updates the same page before the LSN. To this end, for each page, we record the latest LSN that updates the page; and for each log entry, we also record the previous page LSN that records the page status when the log entry reads the page content. In this way, when we replay a record, if the previous page LSN is equal to the page LSN, we can replay the log entry. If the previous page LSN is larger than the page LSN, i.e. another compute node updated the page before this log entry, then we need to wait until the other compute node replays its log entries; if the previous page LSN is smaller to the page LSN, we skip the log entry since the page is newer than the log entry.

In summary, if two or more nodes fail, GaussDB can recover them in parallel. The POD and locks have no conflicts between different compute nodes, and thus they can be recovered fully in parallel. Specifically, for each failed compute node, GaussDB scans the log entries from the checkpoint in an ascending order. For each log entry $(LSN, PID, operation, \texttt{PrePageLSN})$, which records that the operation of this log entry updates the page whose PageLSN is PrePageLSN, GaussDB compares log LSN and PageLSN. If PageLSN $>$ log LSN, GaussDB skips this log; otherwise, GaussDB compares PageLSN and PrePageLSN, if PageLSN = PrePageLSN, GaussDB redoes this operation on this page; otherwise, there is another update before this log in other compute nodes, GaussDB needs to wait on this page until PageLSN = PrePageLSN.

Undo-Segment Based Undo. For undo operations, for each aborted transaction, we undo its operations from its last undo log entry (in undo segments). For each undo log entry, we undo the operations. Note that we undo the update operators in a reversed order of the undo log entries. We first get the active transactions from undo segments, get the last log entry of each active transaction and undo the log entries of these active transactions from the last log entry following the previous LSN order until reaching the first LSN (e.g., the transaction start). Note that the log entries of a transaction must be in the same log stream and cannot be across different log entries. Thus we can undo the logs in one compute node and do not consider other nodes. When we do the redo operations, we have already obtained the newest page in this node, thus we can easily undo the operations.

Example. Figure 5 shows a running example of page recovery. Page P5 is first updated at compute node N0 with LSN of 105 and then updated at N1 with LSN of 106. Note that P5 with PageLSN of 105 is also cached at N0 (called past image). This is because if N1 is down, we can recover P5 from PageLSN of 105 and log entry 106 at N1 without accessing the log on N0. If P5 with PageLSN of 105 is not cached at N0, it also needs to replay the log entry 105 at N0 which is rather expensive. In addition, we can see that with a memory checkpoint, we only need to recover P4 at N0 with log entry of 107 and P9 at N1 with log entry of 109. If we recover from a storage checkpoint, we need to replay many more log entries.

On-Demand Recovery. If a compute node fails, another node takes over and replays the log. At this time, if there is a transaction that accesses a page that has not been replayed yet, the full-log replay method that replays all the logs needs to wait for a long time. To provide instant query processing for this transaction, we propose an on-demand log recovery method, which only replays the required pages and can skip log entries for updating other pages. To this end, GaussDB first identifies the log entries that updates this page, and then only redoes these relevant log entries.

5 GAUSSDB SMART ROUTING

Given a SQL query, GaussDB uses JDBC/ODBC to route the query to an appropriate compute node which then processes the SQL query. A random routing method may not capture the data affinity, e.g. sending a query to a compute node which does not own the pages involved in the query, thus involving page ownership transfer and page transmission, leading to poor performance. To address this issue, we propose a smart routing method.

Graph Partitioning for Page Affinity. We model the pages as a graph where each vertex is a page. There is an edge between two pages if the two pages are co-accessed by a query, and the weight of this edge is the co-access frequency. The graph captures the page affinity. We can use existing graph partitioning algorithms to partition the graph into subgraphs (e.g. strong connected components), and then the pages in each subgraph can be assigned to the same owner node. As it is expensive to build the page graph, we simplify this problem by grouping multiple pages (e.g. 1K) into a single node based on the (primary and foreign) keys in the pages. We periodically update the graph and graph partitions.

Different from [7] that optimizes scalability of shared-nothing databases by partitioning the data, GaussDB aims to dynamically group the pages based on their affinity. We propose a light-weight method that can judiciously route the query to appropriate nodes.

