PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory

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

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

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

Feifei Li
lifeifei@alibaba-inc.com
Alibaba Group

Bo Wang
xiangluo.wb@alibaba-inc.com
Alibaba Group

Jing Fang
hangfeng.fj@alibaba-inc.com
Alibaba Group

Chuan Sun
hualuo.sc@alibaba-inc.com
Alibaba Group

Yuhui Wang
yuhui.wyh@alibaba-inc.com
Alibaba Group

ABSTRACT

Primary-secondary databases often have limited write throughput as they rely on a single primary node. To improve this, some systems use a shared-nothing architecture for scalable multi-primary clusters. However, these face performance issues due to distributed transaction overheads. Recently, shared-storage-based multi-primary cloud-native databases have emerged to avoid these issues, but they still struggle with performance in high-conflict scenarios, often due to expensive conflict resolution and inefficient data fusion.

This paper proposes PolarDB-MP, an innovative multi-primary cloud-native database that leverages both disaggregated shared *memory* and *storage*. In PolarDB-MP, each node has equal access to all data, enabling transactions to be processed on individual nodes without the need for distributed transactions. At the core of PolarDB-MP is the Polar Multi-Primary Fusion Server (PMFS), built on disaggregated shared memory. PMFS plays a critical role in facilitating global transaction coordination and enhancing buffer fusion, seamlessly integrated with RDMA for minimal latency. Its three main functionalities include Transaction Fusion for transaction ordering and visibility, Buffer Fusion providing a distributed shared buffer, and Lock Fusion for cross-node concurrency control. Moreover, PolarDB-MP introduces an LLSN design, establishing a partial order for write-ahead logs generated across different nodes, accompanied by a tailored recovery policy. Our evaluations of PolarDB-MP demonstrate its superior performance when compared to the state-of-the-art solutions. Notably, PolarDB-MP is already in production and undergoing commercial trials at Alibaba Cloud. To our knowledge, PolarDB-MP is the first multi-primary cloud-native database that utilizes disaggregated shared memory and shared storage for transaction coordination and buffer fusion.

*Both authors contributed equally to this work.
†Hao Chen is the corresponding author.

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0422-2/24/06…$15.00
https://doi.org/10.1145/3626246.3653377

CCS CONCEPTS

• **Information systems → Relational database model**.

KEYWORDS

cloud-native database, multi-primary database, disaggregated shared memory

ACM Reference Format:
Xinjun Yang*, Yingqiang Zhang*, Hao Chen†, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory . In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3626246.3653377

1 INTRODUCTION

Many cloud-native databases (such as AWS Aurora [49], Azure Hyperscale [14, 26], Azure Socrates [1] and Alibaba PolarDB [27]) have adopted a primary-secondary architecture based on a disaggregated shared storage architecture, typically consisting of one primary node and one or more secondary nodes. However, this primary-secondary model faces a significant bottleneck in write-heavy workloads due to the usage of a single primary node. Additionally, in scenarios where the primary node fails or shuts down for upgrade, one of the secondary nodes will be promoted to the primary role. This transition, while necessary, leads to a brief period of downtime during the failover process. Consequently, there is a growing demand for multi-primary cloud-native databases that can provide enhanced scalability for write-intensive operations (especially for highly concurrent workloads) as well as improved high availability with seamless failover capability.

The two most popular multi-primary architectures are shared-nothing and shared-storage. In the shared-nothing architecture (*e.g.*, Spanner[11], DynamoDB [15], CockroachDB [15], PolarDB-X [6], Aurora Limitless [2], TiDB [19] and OceanBase [55], etc), the whole database is partitioned. Each node is independent and accesses data exclusively within its designated partition. When a transaction spans multiple partitions, it must require cross-partition distributed transaction mechanisms, such as the two-phase commit policy, which typically induces significant extra overhead [57, 61].

The shared-storage architecture is essentially the opposite, where all data is accessible from all cluster nodes, such as IBM pureScale [20], Oracle RAC [9], AWS Aurora Multi-Master (Aurora-MM) [3] and Huawei Taurus-MM [16]. IBM pureScale and Oracle RAC are the two traditional database products based on shared-storage architecture. They rely on expensive distributed lock management and high network overhead. They are usually deployed in their dedicated machines and are too rigid for a dynamic cloud environment. Consequently, they usually have a much higher total cost of ownership (TCO) than modern cloud-native databases. Aurora-MM and Taurus-MM are the recently proposed products for a multi-primary cloud-native database. Aurora-MM utilizes optimistic concurrent control for write conflict, thus inducing a substantial abortion rate when conflicts occur. In some scenarios, its four-node cluster's throughput is even lower than that of a single node [16]. On the contrary, Taurus-MM adopts the pessimistic concurrent control but it relies on page stores and log replays for cache coherence. As such, it suffers from the high overhead of concurrent control and data synchronization. The eight-node cluster only improves the throughput by 1.8× compared to the single-node version in the read-write workload with 50% shared data.

To address these challenges, this paper proposes PolarDB-MP, a multi-primary cloud-native database via disaggregated shared memory (and with a shared storage). PolarDB-MP inherits the disaggregated shared storage model from PolarDB, allowing all primary nodes equal access to the storage. This enables a transaction to be processed in a node without resorting to a distributed transaction. In contrast to optimistic concurrency control, PolarDB-MP employs pessimistic concurrency control to mitigate transaction aborts caused by write conflicts. Different from other cloud databases that rely on log replay and page servers for data coherence between nodes, PolarDB-MP uses disaggregated shared memory for efficient cache and data coherence. With the growing availability of RDMA networks in cloud vendors' data centers [54], PolarDB-MP is intricately co-designed with RDMA to enhance its performance.

The core component of PolarDB-MP is *Polar Multi-Primary Fusion Server* (PMFS), which is built on disaggregated shared memory. PMFS comprises *Transaction Fusion*, *Buffer Fusion* and *Lock Fusion*. Transaction Fusion facilitates transaction visibility and ordering across multiple nodes. It utilizes a Timestamp Oracle (TSO) for ordering and allocates shared memory on each node to store local transaction data that are accessible remotely by other nodes. This decentralized transaction management approach via shared memory ensures low latency and high performance in global transaction processing. Buffer Fusion implements a distributed buffer pool (DBP), also based on disaggregated shared memory and accessible by all nodes. Nodes can both push modified data to and retrieve data from the DBP remotely. This setup allows swift propagation of changes from one node to others, ensuring cache coherency and rapid data access. Lock Fusion efficiently manages both page-level and row-level locking schemes, thus enabling concurrent data page access across different nodes while ensuring physical data consistency and maintaining transactional consistency.

PolarDB-MP also adeptly manages write-ahead logs across multiple nodes, using a logical log sequence number (LLSN) to establish a partial order for logs from different nodes. LLSN ensures that all logs associated with a page are maintained in the same order as they were generated, thereby preserving data consistency and simplifying recovery processes. On the other hand, PolarDB-MP designs a new recovery policy based on the LLSN framework, effectively managing crash recovery scenarios.

Additionally, PolarDB-MP's message passing and RPC operations are enhanced by a highly optimized RDMA library, boosting overall efficiency. These advanced designs position PolarDB-MP as a robust, efficient solution for multi-primary cloud-native databases.

We summarize our main contributions as follows:

- We propose a multi-primary cloud-native relational database via disaggregated shared memory over shared storage (the first of its kind), delivering high performance and scalability.
- We leverage the RDMA-based disaggregated shared memory to design and implement *Polar Multi-Primary Fusion Server* (PMFS) that achieves global buffer pool management, global page/row locking protocol, and global transaction coordination.
- We propose an LLSN design to offer a partial order for write-ahead logs generated across different nodes, accompanied by a tailored recovery policy.
- We thoroughly evaluate PolarDB-MP with different workloads and compare it with different systems. It shows high performance, scalability, and availability.

This paper is structured as follows. First, we present the background and motivation in Section 2. Then we provide PolarDB-MP's overview and detailed design in Section 3 and Section 4. Next, we evaluate PolarDB-MP in Section 5 and review the related works in Section 6. Finally, we conclude the paper in Section 7.

2 BACKGROUND AND MOTIVATION

2.1 Single-primary cloud-native database

Nowadays, there are many cloud-native databases based on the primary-secondary architecture, such as Aurora [49], Hyperscale [14, 26], Socrates [1] and PolarDB [27]. Figure 1 depicts the typical architecture of a primary-secondary cloud-native database. It usually comprises a primary node for processing both read and write requests and one or more secondary nodes dedicated to handling read requests. Each node is a complete database instance, equipped with standard components. However, a distinctive feature of these databases, as opposed to traditional monolithic databases, is the usage of disaggregated shared storage. This shared storage ensures fault tolerance and consistency, and uniquely, adding more secondary nodes does not necessitate additional storage. This contrasts with conventional primary/secondary database clusters, where each node maintains its own storage.

