From Scale-Up to Scale-Out: PolarDB's Journey to Achieving 2 Billion tpmC

Xinjun Yang
Alibaba Cloud Computing

Feifei Li
Alibaba Cloud Computing

Yingqiang Zhang
Alibaba Cloud Computing

Hao Chen
Alibaba Cloud Computing

Qingda Hu
Alibaba Cloud Computing

Panfeng Zhou
Alibaba Cloud Computing

Qiang Zhang
Alibaba Cloud Computing

Shuai Li
Alibaba Cloud Computing

Zongzhi Chen
Alibaba Cloud Computing

Zheyu Miao
Alibaba Cloud Computing

Rongbiao Xie
Alibaba Cloud Computing

Chuan Sun
Alibaba Cloud Computing

Zetao Wei
Alibaba Cloud Computing

Jing Fang
Alibaba Cloud Computing

Xingxuan Zhou
Alibaba Cloud Computing

Xiaofei Wu
Alibaba Cloud Computing

ABSTRACT

In the past decade, cloud databases have experienced rapid development and growth. PolarDB, Alibaba's cloud-native OLTP database, has evolved significantly to meet the increasing demand for cloud-native architectures and now serves hundreds of thousands of customers across various industries.

This paper presents PolarDB's evolution over the past eight years, with a focus on scalability, performance, and cost-efficiency. Initially, PolarDB adopted a primary-replica architecture based on disaggregated storage, with an emphasis on enhancing single-node performance for scale-up in modern many-core systems. To achieve this, we co-designed PolarDB with cutting-edge hardware, including RDMA, to improve performance. Meanwhile, we refined the internal architecture, including improvements to B+ tree concurrency control and transaction management, ensuring high scalability in scale-up scenarios. More recently, our focus has shifted to scaling out PolarDB to meet the performance and scalability needs of ultra-large-scale applications. By leveraging RDMA, we optimized distributed transaction processing, *transforming PolarDB into a high-performance, high-scalability and cost-effective distributed database*. In the TPC-C benchmark, PolarDB scaled out to 2340 nodes and achieved over 2 billion tpmC, with a jitter rate of no more than 0.16% during the 8-hour stress test. Compared to the second- and third-highest-performing databases in public TPC-C results, PolarDB's tpmC is 2.52× and 2.91× higher, respectively. In terms of cost-effectiveness, PolarDB's per-tpmC cost is 37% and 79.5% lower than that of the other two systems, respectively.

PVLDB Reference Format:
Xinjun Yang, Feifei Li, Yingqiang Zhang, Hao Chen, Qingda Hu, Panfeng Zhou, Qiang Zhang, Shuai Li, Zongzhi Chen, Zheyu Miao, Rongbiao Xie, Chuan Sun, Zetao Wei, Jing Fang, Xingxuan Zhou, and Xiaofei Wu. From Scale-Up to Scale-Out: PolarDB's Journey to Achieving 2 Billion tpmC. PVLDB, 18(12): 5059-5072, 2025.

Feifei Li is the corresponding author.

doi:10.14778/3750601.3750627

1 INTRODUCTION

Over the past decade, applications have increasingly migrated to the cloud for better elasticity and lower costs. In response, cloud databases have experienced explosive growth to accommodate the growing demands of these applications. PolarDB, one of the leading cloud databases, has evolved substantially since its launch, adapting to meet diverse requirements across various industries.

PolarDB adopts a disaggregated storage architecture, with the distributed file system PolarFS [7] designed specifically for cloud databases. PolarDB has been highly optimized to work with this distributed file system and leverages RDMA for fast storage I/O to enhance performance. Initially, PolarDB employed a primary-replica architecture, which is common in database systems. Given the high core count and performance of modern servers, a single primary node was sufficient for many applications. Consequently, during this phase, the primary focus was on improving scalability for scaling up on modern multi-core servers. However, we found that existing databases, such as MySQL, struggled to scale effectively with the increasing core count. For example, MySQL's throughput stagnated once exceeding 64 CPU cores in Sysbench read-write workloads. To address these issues, PolarDB introduced several novel designs aimed at improving scalability in many-core environments. Specifically, we developed PolarIndex, a new concurrency control scheme for the B+ tree that alleviates bottlenecks in write-intensive workloads. Additionally, PolarTrans was proposed to optimize transaction management in high-concurrency scenarios, significantly improving performance. Meanwhile, PolarTrans

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750627

also plays an important role in distributed transaction processing in PolarDB, improving the PolarDB's 2PC protocol.

Recently, our focus has shifted towards scaling out PolarDB to support ultra-large-scale applications, such as e-commerce, banking, and cryptocurrency trading. Our previously proposed multi-primary architecture, PolarDB-MP [63], is optimized for scenarios with high data access contention across nodes and relies on disaggregated shared memory for deployment. While it delivers superior performance in high-contention workloads compared to distributed transaction solutions within moderately sized clusters, its scalability is fundamentally constrained by the physical limitations of disaggregated memory, making it difficult to scale beyond a few hundred nodes. Similar limitations are observed in other shared-storage-based multi-primary databases [9, 17, 28, 37]. For well-partitioned workloads like TPC-C, where cross-partition contention is minimal, PolarDB-MP offers no clear advantage and faces deployment challenges at ultra-large scales. In such scenarios, distributed databases based on data sharding [12, 14, 25, 56] provide a more scalable alternative, supporting clusters with thousands of nodes.

However, these distributed database systems typically rely on TCP/IP-based communication for cross-node messaging, which significantly burdens the CPU, as it must process network messages. As a result, transaction throughput is often limited by network bandwidth [20, 66]. To address this, PolarDB introduces a distributed version co-designed with RDMA to eliminate the communication bottleneck in distributed transactions. At its core is Polar2PC, a novel two-phase commit (2PC) protocol optimized for RDMA.