Query Routing. Given a query, we need to route this query to an appropriate compute node. A straightforward method is to first calculate the pages involved in this query, and then route the query to the node that owns most of the accessed pages for this query. However it is expensive to obtain the accessed pages of a query. To this end, we propose a multi-layer perceptron (MLP) to predict the accessed pages. The input is the query encoding to encode the query features and database schema (including column name, operators, predicate values, and distinct values). The output is an access vector, where each element denotes the access possibility of a page group. Since there could be a large number of pages, each element in the vector does not denote a single page. Instead, each element denotes a page group with multiple pages (e.g. 1K pages), which is the same as the node grouping. Similarly, we can obtain an owned vector of a compute node to capture the pages in each node, where the value of each element in the vector is the percentage of pages owned by this node out of the number of pages in the element. Based on the predicted access vector of each query and the owned vector of each node, we compute the similarity (e.g. cosine) and route the query to the node with the largest similarity.

Lightweight Query Routing. One question is where to deploy the routing module in the database system. The first option is to deploy it at a compute node. However, this method requires re-routing the query if the compute node is not the best choice for the query, which is rather expensive. The second option is to deploy it into the JDBC/ODBC layer, which is very efficient. However, JDBC/ODBC has limited computing resources, i.e. we cannot put model training at the JDBC/ODBC layer. To this end, we deploy the lightweight model inference procedure (i.e. query routing) at the JDBC/ODBC layer and deploy the model training and update procedures at the server side. The query routing model is periodically updated at the compute nodes and synchronized to the JDBC/ODBC layer.

Smart Routing Workflow. Given a query, we use the lightweight query routing model at the JDBC/ODBC layer to route the query to a compute node. Then the compute node processes the query. Note that the compute node will asynchronously send the query and its access page vector to the model fine-tuning module, which will fine-tune and update the routing model based on the query. The server will periodically use the server side model to update the inference model at the JDBC/ODBC layer.

6 GAUSSDB SCALING

Compute Node Expansion/Shrink. If each compute node is busy, we can add new compute nodes to grow compute capacity and reduce the burden on the compute nodes. GaussDB only needs to move some pages to new nodes and then assign page ownership to the new nodes. We also need to update the page routing method that routes queries to new nodes. Note that even if the smart routing method initially directs a query to a sub-optimal compute node, this node has the capability to reroute the query to the optimal node or process the query locally. Consequently, the smart routing method ensures the optimal performance. To this end, we can periodically update the smart routing model. Similarly, we can shrink the compute nodes.

Memory Node Expansion/Shrink. If each memory node is busy, we can add new memory nodes. This requires expanding the POD and global lock manager. For the POD, we use consistent hashing to do the expansion. We first use virtual nodes to get more hash buckets (each bucket corresponds to a virtual node), and then map the virtual nodes to physical nodes. Thus, for POD expansion, we only adjust the POD buckets (virtual nodes). In case we want to move a virtual POD bucket to another node, we just lock the bucket (even if there are active transactions) and move all related POD

entries to the new node, and then release the lock. Since each POD bucket is very small, the POD can be updated online. Note that we do not move the pages and instead the pages are moved to the new node only if there is a relevant query in this new node that writes the page. We expand the global lock manager in a similar manner. We can also shrink the memory nodes using a reverse procedure.
Storage Layer Expansion/Shrink. As we design a distribute block file system, it is easy to expand and shrink the storage layer.

7 EXPERIMENTS

7.1 Experimental Setting

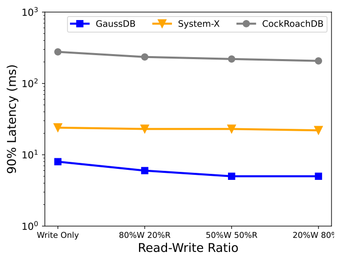
Experimental Environment. We performed all experiments on Huawei cloud, where each underlying physical server was equipped with Intel 6248R*2 CPU with 96 physical cores, 512GB RAM, 192TB SSD disk, Mellanox ConnectX-6 100GE network card, and running EulerOS operating system. We built a compute-memory-storage disaggregation system on the cloud.
Datasets. We conducted the experiments on two standard benchmarks, TPC-C and Sysbench. For TPC-C, we used 10,000 warehouses. For Sysbench, we tested write only, 80% write/20% read, 50% write/50% read, and 20% write/80% read. TPC-C is a well partitionable dataset while Sysbench is a less partitionable dataset. We also evaluated our system on customer workloads in Section 7.7.
Baselines. We compared with two state-of-the-art multi-primary databases, a shared-everything multi-primary database System-X (latest version, anonymized due to legal compliance) and a shared-nothing multi-primary database CockRoachDB (version 23.2). Note that multi-primary Aurora, multi-primary PolarDB and Purescale were not available at the time of this writing, and thus we could not compare with them. As CockRoachDB used a shared-nothing architecture, it can be physically deployed on multiple nodes by sharding the data. As GaussDB adopts a compute-memory-storage disaggregation architecture, we deployed it logically on multiple nodes, i.e. the resource of each node was locally partitioned to 3 parts and each part was respectively used to deploy compute, memory, and storage layers. For System-X, we deployed it on these servers and with an all-SSD shared storage file system OceanStor Dorado[2]. We tuned the performance of System-X (e.g., the global and local buffer size) and CockRoachDB (e.g., partition methods) and reported their best performance.