While the primary-secondary-based databases offer certain benefits, they face significant challenges under write-heavy workloads. Scaling out to improve performance is not an option in such architecture, and scaling up is limited by the available resources on the physical machine. Cloud providers typically maximize the use of physical resources, but since these resources are shared across

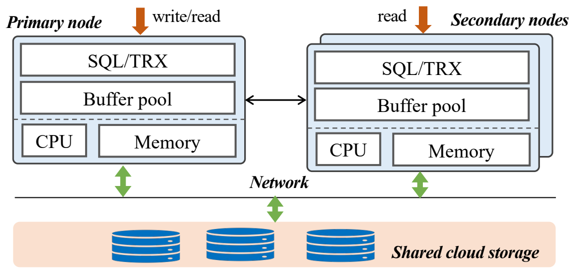


Figure 1: The architecture of a typical primary-secondary-based cloud-native database.

many instances, there's often little room for scaling up a specific instance on the same machine (e.g., adding CPU/memory resources). Furthermore, migrating to a less crowded machine can result in significant downtime and still faces the limitations of a single machine's capacity. Another critical challenge in a single-primary cluster is the issue of failover. If the primary node crashes, it takes time for a secondary node to assume its role, leading to brief downtime. Therefore, to achieve high scalability and availability, the multi-primary database is increasingly becoming a necessity.

2.2 Shared-nothing architecture

The shared-nothing architecture [43] is a prevalent model for scaling out, enjoying widespread use in both distributed computing [52, 56] and distributed databases [6, 19, 25, 45]. In this architecture, each request is processed independently by a single node (processor/memory/storage unit) within a cluster. The primary aim is to eliminate contention between nodes, as well as to remove single points of failure, thus allowing the system to continue functioning despite individual node failures. This also permits upgrades of individual nodes without necessitating a system-wide shutdown.

In databases based on shared-nothing architecture, it typically partitions data across several nodes, with each node having exclusive access (both read and write) to the data in its partition. This architecture offers robust scalability when application traffic is effectively partitioned. However, if a transaction spans more than one partition, it requires distributed transaction processing to maintain the transaction's ACID properties. Managing efficient synchronization among nodes to ensure these properties while maintaining performance is challenging [57, 61], which hinders scalability [13, 21, 40, 44, 47]. Various techniques, like locality-aware partitioning [13, 35, 38, 58], speculative execution [36], consistency levels [23] and the relaxation of durability guarantees [24], have been proposed to mitigate this issue. However, these solutions often lack transparency and require users to understand their intricacies and carefully design their databases. Additionally, when the system needs to scale in or out, data may need to be repartitioned, a process often fraught with heavy, time-consuming data movements [6]. Overall, while shared-nothing architecture offers significant benefits for scalability, these advantages come with their own set of challenges and complexities.

2.3 Shared-storage architecture

The shared-storage architecture represents a stark contrast to the shared-nothing model, as it allows each node within the cluster to read and write any record in the entire database. In such a setup, to facilitate concurrent transaction execution across different nodes, global coordination of transaction execution is necessary. This typically involves mechanisms like a global lock manager and a centralized Timestamp Oracle (TSO). Conventional shared-storage-based databases such as IBM pureScale [20] and Oracle RAC [9] embody this architecture. However, detailed descriptions of their implementation are sparse. These systems often struggle with the complexities and costs associated with distributed lock management and high network overhead. Additionally, they were designed prior to the advent of cloud computing and are typically deployed on their own dedicated hardware, making them unsuited for modern cloud environments. Their rigidity in this context often results in a significantly higher Total Cost of Ownership (TCO) compared to modern cloud-native databases.

Aurora-MM [3] and Taurus-MM [16] are the two recent proposals to bring the multi-primary database to the cloud. Aurora-MM adopts the shared storage architecture and employs optimistic concurrency control to manage write conflicts. This approach can lead to high performance when there is no contention between nodes. However, a significant downside is that, in scenarios with conflicts, such as when different nodes attempt to modify the same data page simultaneously, it results in a high rate of transaction aborts. In such cases, it reports such write conflicts to the application as a deadlock error, requiring applications to detect these errors, roll back transactions, and retry them later. This not only diminishes throughput and consumes additional resources, but also presents a challenge as many applications are not adept at handling high abort rates. According to Taurus-MM 's research [16], Aurora-MM's four-node cluster only shows a throughput improvement of less than 50% compared to a single node under the SysBench read-write workload with a mere 10% data sharing between nodes (the detailed configuration is presented in Section 5). Moreover, in a SysBench write-only workload with 10% shared data, the four-node cluster's throughput is even lower than that of a single node.

To improve performance, Taurus-MM utilizes the pessimistic concurrency control. It introduces a Vector-scalar clocks algorithm for transaction ordering and a hybrid page-row locking mechanism to enable concurrent transaction executions and data access on different nodes. However, this approach encounters issues with buffer coherency. When a node requests a page that has been modified by another node, it must request both the page and corresponding logs from the page/log stores, and then apply the logs to obtain the latest version of the page. This process typically involves storage I/Os, which can impact performance, and the log application also consumes extra CPU cycles. In their evaluations [16], the throughput of Taurus-MM 's eight-node cluster is approximately 1.8× that of a single node under the SysBench write-only workload with 30% shared data, illustrating the trade-offs and challenges in optimizing multi-primary cloud databases.

2.4 MVCC and transaction isolation

Multi-version concurrency control (MVCC) is currently the most popular transaction management scheme in modern database [10, 32, 53] and it is used in almost every major relational database. MVCC differs from traditional methods by not overwriting data with updates; instead, it creates new versions of the data item. This results in multiple versions of each data item being stored simultaneously. The visibility of these versions to a transaction is determined by the database's isolation level, with snapshot isolation being a common choice. Under snapshot isolation, each transaction sees a snapshot of the database as it was at the start of the transaction, ensuring consistent data view throughout its duration. Transactions use timestamps or transaction IDs to identify the correct versions of data to read, providing a mechanism that effectively isolates read and write operations from each other. This isolation is achieved without relying on locks, thereby reducing potential bottlenecks and improving performance, especially in environments with high levels of concurrency.

However, in a shared-storage multi-primary database, determining the visibility of a data item under MVCC poses a significant challenge. When a transaction reads a version of a data item on one node, it cannot locally determine the data's visibility, as the item may have been concurrently updated by a transaction on another node. To accurately determine if the current version is visible to a transaction's read view under the given isolation level, a node must have access to global transaction information. Synchronizing this global transaction information across different nodes typically involves considerable overhead. This complexity is a critical issue that hasn't been thoroughly addressed in existing systems like Aurora-MM or Taurus-MM. In PolarDB-MP, we tackle this challenge with an innovative design, Transaction Fusion with PMFS. Our approach decentralizes the whole transactions' information on each node. Each node only maintains its own local transactions' information and can be accessible by other nodes via RDMA. This enables a transaction to accurately and efficiently determine a data item's visibility under MVCC, supporting various isolation levels.

2.5 Opportunities with RDMA

A multi-primary database cluster inevitably needs message passing for data synchronization and concurrency control over different nodes, making the network a significant concern in improving performance. Fortunately, with advancements in network technology, such as the ultra-low latency and 400Gb/s throughput offered by devices like the ConnectX-7 InfiniBand adapter [34], and the widespread availability of RDMA (Remote Direct Memory Access) in commodity clusters [17, 62], network bottlenecks are increasingly becoming less significant. At Alibaba, the RDMA network is a core infrastructure component, and PolarDB is co-designed with RDMA [54]. In the Alibaba public cloud, PolarDB clusters are all outfitted with RDMA networks. The existence of these RDMA networks provides a unique opportunity for PolarDB to develop its multi-primary variant, PolarDB-MP, with much less network hardware costs. PolarDB-MP leverages the RDMA-based disaggregated shared memory to directly transmit data pages between different nodes. Additionally, both the lock manager and transaction coordination messages are transmitted via the RDMA network.

3 POLARDB-MP OVERVIEW

Figure 2 offers a comprehensive overview of its architecture. PolarDB-MP adopts a disaggregated shared storage, using PolarStore and PolarFS [7], but it also maintains compatibility with any other disaggregated shared storage solutions. To effectively support multiple

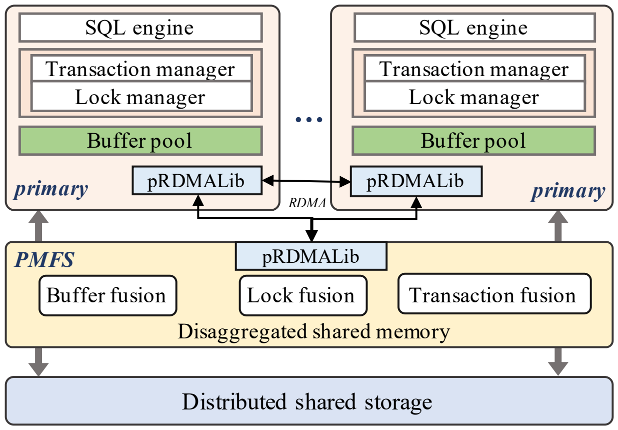


Figure 2: The architecture of PolarDB-MP.

primary nodes on the shared storage, we introduce the Polar Multi-Primary Fusion Server (PMFS) based on the disaggregated memory architecture (on top of the disaggregated shared storage). It enables all nodes to have equal access to the shared data in memory and concurrently process read/write requests. The key function of PMFS is to manage the global transaction concurrency and buffer coherency across different nodes.