7.2 End-to-End Performance

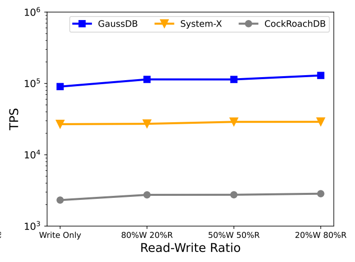
We compared the end-to-end performance of GaussDB with System-X and CockRoachDB on TPC-C with 10,000 warehouses. We used four metrics: 90% latency, QPS (query per second), TPS (transaction per second), and tpmC (new order transactions per minute). Table 1 shows the results on three nodes. We had the following observations. First, GaussDB achieved lower latency than System-X and CockRoachDB, even 4-30 times better. For example, the latency of GaussDB was only 11 milliseconds, and those of System-X and CockRoachDB were respectively 40 milliseconds and 300 milliseconds. System-X and GaussDB were better than CockRoachDB because CockRoachDB required two-phase commit for distributed transactions which was costly to synchronize across multiple nodes while System-X and GaussDB did not require two-phase commit

Table 1: Performance on TPC-C.

Systems	90%Latency(ms)	QPS	TPS	tpmC
System-X	39.68	500603	56693	1530723
CockRoachDB	302.02	37920	4294	115952
GaussDB	11.01	958549	108556	2931011



(a) 90% Latency (ms)



(b) TPS

Figure 6: Performance on Sysbench.

and thus had lower latency. GaussDB was better than System-X because System-X involved many page transmissions across different nodes while GaussDB significantly reduced page transmission based on smart routing and shared memory. Second, in terms of QPS and TPS, GaussDB was better than System-X, which was in turn better than CockRoachDB. This was attributed to the transaction processing of GaussDBwhich did not use two-phase commit and reduced page transmission across multiple nodes.

We also evaluated the end-to-end performance on the Sysbench dataset. Figure 6 shows the results on three nodes. We had the following observations. First, GaussDB was better than System-X, which in turn was better than CockRoachDB in terms of both throughput and latency. For example, the throughput of GaussDB was 4 times that of of System-X and 20 times that of CockRoachDB. The latency of GaussDB was 1/4 of that of System-X and 1/30 of that of CockRoachDB. This was because GaussDB had an effective transaction processing method by reducing the overhead of storage access and page transmission. Second, for both write-only and read-write workloads, GaussDB was better than System-X and CockRoachDB. This was attributed to our effective transaction processing method. Third, the three databases achieved higher performance on read-write workloads than on write-only workloads because read-write workloads had less transaction conflicts.

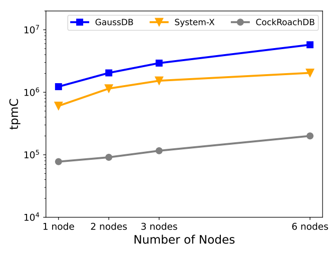
7.3 Evaluation on Scale-out

We evaluated the scale-out of GaussDB, CockRoachDB and System-X by varying the number of nodes and then measuring 90% latency, QPS, TPS, and tpmC on TPC-C. Figure 7 shows the experimental results. We made the following observations.

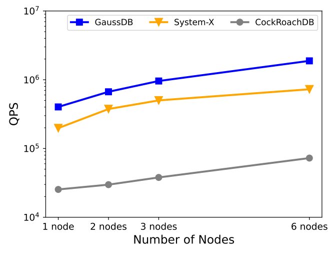
First, we could see that with the increase of the number of nodes, the throughput (including tpmC, QPS, and TPS) of all the systems increased. This was because all of these systems were able to take advantage of more resources to process queries.