PMFS is implemented with a disaggregated shared memory, typically consisting of multiple nodes and providing high availability. PMFS comprises three core components: Transaction Fusion, Buffer Fusion and Lock Fusion. Transaction Fusion aims to manage global transaction processing, guaranteeing the transaction's ACID. It maintains a global Timestamp Oracle (TSO) to order transaction committing. Moreover, Transaction Fusion supports global transaction visibility to achieve snapshot isolation under multi-version concurrency control (MVCC).

Buffer Fusion, equipped with a distributed buffer pool (DBP), plays a crucial role in maintaining buffer coherency across all nodes. The DBP buffers a number of data pages for fast access. When a node makes updates to a data page, it pushes this updated page to the DBP at an opportune moment. Subsequently, if another node requires this page, it can retrieve the most recent version directly from the DBP. The connection between the primary nodes and the DBP is facilitated by a high-speed RDMA network, ensuring rapid movement of data pages across different nodes. The DBP leverages disaggregated shared memory to store these data pages. Since each node has a local buffer, Buffer Fusion is also tasked with implementing buffer coherency for all nodes.

Lock Fusion is responsible for implementing two locking protocols: page-locking (PLock) and row-locking (RLock). The PLock protocol ensures the physical consistency of a page during concurrent access by different nodes. A node can write or read a page only if it holds the corresponding exclusive/shared PLock. Meanwhile, the RLock protocol guarantees the transactional consistency of user data, and typically obeys the two-phase locking protocol. To reduce the messaging overhead caused by RLock requests and avoid the centralized metadata of the locking state, PolarDB-MP embeds row lock information within the row data itself, maintaining only the wait-for relation in Lock Fusion.

Additionally, PolarDB-MP proposes the logical log sequential number (LLSN) to order the redo logs (write-ahead log) across different nodes. LLSN provides a partition order for all redo logs generated by different nodes, which guarantees each page's redo log is maintained in the order they are generated. This scheme ensures the correctness of the recovery after a system crash.

In PolarDB-MP, all communications between the primary nodes and PMFS leverage one-sided RDMA or RDMA-based RPC. This approach ensures efficient and low-latency message passing. Similar to PolarDB, PolarDB-MP also incorporates a standby node to ensure high availability across regions. Changes occurring in the primary cluster are synchronized to the standby cluster using the write-ahead log, thereby maintaining the integrity and continuity of the database even in the event of a regional failure.

Given the extensive existing literature on disaggregated shared memory [4, 31, 42, 46] and our previous work integrating it into a relational database (PolarDB) [8, 41, 60], we do not delve into its detailed implementation in this paper, as it is not the primary focus of our contribution.

4 DESIGN AND IMPLEMENTATION

4.1 Transaction Fusion

In a multi-primary database like PolarDB-MP, transaction ordering and visibility are critical for maintaining isolation and consistency. To achieve high performance and maintain simplicity, PolarDB-MP employs a Timestamp Oracle (TSO) for transaction ordering. The TSO is an integral part of the Transaction Fusion on the PMFS. As a transaction reaches its commit phase, it requests a commit timestamp (CTS) from TSO. This CTS is a logical, incrementally assigned timestamp, ensuring that orderly transaction processing is maintained. The CTS is usually fetched by using a one-sided RDMA operation, which is typically completed within several microseconds and has been found to not be a bottleneck in our tests.

To efficiently manage all transaction information in the cluster, PolarDB-MP employs a decentralized method to distribute this information across all nodes. Each node in PolarDB-MP reserves a small portion of memory to store its local transaction information. A node can remotely access the other node's transaction information via one-sided RDMA. Figure 3 illustrates this design. Every node maintains the information for its local transactions in the *Transaction Information Table* (TIT). TIT plays a crucial role in managing transactions by maintaining four key fields for each transaction: *pointer*, *CTS*, *version* and *ref*. The *pointer* is the transaction object pointer, *CTS* marks the transaction's commit timestamp, *version* identifies different transactions in the same slot, and *ref* is a flag indicating if any transactions are waiting for this one to release its locks, which will be introduced in Section 4.3.2.

When a transaction starts on a node, a locally incremental and unique ID will be allocated to this transaction. A free TIT slot (identified by a null transaction pointer) is then allocated to this transaction, storing its pointer and setting other fields appropriately. As TIT slots can be reused, the version field differentiates transactions occupying the same slot at different times, incrementing with each new transaction. Initially, a transaction's ref is set to zero and CTS is set as *CSN_INIT*. To globally identify a transaction, PolarDB-MP combines the *node_id*, *trx_id*, *slot_id*, and *version* into a global transaction ID (g_trx_id). With this g_trx_id, any node can remotely access a transaction's CTS from the target node through

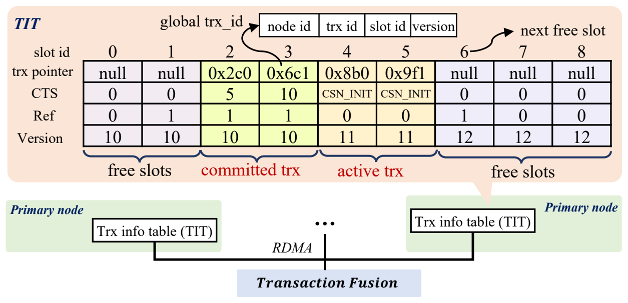


Figure 3: The design of transaction information table (TIT).

RDMA. The whole transaction's information is distributed on different nodes. Each node only needs to maintain its local transactions' metadata. The transaction ID and TIT slot can be allocated locally without communicating with a coordinator, reducing overhead and simplifying transaction management in the multi-primary system. Additionally, PolarDB-MP adds two extra metadata fields for each row to store the g_trx_id and CTS. These two fields can help to determine the data's visibility.

Transaction visibility. Like most modern databases, PolarDB-MP also implements MVCC, which is a popular technique used to maintain consistency while allowing concurrent transactions. In PolarDB-MP, the process of updating a row involves storing the current global transaction ID (g_trx_id) in the row's metadata. This g_trx_id is crucial for tracking the transaction that last modified the row. When a transaction reaches the commit stage, it updates the CTS in the metadata of the rows affected by that transaction, provided these rows are still in the buffer. If a row is no longer in the buffer at the time of committing the transaction, its CTS remains at the default value (*CSN_INIT*).

These two pieces of metadata, g_trx_id and the CTS, are instrumental in determining the visibility of a row to a given transaction. Each transaction in PolarDB-MP is assigned a read view, which consists of its own g_trx_id and the current CTS fetched from the TSO. This read view plays a crucial role in the MVCC mechanism. A row is visible to a transaction if the row is committed before the transaction's read view. When a transaction performs a read operation, it initially accesses the latest version of the row. However, if this version is not visible to the transaction (based on its read view), it utilizes undo logs to construct the previous versions of the row. This process continues until it finds a version of the row that is visible to the transaction.

However, in a multi-primary database, the challenge is how to get a row's CTS as the row may be updated by another node. PolarDB-MP address this by using the TIT design, outlined in Algorithm 1. If the row's CTS field is filled with a valid value, not the initialized value (*CSN_INIT*), it can directly get the CTS from the row (line 2-5). But, in some cases where the row's CTS is not filled (typically because the row was evicted from the buffer upon transaction commit), the TIT is used to acquire the CTS of this row. It needs first to retrieve the transaction ID (g_trx_id) from the row's metadata fields. Then it can get the corresponding TIT slot with this g_trx_id. If the g_trx_id belongs to the current node, the TIT slot can be directly read from the local TIT (line 9). Otherwise, it needs a one-sided remote RDMA read from the target node (line 11). It's crucial to ensure that the g_trx_id version in the TIT matches the one being checked (line 13). A mismatch indicates that the TIT slot has been reused by a different transaction, implying that the original transaction has already been committed. In this case, we can return a minimal CTS value to indicate this row is visible to all transactions (line 15). This is because a TIT slot is only freed and reused if its CTS is smaller than the CTS of all active views. If the slot is valid, we can get the CTS from the slot. In case the transaction is still active, it returns a maximum CTS value to indicate that it is not visible to any transaction except itself (line 16).

The design of the TIT is notably RDMA-friendly, allowing for efficient remote access to TIT slots across nodes via a one-sided RDMA interface. During system startup, each node synchronizes the starting address of its TIT with other nodes. Each global transaction ID in the system includes a reference to a specific TIT slot, indicating its position or offset within the TIT. When a node needs to access information from a TIT slot located on another node, it first calculates the remote address. This calculation is based on the synchronized starting address of the TIT and the offset of the specific TIT slot. Once the remote address is determined, the node can directly access the required data from the TIT slot of the remote node using one-sided RDMA.

TIT recycle. The TIT has a limited memory footprint. To manage this efficiently, a background thread reclaims and frees up used TIT slots for reuse. A transaction's TIT slot is eligible for recycling when its changes are visible to all other transactions. This implies that if a slot is reused, the associated transaction's changes are visible to any transaction. To implement this, each node runs a background thread that sends its minimal view to Transaction Fusion. Transaction Fusion consolidates these views to form a global minimum view, which is then broadcast to all nodes. The nodes recycle TIT slots if their CTS is smaller than the global minimal view's CTS.

Timestamp fetching. For read transactions, acquiring the current CTS from the TSO is necessary. To reduce the overhead of frequent CTS fetching, especially under the read committed isolation, we utilize the Linear Lamport Timestamp approach from PolarDB-SCC [54]. It allows a request to reuse a recently fetched timestamp if it was obtained after the request's arrival, significantly cutting down on the number of timestamp fetching operations while maintaining consistent isolation levels. The correctness and effectiveness of this approach have already been proven in PolarDB-SCC [54].

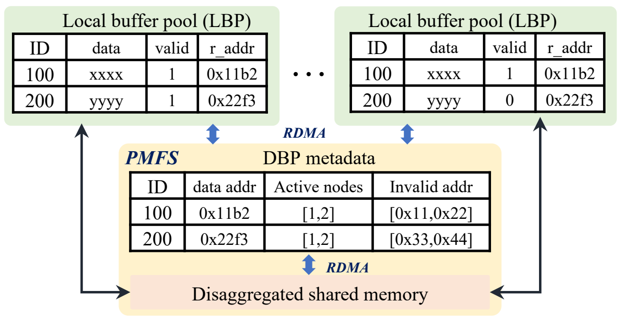
4.2 Buffer fusion

Each PolarDB-MP node can update any data page, leading to frequent transfers of pages between different nodes. To facilitate fast cross-node data movement, PolarDB-MP proposes Buffer Fusion, where nodes push their data pages to Buffer Fusion's distributed buffer pool (DBP) and subsequently another node can then access a page modified by its peers from this DBP. In this case, the pages can be efficiently moved between different nodes with low latency. Additionally, the DBP utilizes disaggregated shared memory for its buffer, allowing for a sizable and scalable buffer pool.

Figure 4 presents the Buffer Fusion design. Each node has its own local buffer pool (LBP), a subset of Buffer Fusion's DBP. Within the LBP, we introduce two extra fields for each page's metadata: *valid* and *r_addr*. The *valid* field indicates whether the page has

Algorithm 1 Get the CTS of a row

```
 1: function GetCTSForRow(row)
 2:     if row.CTS != CTS_INIT then
 3:         # the row's CTS is filled with a valid value
 4:         return row.CTS
 5:     end if
 6:     # the row's CTS is not filled and get it from TIT
 7:     g_trx_id = GetTrxID(row)
 8:     if g_trx_id.node_id == CURRENT_NODE_ID then
 9:         TIT_slot = GetTrxInfoLocally(g_trx_id)
10:     else
11:         TIT_slot = GetTrxInfoRemotely(g_trx_id)
12:     end if
13:     if TIT_slot.version != g_trx_id.version then
14:         # slot is reused by other transaction
15:         return CSN_MIN
16:     end if
17:     if TIT_slot.CTS == CTS_INIT then
18:         # the transaction is still active
19:         return CSN_MAX
20:     end if
21:     return TIT_slot.CTS
22: end function
```



ID	data	valid	r_addr
100	xxxx	1	0x11b2
200	yyyy	1	0x22f3

ID	data	valid	r_addr
100	xxxx	1	0x11b2
200	yyyy	0	0x22f3

ID	data addr	Active nodes	Invalid addr
100	0x11b2	[1,2]	[0x11,0x22]
200	0x22f3	[1,2]	[0x33,0x44]

Figure 4: The design of buffer fusion.

been modified by other nodes, and *r_addr* points to the page's address in the DBP. When accessing a page from its LBP, a node first checks the page's validity. If it's not valid, it retrieves the page from DBP using its *r_addr* via the one-sided RDMA interface. In the Buffer Fusion, it also maintains some metadata for each page for the purposes of tracking the node IDs that have copies of the page, the addresses of their invalid flags, and the page's address in the DBP. When a new version of a page is stored in the DBP, Buffer Fusion remotely invalidates the copies on other nodes via the address of the invalid flag. In the LBP, the dirty pages (pages modified since being loaded into the LBP) are periodically flushed to the DBP in the background, or on-demand when releasing the corresponding PLock. PolarDB-MP adopts the "no force at commit" policy, commonly used to enhance performance in many databases. In PolarDB-MP, before flushing a dirty page to the DBP, PolarDB-MP also forces the corresponding logs to storage. This ensures that a page can be evicted from the LBP if it has been flushed to the DBP, and it can be recovered from logs in the event of a DBP failure.

Page access. In PolarDB-MP, each node can directly access the page if it is in the LBP and valid. If invalid, it retrieves the page from the DBP via the one-sided RDMA interface. If the page is not

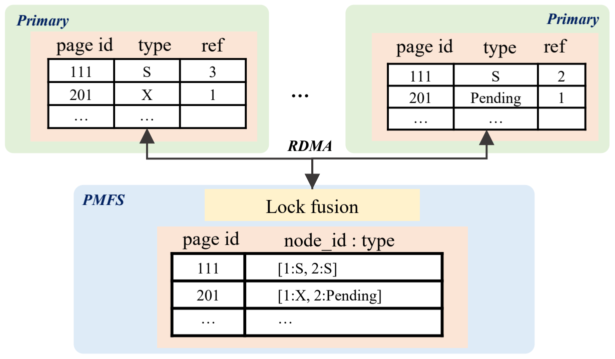


Figure 5: The design of PLock.

in the LBP, an RPC call to Buffer Fusion checks its presence in the DBP. If the page is in the DBP, it adds the primary node's ID to the page's active nodes and returns the page's address to the node for remote reading from DBP. If the page is not in the LBP or DBP, it is read from the shared storage. Once loaded by a node, the page is registered to the DBP and remotely written to it. Concurrent page access across different nodes and the consistency of the database's internal structures, like B-tree pages, are maintained using PLock, which will be detailed in the subsequent section.

4.3 Lock fusion

Lock Fusion implements both the page-locking (PLock) and row-locking (RLock) protocols. PLock is similar to the page latch in single-node databases, ensuring atomic access to a page and the consistency of internal structures. RLock, on the other hand, maintains transactional consistency across nodes, following the two-phase locking protocol, which is typically used in many databases.

4.3.1 PLock protocol. The PLock is designed to maintain physical data consistency. It does not apply to concurrent page access within a single node. The internal page concurrency control within a single node is still the same as before. PLock is managed at the node level, as depicted in Figure 5. Each node keeps track of the PLock it holds or is waiting for, and a reference count indicates the number of threads using a particular PLock. Lock Fusion maintains all PLock information, tracking each lock's status. In PolarDB-MP, before performing any update/read to a page, the node must hold the corresponding X/S PLock. When a node requires a PLock, it first checks its local PLock manager to see if it already holds the required or a higher lock level. If not, it requests the PLock from Lock Fusion via RDMA-based RPC. Lock Fusion handles the request, checking for conflicts before responding. If a conflict exists, the requesting node is suspended and later awoken when its PLock request can be granted. Furthermore, when a PLock is released by a node, this change is communicated to Lock Fusion, which then updates the lock's status. Lock Fusion also notifies any nodes waiting for the released PLock, enabling them to proceed with their operations.

Lazy releasing. Due to the temporal locality [5], a page will be referenced again in the near future after it is accessed. Thus PolarDB-MP implements a strategy to minimize the PLock-related RPC overhead. Instead of releasing its PLock back to Lock Fusion immediately after use, a node decreases the reference count for the PLock. The lock becomes available for release once this count drops to zero, but it is still temporarily retained by the node. If the same node needs to acquire the PLock again, and the requested lock type is not stronger than the currently held type, the PLock can be granted locally. This method effectively reduces the RPC overhead for pages that are frequently accessed on the same node. In scenarios where a different node requests a conflicting lock type, Lock Fusion intervenes by sending a negotiation message to the node currently holding the lock. This message prompts the holding node to release the lock once its reference count reaches zero.

A node can locally grant a PLock if it already holds an equal or stronger type of the PLock. However, to prevent potential lock starvation of other nodes, when a node receives a negotiation message for a PLock, it cannot autonomously guarantee this PLock for its internal transactions. Instead, it must communicate with Lock Fusion, which manages the granting of locks to nodes in a First-In, First-Out order. This approach ensures a fair and efficient distribution of locks across different nodes, maintaining the balance between local optimization and global resource allocation.