Second, GaussDB outperformed System-X which outperformed CockRoachDB for throughput. GaussDB had 2.5 times higher TPS than System-X and 20 times higher than CockRoachDB. The reasons were three-fold. Firstly, CockRoachDB used two-phase commit for distributed transaction processing, which was rather costly, while GaussDB and System-X did not involve two-phase commit. Secondly, GaussDB achieved much better page affinity based on smart

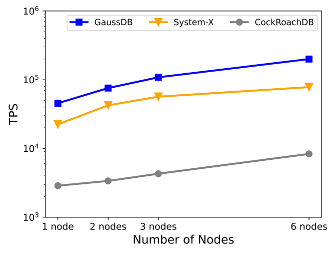
[2]https://e.huawei.com/en/products/storage/all-flash-storage



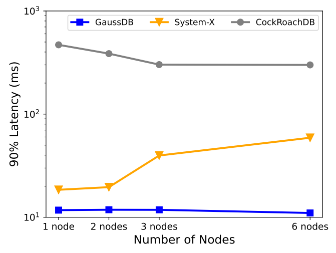
(a) tpmC



(b) QPS

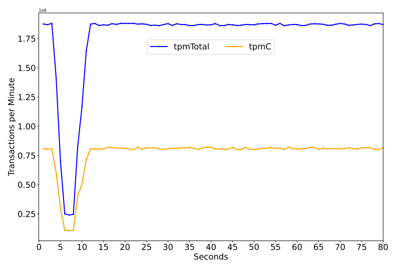


(c) TPS

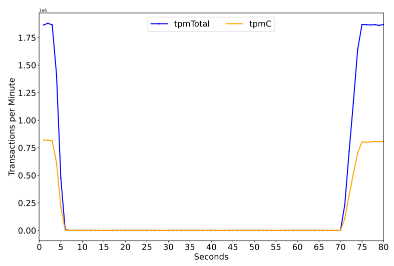


(d) 90% Latency (ms)

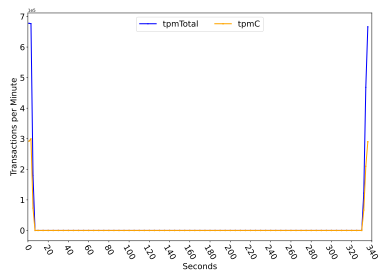
Figure 7: Evaluation of Scale-out on TPC-C



(a) `GaussDB` Recovery Curves



(b) `System-X` Recovery Curves



(c) `CockRoachDB` Recovery Curves

Figure 8: Recovery for Compute Node Failure on TPC-C

routing and thus reduced the page transmission cost compared to System-X. Thirdly, GaussDB used shared memory to reduce latency and addressed the data skew problem because if a page was not in the compute nodes, GaussDB got the page from shared memory, but System-X might have to fetch the page from the storage layer, which was rather slow. Especially for data skew, GaussDB used the smart routing method that judiciously assigned pages into different nodes and used a shared buffer pool to improve the performance.

Third, GaussDB had much lower latency than System-X, which in turn had much lower latency than CockRoachDB. For example, the latency of GaussDB was only 11 milliseconds, that of System-X was 50 milliseconds, and that of CockRoachDB was 300 milliseconds. GaussDB had lower latency because (1) GaussDB used smart routing to organize the data so that it had high probability to use the local cache to answer queries and did not need to fetch data from the storage layer; (2) even if data was not in the local cache, GaussDB used the shared memory layer to achieve low latency.

Fourth, with the increase of the number of nodes, GaussDB still achieved much higher performance than System-X and CockRoachDB. This was attributed to our disaggregation architecture and smart routing method. The former allowed for high scalability and elasticity, and the latter reorganized the data based on page affinity. With the increase of the number of nodes, the latency of CockRoachDB was reduced because CockRoachDB had more resources to process the queries in parallel by partitioning the data; the latency of System-X was increased because System-X needed to get all relevant data to a compute node to process queries; the latency of GaussDB was stable because GaussDB used a local buffer pool and shared memory to reduce the latency.

7.4 Evaluation on Failure Recovery

7.4.1 Compute Node Failure. We evaluated our recovery method for compute node failure, i.e. we killed a compute node after three seconds of running and evaluated recovery performance. We compared GaussDB with System-X and CockRoachDB on the TPC-C dataset. Figure 8 shows the results. We made the following observations. First, as shown in Figure 8b, System-X had much longer recovery time than GaussDB because it needed to recover the failed node from the storage layer. As shown in Figure 8a, GaussDB had much better recovery speed. The RTO (recovery time object) of GaussDB was about 8 seconds, while the RTO of System-X was 70 seconds. This was because GaussDB could use the memory checkpoints to recover failed nodes, which was fairly efficient. As shown in Figure 8c, CockRoachDB had much longer recovery time, and the RTO was longer than 300 seconds, because CockRoachDB had to synchronize different nodes. Second, GaussDB had much better long-term stability, i.e. the performance fluctuation was very low. This was attributed to the smart routing method that reduced the query routing overhead and page transmission overhead. Third, the performance of System-X and CockRoachDB degraded to zero during recovery while GaussDB did not. This was because GaussDB designed an on-demand recovery technique which only recovered targeted pages and did not need to replay all the redo logs.

7.4.2 Memory Node Failure. Next, we evaluated the case of memory node recovery by killing a memory node. As only GaussDB had a memory layer, we only evaluated GaussDB. Figure 9 shows the results. GaussDB achieved very high recovery speed for memory node recovery. This was because the memory layer was stateless, and the memory node recovery did not affect transaction processing. Moreover, the memory node recovery speed was better than compute node recovery because the memory nodes were stateless.

7.4.3 Compute and Memory Node Failure. We evaluated the case of both compute and memory node recovery by killing a compute node and a memory node simultaneously. Figure 10 shows the results. We could see that GaussDB achieved high recovery speed even for concurrent compute and memory recovery. This was attributed to our shared memory and transaction processing technique.

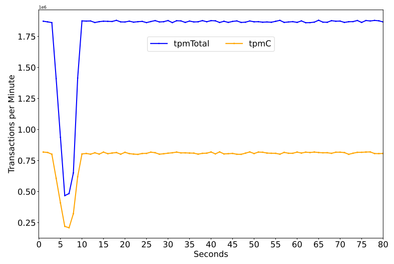


Figure 9: GaussDB Recovery for Memory Failure on TPC-C

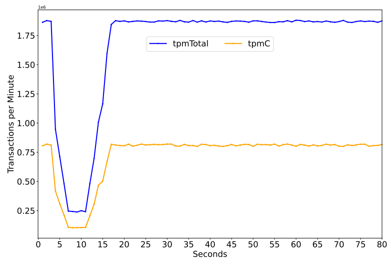


Figure 10: GaussDB Recovery for Compute and Memory Failure on TPC-C

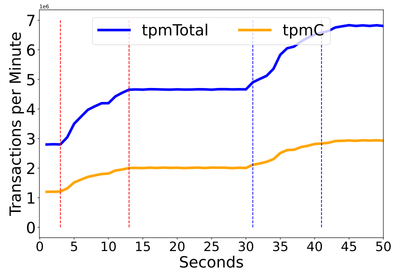


Figure 11: GaussDB Node Expansion Elasticity on TPC-C

Table 2: Evaluation with/without Smart Routing on TPC-C

	without	with
90% Latency (ms)	17.36	11.01
QPS	453998	958549
TPS	51415	108556
tpmC	1388217	2931011

Table 3: Evaluation on GaussDB with/without Page Ownership Management and Read Authorization on Sysbench

TPS	without	with
Write only	46734	90343
80% W 20% R	52981	113934
50% W 50% R	53326	113762
20% W 80% R	57467	129439

7.5 Evaluation on Elasticity

We evaluated the elasticity of GaussDB by expanding the cluster from 1 node to 2 nodes, and then to 3 nodes. Figure 11 shows the results. GaussDB achieved high elasticity performance as we expanded the cluster with more nodes. The expansion time was about 10 seconds. This was because GaussDB had a memory layer to reduce the repartition overhead and thus achieved high elasticity.