Physical consistency. PLock maintains the physical consistency for both physical data and internal structures. To update or read a page on any node, the node must first acquire the appropriate exclusive or shared PLock for that page. This process ensures that when a node updates a page locally, other nodes are excluded from reading this page as they cannot obtain the shared PLock. If a node is about to release an exclusive PLock and the corresponding page has been modified, the node will push this updated page to the DBP. Meanwhile, it will inform Buffer Fusion to invalidate the page on other nodes. As a result, if another node requests this page after holding the necessary PLock, it will discover that its local version is now invalid and will subsequently retrieve the latest version of the page from DBP. On the other hand, to guarantee the consistency of internal structure, such as B-tree, PolarDB-MP adopts a similar approach to that used in single-node databases. Changes to the B-tree structure, such as page splits or merges, are executed within internal mini-transactions. During these operations, the corresponding pages' PLocks are held until the mini-transaction is committed. It ensures that no transaction, whether within the same node or on other nodes, encounters an inconsistent B-tree structure. This careful management of PLock thus plays a crucial role in preserving the integrity and consistency of the database's structure and data across its multi-primary environment.

4.3.2 RLock protocol. Within a single-node database, row-level locking is typically employed to ensure transaction consistency and isolation. In PolarDB-MP, where multiple transactions may run concurrently on different nodes and potentially update the same data, a global row-locking (RLock) protocol is indispensable. To reduce message traffic associated with row locking, PolarDB-MP embeds row lock information directly within each row and only maintains the wait-for relation on Lock Fusion. For each row, an additional field indicates the locking transaction's ID. When a transaction attempts to lock a row, it simply writes its global transaction ID (as introduced in Section 4.1) into this field. If the row's transaction ID field is already taken by an active transaction, a conflict is detected, and the current transaction must wait. In PolarDB-MP, when attempting to update a row, it must already hold an X PLock lock on the page containing the row. So only one

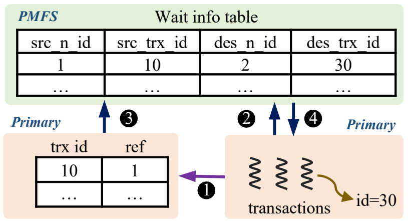


Figure 6: The design of the RLock policy.

transaction has access to write a row's locking information, and then only one transaction can successfully lock this row. Similar to the singe-node database, RLock protocol also obeys the two-phase locking policy that locks are held until the transaction is committed.

In the RLock design, determining whether a row is locked by another transaction involves checking the active status of a transaction. In a multi-primary database, ascertaining whether a given global transaction ID represents an active transaction can be challenging. However, PolarDB-MP's TIT design facilitates this process efficiently. With a global transaction ID, which combines node_id, local trx_id, slot_id, and version, one node can retrieve the transaction's CTS from either the local TIT or remote TIT on another node. If this CTS is populated with a valid value or if the corresponding TIT slot has been reused, it indicates that the transaction has been committed and is no longer active. Conversely, if the CTS field remains at its default value, the transaction is still considered active. To minimize excessive remote memory access, PolarDB-MP synchronizes the global minimal active transaction ID across nodes in the background. Thus, if a transaction's global ID is less than this global minimal active transaction ID, the transaction is no longer active. This approach streamlines the process of lock checking and enhances the efficiency of transaction management in the multi-primary database.

The transaction ID in the row functions as a lock indicator. So this protocol only supports exclusive (X) lock. The shared (S) lock on a row is not supported in PolarDB-MP, but it's acceptable. In PolarDB-MP or many other databases (*e.g.*, the MySQL-variants), most read requests are snapshot reads, handled via multi-version concurrency control (MVCC), thus not requiring locks. When reading a row, a transaction fetches the latest version without needing a row lock, though the page must be S-locked through PLock. The transaction then verifies the visibility of the version based on its transaction ID. If the version is not visible, the transaction reconstructs a visible version using undo logs, while the S-lock on the page prevents other threads from modifying the row during this process. So discarding the S type of row-lock does not affect the processing of read requests. In some rare cases, it requires to X lock a record. PolarDB-MP will upgrade the S lock to the X lock. Generally, this design has little impact on performance but saves a lot of message passing.

Locking processing. We demonstrate the RLock processing in Figure 6. Consider a scenario where transaction T30 on node-2 attempts to exclusively lock (X-lock) a row. It finds that the row is already X-locked by another transaction (T10) by examining the row's metadata. In response, T30 first remotely adjusts the reference ('ref') field in the remote transaction's (T10's) metadata. This action signals that there is a waiting transaction (T30) pending the release of the lock by T10. Then T30 communicates with Lock Fusion, sending information about its wait status. This information is added to a *wait info table* in Lock Fusion. Once T10 completes its transaction and commits, it checks its 'ref' field. Finding that another transaction (T30) is waiting, T10 notifies Lock Fusion that it has committed. Lock Fusion, upon receiving T10's notification, consults the wait info table, and then notifies T30, which can now wake up and continue its process.

4.4 Logs ordering and recovery

Logging scheme. PolarDB-MP adopts an ARIES-style [30] logging technique, utilizing redo (write-ahead) logs for data recovery and undo logs for rolling back uncommitted changes. In PolarDB-MP, each node maintains its own sets of redo log and undo log files. This design enables different nodes to simultaneously synchronize these logs to the storage without the need for explicit concurrency control mechanisms. The persistency strategy for redo logs in PolarDB-MP remains consistent with conventional solutions. Specifically, before committing a transaction, the corresponding redo logs are synchronized to the storage system. This ensures the durability of committed changes. Within a node, each log entry in the redo log is assigned a unique and incremental log sequence number (LSN) at the time of generation. Importantly, this LSN also serves as the offset within the redo log file. Consequently, the order of persistence for redo logs reflects the order of their generation within a node.

LLSN. However, in PolarDB-MP, a notable challenge arises from the fact that different nodes can independently update the same page, leading to the generation of redo logs across these nodes. This situation results in multiple nodes having different redo logs for the same page. As each redo log records the change made to a specific page, the key to recovery in this context is applying the redo logs for the same page in the order they were generated, while logs from different pages can be applied in any sequence. This understanding leads to the realization that it's unnecessary to maintain a total order for all redo logs; instead, ordering needs to be ensured only for logs corresponding to the same page.

Addressing this challenge, PolarDB-MP introduces the logical log sequence number (LLSN), establishing a partial order for logs from different nodes, specifically ensuring that logs related to the same page are maintained in their generation order. However, LLSN doesn't impose any specific order on logs from different pages, which is also unnecessary. To implement this, each node maintains a node-local LLSN that automatically increments with every log generation. When a node updates a page and generates a log, the new LLSN is recorded in the page metadata and also assigned to the corresponding log. If a node reads a page from storage or the DBP, it updates its local LLSN to match the accessed page's LLSN, provided that the page's LLSN exceeds the node's current LLSN. This ensures that the node's LLSN remains synchronized with the pages it accesses. Subsequently, when a node updates a page and generates logs, its LLSN is incremented, guaranteeing that the new LLSN is larger than that of any node that previously updated the page. Thanks to the PLock design, only one transaction can update a page at a time. Thus, when a page is sequentially updated on different nodes, the LLSN effectively maintains the logs in their generation order. Moreover, logs must be persisted in the order they are generated. Within one node, this is guaranteed by persisting logs in LSN order. When a page is updated across two nodes, one node pushes its updated page to the DBP before releasing the PLock, allowing the next node to retrieve it from the DBP. PolarDB-MP forces that a node persists all logs related to a page before pushing the page to the DBP. This ensures that, when a node updates a page, previous logs related to the page, generated by other nodes, are already stored to the storage. Consequently, logs associated with the same page are persistently ordered as they are generated, maintaining data consistency across multiple nodes.

Recovery. During the recovery, it's essential to apply redo logs to restore modified data pages accurately. As previously outlined, the correct database recovery relies on applying the redo logs in the order of their LLSNs. While LLSNs within a single log file are always incremental, they may overlap when considering different log files on different nodes. The requirement during recovery is to apply the logs that belong to the same page in their LLSN order. The naive way is to load all the log files and sort the log entries with their LLSNs, but this requires too much memory to hold all the log data and wastes CPU cycles during the huge data sorting. To avoid this, PolarDB-MP only reads a small part of logs each time, then applies them. Each time, it only reads a chunk of data from each file and determines a bounded LLSN ($LLSN_{bound}$) in these chunks. $LLSN_{bound}$ can guarantee that all the remaining logs' LLSNs in the files are larger than $LLSN_{bound}$. PolarDB-MP only picks the log data whose LLSN are smaller than $LLSN_{bound}$ in these chunks and parses them for the application. The other log data whose LLSN are larger than $LLSN_{bound}$ will be left to the next batch.

Undo logs are also protected by its redo logs and they are recovered after applying all redo logs. In PolarDB-MP, distributed transactions are unnecessary as transactions can be executed on a single node. Consequently, we can perform rollbacks for uncommitted transactions in the same manner as single-node databases, directly applying the undo logs.