7.6 Ablation Study

7.6.1 Evaluation on Smart Routing. We evaluated our smart routing method. We compared GaussDB with smart routing and without smart routing. Table 2 shows the results. We could see that GaussDB with smart routing achieved much higher performance than GaussDB without smart routing. For example, the throughput was doubled by using smart routing. This was because without smart routing, GaussDB might route a query to a suboptimal compute node and thus this node had to pull pages from remote nodes to satisfy the query; with smart routing, GaussDB routed the queries to most relevant compute nodes and reduced the query routing across different compute nodes. In addition, with smart routing, the relevant pages (co-accessed by transactions) were grouped together, and thus it significantly reduced the number of page transmissions across nodes. Note that System-X also had routing strategies. Even without smart routing, GaussDB was still better than System-X on latency. We also added the overhead of the model learning cost and the model maintaining cost. We only used one core to fine-tune the model and updated the model, and thus the overhead was very small, only about 1% (1 core vs 96 cores).

7.6.2 Evaluation on Page Ownership Management and Read Authorization. We compared our method with and without page ownership management and read authorization. Table 3 shows the results. We could see that GaussDB with page ownership management and read authorization improved the performance by 2.2 times. This was because page ownership management and read authorization reduced page transmission across different compute nodes, and moreover read authorization avoided remote page access.

7.6.3 Evaluation on Past Images. We evaluated the effectiveness of keeping past images of pages. First, past images would be eliminated if they were flushed to the storage layer, and thus they only took up limited space (only about 1% in our experiments). Second, the average retention time of keeping past images was 10 minutes. Third, past images can improve the recovery time by 1.8 times.

7.7 Customer Use Cases

We also evaluated the performance of GaussDB on customer scenarios. First, consider an airline company with 162 TB data and 116 million users. There were multiple business scenarios on this database, including ticket booking, arrival and departure, seat reservation, passenger check-in, etc, and thus this was a less partitional dataset. The TPS of GaussDB and System-X were respectively 12,870 and 5,993. So GaussDB outperformed System-X by 115%. Second, consider a power grid company with 45 TB data and 252 million users. This was also a less partitional dataset. The TPS of GaussDB and System-X were respectively 3,961 and 1,878, and GaussDB outperformed System-X by 111%. The two customer scenarios showed that GaussDB achieved higher performance on customer workloads.

8 CONCLUSION

We present a cloud-native multi-primary database GaussDB, which is a compute-memory-storage disaggregation system that provides multi-writer capabilities. GaussDB logically partitions the pages to different compute nodes and assigns the ownership of each page to a compute node. GaussDB designs an effective transaction processing method across multiple nodes. GaussDB adopts an effective page placement and query routing method to capture data affinity. GaussDB devises a two-tier failure recovery method to improve recovery performance. We have implemented and deployed GaussDB internally at Huawei and with customers, and the results showed that GaussDB achieved much higher throughput, lower latency, and faster failure recovery compared to several state-of-the-art multi-primary databases.

ACKNOWLEDGMENTS

This paper was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (61925205, 62232009, 62102215), Zhongguancun Lab, Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist). Guoliang Li is the corresponding author.

REFERENCES

[1] Daniel Abadi, Anastasia Ailamaki, David G. Andersen, Peter Bailis, Magdalena Balazinska, Philip A. Bernstein, Peter A. Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Y. Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh M. Patel, Andrew Pavlo, Raluca A. Popa, Raghu Ramakrishnan, Christopher Ré, Michael Stonebraker, and Dan Suciu. 2022. The Seattle report on database research. *Commun. ACM* 65, 8 (2022), 72–79.

[2] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, et al. 2019. Socrates: The New SQL Server in the Cloud. In *SIGMOD*. 1743–1756.

[3] Wei Cao, Feifei Li, and et al. 2022. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *ICDE*. IEEE, 2859–2872. https://doi.org/10.1109/ICDE53745.2022.00259

[4] Wei Cao, Yang Liu, Zhushi Cheng, et al. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *FAST*. USENIX Association, 29–41.

[5] Wei Cao, Yingqiang Zhang, Xinjun Yang, et al. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *SIGMOD*. 2477–2489.

[6] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 251–264. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[7] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. 2010. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1 (2010), 48–57. https://doi.org/10.14778/1920841.1920853

[8] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, et al. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. 215–226.

[9] Alex Depoutovitch, Chong Chen, Jin Chen, et al. 2020. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *SIGMOD*. 1463–1478.

[10] Alex Depoutovitch, Chong Chen, Per-Åke Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, and Calvin Sun. 2023. Taurus MM: bringing multi-master to the cloud. *Proc. VLDB Endow.* 16, 12 (2023), 3488–3500. https://doi.org/10.14778/3611540.3611542

[11] Haowen Dong, Chao Zhang, Guoliang Li, and Ji Sun. 2024. Cloud-Native Databases: A Survey. *IEEE Transaction Knowledge Data Engineering* 34, 3 (2024), 1096–1116.