5 EVALUATION

5.1 Experimental setup

Test platform. PolarDB-MP is implemented with a commercial cloud-native database (PolarDB) with disaggregated shared storage. In our test, we use the PolarStore [7] as the storage layer. PolarDB-MP is already in production and undergoing commercial trials at Alibaba Cloud. Our evaluations are all conducted in our public cloud environment. Each of the underlying physical machines is equipped with 2 Intel Xeon Platinum 8369B @2.90GHz CPUs and 1TB DDR4 DRAM, running CentOS-7 OS. These physical machines are connected by a 100Gbps Mellanox ConnectX-6 network.

System configurations. At Alibaba, the most popular type of PolarDB instances is 8 vCPUs and 32GB memory (*8c32g*). We test PolarDB-MP with the same configuration (8c32g) in most test cases. We set the primary node's local buffer pool size to 24GB (75% of available memory space and this is a default setting in PolarDB). We also test PolarDB-MP with the 32 vCPU instances and especially deployed with 32 primary nodes, a total of more than one thousand vCPUs in a cluster. Finally, we adopt the *read committed* isolation in all systems during the evaluation. This is the default isolation level for many databases, and it is widely used for many user applications.

Workloads. We evaluate PolarDB-MP with three standard OLTP benchmarks (SysBench [22], TPC-C [12] and TATP [33]) and a real production workload from Alibaba (with a profiled mix of 3:2:5 insert:update:select ratio). Unless otherwise stated, we adopt the default configurations for all of these workloads.

Additionally, to evaluate the scalability and flexibility of PolarDB-MP in handling varying degrees of data sharing across nodes, we adapted SysBench following the configuration approach used in Taurus-MM [16]. In our multi-node cluster setup, tables were logically divided into N + 1 groups, where N represents the number of nodes. The first N groups of tables were designated as private, with each node being assigned to a specific group and exclusively accessing the tables within it. The last group was shared, allowing any node to access its tables. The degree of sharing was controlled by specifying a percentage X, where X% of queries targeted the shared tables, and the remaining queries were directed to the private tables of each node. This configuration allowed us to test PolarDB-MP's performance across a spectrum of data-sharing scenarios.

5.2 Overall Performance

SysBench workloads. We first utilized SysBench to benchmark PolarDB-MP's performance, as depicted in Figure 7. For this test, we used instances each equipped with 8vCPU and 32GB of memory. We configured SysBench with 40 tables per group, and each table contained 1 million records. We evaluate the throughput under different workloads, read-only, read-write, and write-only, varying the percentage of shared data from 0% to 100%. In Figure 7, we present the absolute throughput on the left y-axis and the relative throughput, normalized to a single-node setup, on the right y-axis for each workload type.

For the read-only workload, PolarDB-MP always demonstrates linear scalability because there is no contention between different nodes. This is attributed to the absence of contention between nodes, with the only additional overhead being the fetching of timestamps from the TSO. However, this overhead is minimized through the use of RDMA and Linear Lamport Timestamp [54], rendering it negligible in terms of performance impact. In read-write and write-only workloads, we observe near-linear scalability in scenarios with well-partitioned data (0% shared data). However, as the percentage of shared data increases, scalability starts to decline due to the added overheads of data synchronization and transaction coordination. Despite this, scalability remains considerable. Remarkably, in scenarios with 100% shared data, the eight-node cluster enhances throughput by 5.4 times in read-write workloads and 3 times in write-only workloads compared to a single-node deployment. These high performance and scalability are largely due to the efficiency of PolarDB-MP's PMFS. The PMFS plays a crucial role in managing data and transaction coordination efficiently, even under high data contention scenarios.

TATP performance. We then evaluate PolarDB-MP using the TATP benchmark, as depicted in Figure 8. TATP workloads consist of user-related queries, prominently involving unique subscriber IDs. This characteristic facilitates effective database partitioning by subscriber ID, significantly reducing contention between different nodes when different nodes access different partitions. In this test,

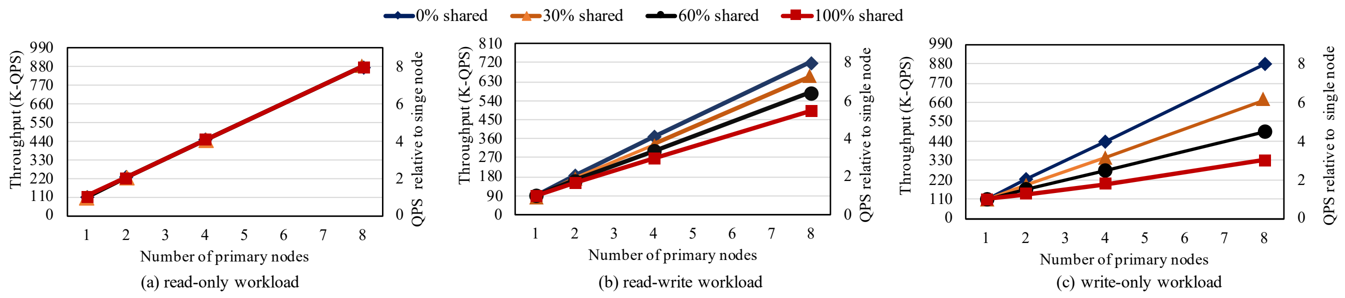


Figure 7: SysBench performance.

we configure TATP with 20 million subscribers per node. The results indicate that PolarDB-MP achieves linear scalability. When the workload is well-partitioned, different nodes handle separate data partitions This eliminates the necessity for inter-node data transfer. Under these circumstances, each data page is exclusively accessed by a single node, and the requirement for PLock arises only once for each data page at its initial access point. Consequently, PolarDB-MP does not incur additional overhead in TATP workloads, demonstrating efficient scalability and performance.

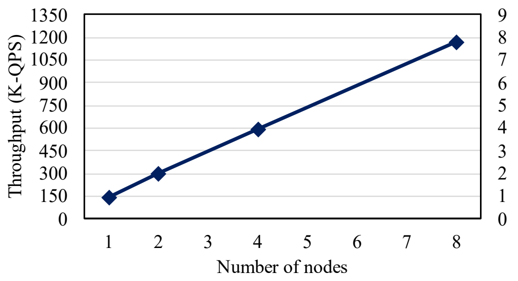


Figure 8: TATP performance.

TPC-C performance within a large-scale cluster. We then conducted an evaluation of PolarDB-MP within a large-scale cluster using the TPC-C benchmark. For this test, we configured PolarDB-MP to scale up to 32 nodes, each equipped with 32 virtual CPUs (vCPUs), culminating in a total of 1024 vCPUs. In line with previous research [16, 29], we set the think/keying time in TPC-C to zero. TPC-C primarily simulates transactions related to a single warehouse, with only about 11% of transactions involving cross-warehouse operations. In our testing, we focus on recording New Order transactions per second (tpmC) as well as transaction 95% latency, as depicted in Figure 9. The top subfigure's left y-axis indicates the absolute throughput, measured in tpmC, while the right y-axis shows the relative throughput, normalized to the performance in a single-node setup. The results indicate that PolarDB-MP demonstrates near-linear scalability from 1 to 24 nodes. Even when scaled out to 32 nodes, despite a minor decrease in scalability, there is a notable improvement in performance compared to the 24-node configuration. Notably, at 32 nodes, PolarDB-MP achieved an impressive throughput of 9.1 million tpmC, which is 24 times the throughput of a single node. Regarding the P95 latency, there is a slight increase as the number of nodes grows, suggesting that PolarDB-MP maintains latency effectively even as it scales out. This impressive performance of PolarDB-MP can be attributed to its efficient handling of scenarios involving shared data, minimizing overhead. Furthermore, the system does not incur extra overhead in situations where data is not shared among nodes. Additionally, the disaggregated shared memory architecture of PMFS enables PolarDB-MP to support a large number of nodes for scaling out. These capabilities allow PolarDB-MP to exhibit not only high performance but also significant scalability in the TPC-C benchmark. These findings underscore PolarDB-MP's potential as a robust solution for large-scale, distributed database environments, offering both performance efficiency and scalability.

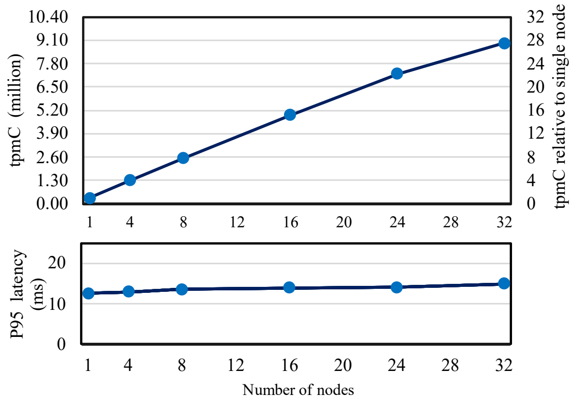


Figure 9: Performance of TPC-C within a large-scale cluster.

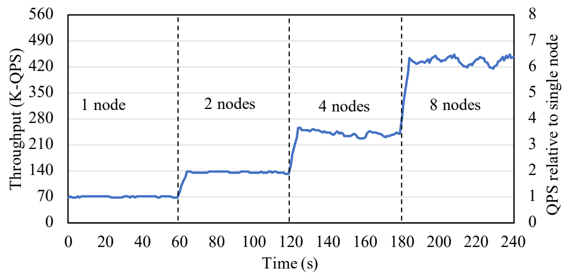


Figure 10: Alibaba production workload.