[12] Andi Gutmans. 2022. Introducing AlloyDB for PostgreSQL: Free yourself from expensive, legacy databases. https://cloud.google.com/blog/products/databases/introducing-alloydb-for-postgresql.

[13] 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.

[14] Guoliang Li, Haowen Dong, and Chao Zhang. 2022. Cloud Databases: New Techniques, Challenges, and Opportunities. *VLDB* 15, 12 (2022), 3758–3761.

[15] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. openGauss: An Autonomous Database System. *Proc. VLDB Endow.* 14, 12 (2021), 3028–3041. https://doi.org/10.14778/3476311.3476380

[16] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162. https://doi.org/10.1145/128765.128770

[17] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD*. 115–130.

[18] Vivek R. Narasayya and Surajit Chaudhuri. 2021. Cloud Data Services: Workloads, Architectures and Multi-Tenancy. *Foundations and Trends in Databases* 10, 1 (2021), 1–107.

[19] Vivek R. Narasayya and Surajit Chaudhuri. 2022. Multi-Tenant Cloud Data Services: State-of-the-Art, Challenges and Opportunities. In *SIGMOD*. 2465–2473.

[20] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD*. 131–141.

[21] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. 2021. Magic Quadrant for Cloud Database Management Systems. *Gartner (2021, December 13)* (2021), 1–37.

[22] Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan. 2022. Moneyball: Proactive Auto-Scaling in Microsoft Azure SQL Database Serverless. *VLDB* 15, 6 (2022), 1279–1287.

[23] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, et al. 2022. Cloud-Native Transactions and Analytics in SingleStore. In *SIGMOD*. 2340–2352.

[24] IBM DB2 PureScale. [n.d.]. DB2 pureScale: Best Practices for Performance and Monitoring. Technical Report. IDUG DB2 Technical Conference, Denver, CO, USA.

[25] Oracle RAC. [n.d.]. Oracle Real Application Clusters 19c Technical Architecture. https://www.oracle.com/webfolder/technetwork/tutorials/architecture-diagrams/19/rac/pdf/rac-19c-architecture.pdf.

[26] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, João Carreira, Neeraja Jayant Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What serverless computing is and should become: the next phase of cloud computing. *Commun. ACM* 64, 5 (2021), 76–84.

[27] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, et al. 2020. Cloudburst: Stateful Functions-as-a-Service. *VLDB* 13, 11 (2020), 2438–2452.

[28] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *SIGMOD*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1493–1509. https://doi.org/10.1145/3318464.3386134

[29] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, et al. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. 1041–1052.

[30] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, et al. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *SIGMOD*. 789–796.

[31] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *SIGMOD*, Pablo Barceló, Nayat Sánchez Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 295–308. https://doi.org/10.1145/3626246.3653377

[32] Yifei Yang, Matt Youill, Matthew E. Woicik, et al. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *VLDB* 14, 11 (2021), 2101–2113.

[33] Chao Zhang, Guoliang Li, and Tao Lv. 2024. HyBench: A New Benchmark for HTAP Databases. *Proc. VLDB Endow.* 17, 5 (2024), 939–951. https://www.vldb.org/pvldb/vol17/p939-zhang.pdf

[34] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *SIGMOD*. 415–432. https://doi.org/10.1145/3299869.3300085

[35] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, et al. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *VLDB* 14, 10 (2021), 1900–1912.

[36] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, et al. 2021. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1900–1912.

[37] Weixing Zhou, Qi Peng, Zijie Zhang, Yanfeng Zhang, Yang Ren, Sihao Li, Guo Fu, Yulong Cui, Qiang Li, Caiyi Wu, Shangjun Han, Shengyi Wang, Guoliang Li, and Ge Yu. 2023. GeoGauss: Strongly Consistent and Light-Coordinated OLTP for Geo-Replicated SQL Database. *SIGMOD* 1, 1 (2023), 62:1–62:27. https://doi.org/10.1145/3588916

[38] Tobias Ziegler, Philip A Bernstein, Viktor Leis, and Carsten Binnig. 2023. Is Scalable OLTP in the Cloud a Solved Problem?. In *CIDR*.