Production workload. Finally, we run a trading service workload from Alibaba's production environment. It is memory-intensive and contains a significant portion of write transactions. Figure 10 plots the throughput timelines of PolarDB-MP. This testing starts with only one node. We add more nodes to the cluster at the time of 60, 120, and 180 seconds. When having more nodes, the throughput is significantly improved. Since the workload is well-partitioned at the application level, it does not have too much conflict among the different nodes. Therefore, it shows near-linear scalability.

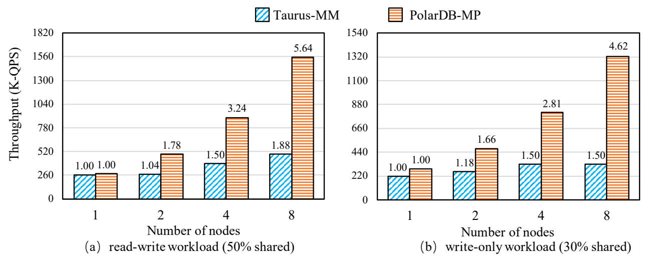


Figure 11: Compare with Taurus-MM

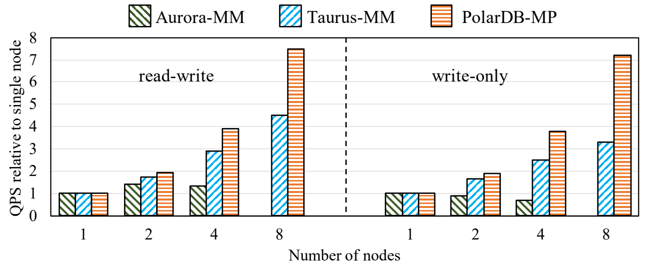


Figure 12: Compare with Aurora-MM and Taurus-MM within light conflict scenarios (10% shared).

5.3 Compare with Aurora-MM and Taurus-MM

Aurora MM [3] and Taurus-MM [16] represent the latest advancements in multi-primary cloud-native databases. However, they are currently neither available in the public cloud nor open-sourced [1]. Thus, we compare with Taurus-MM and Aurora-MM based on performance data from Taurus-MM's paper. In this test, we mirrored Taurus-MM's setup, including using nodes with 28 CPU cores and identical SysBench configurations. Figure 11 depicts the performance comparing with Taurus-MM. In this test, we configured shared data percentages at 50% for read-write and 30% for write-only workloads, reflecting the highest shared data scenarios in Taurus-MM 's evaluation. The y-axis of the figure represents throughput, while the numbers on each bar indicate scalability, defined as the relative throughput normalized to its own single-node performance. Our evaluation shows that PolarDB-MP shows comparable performance to Taurus-MM in a single-node setup. However, PolarDB-MP's superiority becomes evident in multi-node setups. For example, in read-write and write-only workloads, PolarDB-MP's throughput is respectively 3.17 times and 4.02 times that of Taurus-MM in an either-node cluster. Notably, Taurus-MM's scalability within an eight-node setup reaches only 1.88 in read-write and 1.5 in write-only workloads. Furthermore, increasing nodes from 4 to 8 in Taurus-MM results in just a 25% improvement in read-write workload and no improvement in write-only workload. In contrast, PolarDB-MP's scalability with eight-node is 5.64 in read-write workload and 4.62 in write-only workload, which is substantially higher than those of Taurus-MM. This significant enhancement in performance highlights PolarDB-MP's high scalability and efficiency, particularly in scenarios with increased data contention among nodes.

Taurus-MM's paper only presents Aurora-MM's relative performance with the configuration of 10% shared data. We then compare with Aurora-MM in the same configuration. Figure 12 shows the performance compared with Aurora-MM and Taurus-MM int light conflict scenarios(SysBench workloads with 10% shared). As Aurora-MM supports up to only 4 nodes, its 8-node results are omitted. Even with a low percentage of shared data, Aurora-MM shows no improvement from 2 to 4 nodes in read-write workloads, and in write-only workloads, 2 and 4 nodes perform worse than a single node. This is attributed to Taurus-MM 's use of optimistic concurrency control. In these light conflict scenarios, though Taurus-MM exhibits higher scalability than Aurora MM, it still lags behind PolarDB-MP, especially in the eight-node cluster.

[1]As of the end of 2022, Aurora MM is no longer available in the public cloud, and it is unable to be tested now.

5.4 Performance of secondary index updates

In this test, we compare PolarDB-MP's performance under global secondary index (GSI) updates with some shared-nothing-based databases, *e.g.*, TiDB, CockroachDB, and OceanBase, as shown in Figure 13. They are all based on the shared-nothing architecture, in which the GSI is also partitioned. In this case, when updating a GSI, it has to update more than one partition, one for the primary key update and another for the secondary key update. So the two-phase commit must be applied to update the GSI, and it will significantly slow down the performance. In this evaluation, we gradually increase the number of GSI in a table and measure the sustained throughput with a high random insertion pressure and the latency under single thread. As expected, PolarDB-MP significantly outperforms other systems. Especially with one GSI, PolarDB-MP's throughput only drops by 20% compared to the throughput without GSI, while other databases nearly drop by 60%-70%. When having 8 GSI, TiDB's, CockroachDB's and OceanBase's throughput are less than 20% of the throughput without GSI, but PolarDB-MP's performance is still acceptable. The latency also shows a similar trend to the throughput. PolarDB-MP's high-performance benefits from its fundamental design. PolarDB-MP does not rely on distributed transaction processing when conflict occurs on different nodes, however, it exchanges data pages via a high-speed RDMA network under a distributed locking scheme. This enables concurrent updates on different nodes while guaranteeing lower latency than conventional distributed transaction schemes.

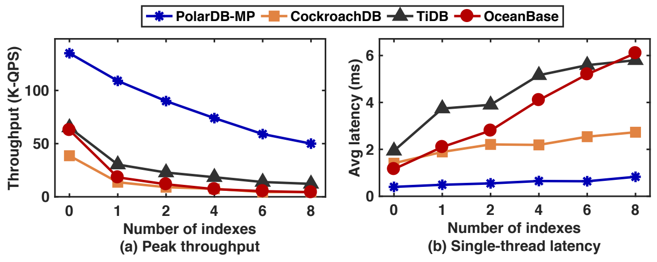


Figure 13: Performance of global secondary index updates.

5.5 Recovery

Finally, we focused on evaluating the recovery performance of PolarDB-MP. In this test, we set up a two-node cluster, with each node running SysBench read-write workloads. These nodes were configured to access different groups of tables. To simulate a crash scenario, we randomly kill node-1. The throughput timelines for both node-1 and node-2 during this event were captured and are illustrated in Figure 15. At the 15-second mark, node-1 was killed and then immediately restarted. A key observation following the crash was that node-2 continued to serve applications without any disruption, maintaining its original throughput. This uninterrupted service was due to the fact that node-1 and node-2 were not sharing data in this particular test setup. Consequently, node-2's operations and its transaction processing capabilities were independent of node-1, allowing it to continue functioning normally post the node-1 crash. Furthermore, it was observed that node-1 was able to resume operations swiftly, within just 10 seconds after the crash. This rapid recovery can be attributed to PolarDB-MP 's architecture, where node-1 could retrieve most of the necessary recovery data from the disaggregated shared memory, rather than relying solely on shared storage. This approach significantly reduces overhead and accelerates the recovery process. This test highlights PolarDB-MP 's resilience and efficient recovery capabilities, demonstrating its potential as a robust and reliable solution for environments where high availability and minimal downtime are crucial.

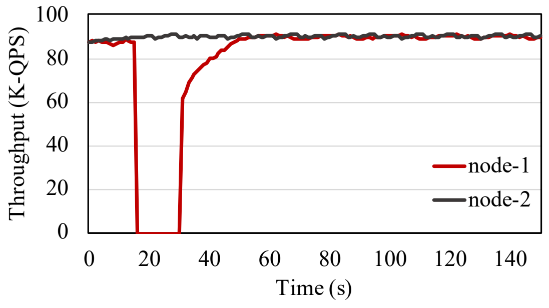


Figure 14: PolarDB-MP recovery.

6 RELATED WORKS

Single-primary cloud-native database. Many existing cloud-native databases are based on the primary-secondary architecture, such as AWS Aurora [49], Azure Hyperscale [14, 26], Azure Socrates [1], and Alibaba PolarDB [27]. They typically consist of a single primary node to process read/write requests and one or more secondary nodes to handle read requests. This design allows for scaling out read capacity by adding more secondary nodes. However, a significant limitation arises in write-heavy scenarios due to the single primary node for write operations. Some of these databases often claim the capability to automatically scale up the primary node in response to increased write loads. Yet, this scalability is inherently constrained by the resources available on the physical machine hosting the primary node. In scenarios where the machine lacks available resources, such as CPU or memory, scaling up becomes challenging, if not impossible.

Shared-nothing databases. The shared-nothing architecture is a prevalent scaling-out strategy, widely adopted by both key-value stores [25, 37] and relational databases (*e.g.*, CockroachDB [45], Spanner [11], PolarDB-X [6], TiDB [19] and OceanBase [55], etc). This architecture enables databases to distribute data across multiple nodes, where each node operates independently, managing its partition of the data. However, as discussed in Section 1 and Section 2, these shared-nothing-based relational databases encounter certain challenges. In contrast, PolarDB-MP employs shared storage at the storage layer and utilizes the shared memory for coordination, which offers a different approach to scalability and performance.

Shared-storage databases. Multi-primary databases based on shared-storage design are an alternative to the shared-nothing architecture, allowing each node equal access (read/write) to any record in the database. The traditional shared-storage- databases, like IBM DB2 Data Sharing [20] and Oracle RAC [9], suffer from expensive distributed lock management and high network overhead. Additionally, these traditional systems often lack the flexibility needed for dynamic cloud environments and generally incur a higher total cost of ownership (TCO) compared to modern cloud-native databases. Recent developments in cloud-native database technology, such as Aurora MM [3] and Taurus-MM [16], have introduced multi-primary capabilities to cloud environments. Both cloud databases rely heavily on log shipping and log replay as ways of page synchronization between nodes, introducing additional overhead and thus inefficient buffer cache coherence. In addition, Aurora MM, for example, adopts optimistic concurrency control for managing write conflicts. However, this method comes with its own set of trade-offs. While optimistic concurrency control can improve performance under low conflict conditions, it can lead to significant overhead when conflicts do occur. On the other hand, Taurus-MM utilizes pessimistic concurrency control. However, it has its drawbacks, particularly in the synchronization of pages between nodes as mentioned above. This synchronization often involves storage layer I/O and log application, which can be resource-intensive and impact overall system performance.

Distributed transaction optimization. Recently, there are also some research works to optimize distributed transaction processing for high performance. Tell [28], FaRM [18], FORD [59], DrTM[51] and DrTM-H [50] exploit RDMA network to improve performance in main-memory databases. These main-memory databases are not practical for many OLTP workloads that usually have TB/PB of data and require persistency. However, PolarDB-MP supports PB-level storage. SLOG [39], Calvin [48] all focus on the cross-region database cluster, where usually the network is the bottleneck.

7 CONCLUSION

This paper presents PolarDB-MP, a multi-primary cloud-native database that leverages a disaggregated shared memory framework. In PolarDB-MP, each node within the cluster has equal access to all data, enabling transactions to be processed on individual nodes without the need for distributed transactions. The core component of PolarDB-MP is the Polar Multi-Primary Fusion Server (PMFS), which integrates disaggregated shared memory. PMFS comprises essential functions: Transaction Fusion for transaction ordering and visibility, Buffer Fusion for a global shared buffer, Lock Fusion for concurrency control. These components are seamlessly integrated with modern RDMA network technology, enhancing performance. In our evaluations, PolarDB-MP shows a significant advantage over leading solutions like Taurus-MM.

ACKNOWLEDGMENTS

We are grateful to anonymous reviewers for their constructive feedback and valuable suggestions.

REFERENCES

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

[2] AWS. Amazon Aurora Limitless Database. https://aws.amazon.com/cn/blogs/aws/join-the-preview-amazon-aurora-limitless-database/, 2023.

[3] Eric Boutin and Steve Abraham. Amazon Aurora Multi-Master: Scaling Out Database Write Performance. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Amazon_Aurora_Multi-Master_Scaling_out_database_write_performance_DAT404-R1.pdf, 2019.

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

[5] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. SSD Bufferpool Extensions for Database Systems. *Proceedings of the VLDB Endowment*, 3:1435–1446, 2010.

[6] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, et al. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2859–2872. IEEE, 2022.

[7] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. PolarFS: an Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

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

[9] Sashikanth Chandrasekaran and Roger Bamford. Shared Cache-the Future of Parallel Databases. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, pages 840–840. IEEE Computer Society, 2003.

[10] Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying vMVCC, A High-performance Transaction Library Using Multi-version Concurrency Control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*, pages 871–886, Boston, MA, July 2023. USENIX Association.

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

[12] Transaction Processing Performance Council. On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/, 1992. "[accessed-April-2022]".

[13] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. Schism: A Workload-driven Approach to Database Replication and Partitioning. 2010.

[14] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 666–679, 2019.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

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

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

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

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

[20] Jeffrey W. Josten, C Mohan, Inderpal Narang, and James Z. Teng. DB2's Use of the Coupling Facility for Data Sharing. *IBM Systems Journal*, 36(2):327–351, 1997.

[21] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a High-performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[22] Alexey Kopytov. Sysbench: A System Performance Benchmark. *http://sysbench.sourceforge.net/*, 2004.

[23] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency Rationing in the Cloud: Pay Only When It Matters. *Proceedings of the VLDB Endowment*, 2(1):253–264, 2009.

[24] Vashudha Krishnaswamy, Divyakant Agrawal, John L. Bruno, and Amr El Abbadi. Relative Serializability: An Approach for Relaxing the Atomicity of Transactions. *journal of computer and system sciences*, 55(2):344–354, 1997.

[25] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[26] Willis Lang, Frank Bertsch, David J DeWitt, and Nigel Ellis. Microsoft Azure SQL Database Telemetry. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 189–194, 2015.

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

[28] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. On the Design and Scalability of Distributed Shared-data Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 663–676, 2015.

[29] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority Mechanisms for OLTP and Transactional Web Applications. In *Proceedings of the 20th International Conference on Data Engineering (ICDE 2004)*, pages 535–546, Boston, MA, USA, 2004.

[30] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[31] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference (USENIX ATC '15)*, pages 291–305, Santa Clara, CA, USA, 2015.

[32] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, pages 677–689, 2015.

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

[34] NVIDIA. NVIDIA CONNECTX-7 NDR 400G INFINIBAND ADAPTER CARD. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/infiniband-adapters/infiniband-connectx7-data-sheet.pdf, 2022. "[accessed-November-2023]".

[35] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 61–72, 2012.

[36] Andrew Pavlo, Evan PC Jones, and Stanley Zdonik. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *arXiv preprint arXiv:1110.6647*, 2011.

[37] Somasundaram Perianayagam, Akshat Vig, Doug Terry, Swami Sivasubramanian, James Christopher Sorenson III, Akhilesh Mritunjai, Joseph Idziorek, Niall Gallagher, Mostafa Elhemali, Nick Gordon, et al. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1037–1048, 2022.

[38] Abdul Quamar, K Ashwin Kumar, and Amol Deshpande. SWORD: Scalable Workload-aware Data Placement for Transactional Workloads. In *Proceedings of the 16th international conference on extending database technology*, pages 430–441, 2013.

[39] Kun Ren, Dennis Li, and Daniel J Abadi. SLOG: Serializable, Low-latency, Geo-replicated Transactions. *Proceedings of the VLDB Endowment*, 12(11), 2019.

[40] Kun Ren, Alexander Thomson, and Daniel J Abadi. Lightweight Locking for Main Memory Database Systems. *Proceedings of the VLDB Endowment*, 6(2):145–156, 2012.

[41] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. Persistent Memory Disaggregation for Cloud-Native Relational Databases. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 498–512, 2023.

[42] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*, pages 323–337, 2017.

[43] Michael Stonebraker. The Case for Shared Nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[44] Michael Stonebraker and Uĝur Çetintemel. "One Size Fits All" An Idea Whose Time Has Come and Gone. In *Making Databases Work: the Pragmatic Wisdom of*

Michael Stonebraker, pages 441–462. 2018.

[45] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. CockroachDB: The Resilient Geo-distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[46] Konstantin Taranov, S. D. Girolamo, and T. Hoefler. CoRM: Compactable Remote Memory over RDMA. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, pages 1811–1824, 2021.

[47] Alexander Thomson and Daniel J Abadi. The Case for Determinism in Database Systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.

[48] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[49] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design Considerations for High Throughput Cloud-native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.

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

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

[52] Tom White. *Hadoop: The Definitive Guide.* " O'Reilly Media, Inc.", 2012.

[53] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proceedings of the VLDB Endowment*, 10(7):781–792, 2017.

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

[55] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. OceanBase: a 707 Million tpmC Distributed Relational Database System. *Proceedings of the VLDB Endowment*, 15(12):3385–3397, 2022.

[56] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache Spark: a Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56–65, 2016.

[57] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The End of a Myth: Distributed Transactions Can Scale. *arXiv preprint arXiv:1607.00655*, 2016.

[58] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware Partitioning in Parallel Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 17–30, 2015.

[59] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 51–68, 2022.

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

[61] Tao Zhu, Zhuoyue Zhao, Feifei Li, Weining Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, Haining Li, and Huiqi Hu. Solar: Towards a Shared-Everything Database on Distributed Log-Structured Storage. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 795–807, 2018.

[62] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.