Unlike the standard 2PC protocol [22, 54], where the coordinator node (CN) sends a prepare message to each participant node (PN) during the prepare phase, Polar2PC utilizes a one-sided RDMA interface to directly check the execution state of the PNs, determining if the transaction is ready to commit. In the commit phase, to integrate with the PolarTrans design, the in-memory transaction states are updated via the one-sided RDMA interface. This allows for a fast acknowledgment to the application, with the release of PN resources occurring in the background while still maintaining the same ACID guarantees. Compared to the standard 2PC protocol, Polar2PC significantly reduces network overhead, leading to improved performance. Meanwhile, it maintains compatibility with the X/Open XA standard [55], ensuring seamless integration into existing ecosystems.

Finally, we evaluate PolarDB with the TPC-C [15] benchmark. Despite being proposed decades ago, TPC-C remains a valuable benchmark today and continues to be widely used for evaluating modern databases [12, 25, 30, 58, 65]. In our evaluation, we scale out the PolarDB cluster to 2340 nodes, achieving over 2 billion tpmC with a jitter rate of no more than 0.16% during the 8-hour stress test. When compared to the second- and third-highest-performing systems in the publicly available TPC-C results [2], PolarDB's tpmC is 2.52× and 2.91× that of the other two systems, respectively. Additionally, PolarDB demonstrates superior cost-efficiency, with a per-tpmC cost 37% and 79.5% lower than that of the other two systems, respectively. These results demonstrate PolarDB's exceptional performance, scalability, and cost-effectiveness.

We summarize our main contributions as follows:

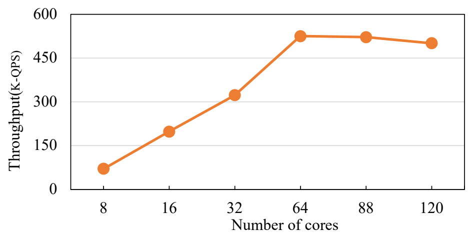

Figure 1: The scalability of MySQL

- We present the evolution of PolarDB from scale-up to scale-out and introduce the design and architecture of PolarDB.
- We detail the implementation of PolarDB's scaling-up optimizations and its distributed transaction design for scaling-out.
- We evaluate PolarDB using the official TPC-C benchmark, achieving 2 billion tpmC with a low cost per tpmC, achieving the highest performance in the public TPC-C results.

This paper is structured as follows. First, we present the background and motivation in Section 2. Then we provide PolarDB's overview and detailed design in Section 3 and Section 4. Next, we evaluate PolarDB in Section 5 and review the related works in Section 6. Finally, we discuss PolarDB's evolution in Section 7 and conclude the paper in Section 8.

2 BACKGROUND AND MOTIVATION

2.1 Database scale-up

Since the launch of AWS Aurora [58] in 2014, a growing number of cloud databases have emerged, such as Azure SQL Database [33], TaurusDB [1] and PolarDB [36]. Many of these adopt a primary-replica architecture. In this architecture, a primary node handles both read and write requests, while one or more replica nodes handle only read requests. Since only the primary node processes write requests, it must be scaled up to improve performance in write-intensive workloads. However, most existing cloud databases have focused more on the evolution of the architecture from on-premises to cloud deployment, with relatively little attention given to enhancing single-node scalability on modern many-core systems.

As MySQL serves as the foundation for many cloud databases, we use it as a representative example to investigate scale-up limitations. Figure 1 illustrates MySQL's performance as we increase the allocated CPU resources, demonstrating its ability to scale up. We observe that when the number of CPU cores is below 64, MySQL's throughput scales linearly with CPU resources. However, beyond 64 CPU cores, throughput saturates, failing to scale further despite additional computing resources. This suggests that MySQL's internal design becomes a bottleneck, preventing efficient utilization of modern many-core servers. Given that current servers often feature hundreds of CPU cores, MySQL's limited scalability restricts its ability to fully exploit available hardware resources, making scale-up improvements essential for cloud databases. In our investigation, we found two main bottlenecks limiting scalability in the single-node setup. One is the B+ tree concurrency limitation, and the other is the internal transaction management component.

B+ tree concurrency. The B+ tree is a widely used data structure for indexing in many databases [10, 23, 27, 41, 47, 62], including

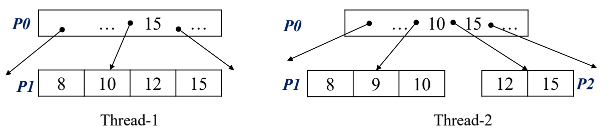

Figure 2: An example of concurrent access to a B+ tree

MySQL and its variants. To support concurrent access, most implementations employ latch [1] mechanisms at the B+ tree page level. During an insertion, a page may overflow, requiring it to be split into two pages, with the parent page also updated to add a new record. This structural modification operation (SMO) introduces challenges for concurrency control, especially when executed alongside other B+ tree operations. For example, as shown in Figure 2, when Thread-1 (T1) searches for value 12 in the B+ tree, page P0 directs the search to page P1. Before T1 latches P1, Thread-2 (T2) may insert value 9 into P1, triggering an SMO that moves value 12 to a newly created page P2. In this scenario, T1 fails to find value 12 on P1, leading to inconsistencies in the search process. To address this issue, a common approach is to hold the parent page latch until the child page is successfully latched. Additionally, during an SMO, all pages involved in the operation must be latched before the modification begins, preventing search threads from observing an inconsistent B+ tree. However, this introduces a new challenge: while search operations acquire latches from higher levels to lower levels, SMOs originate at lower levels and acquire latched upward, potentially leading to deadlocks. To eliminate this risk, a mainstream solution is to prevent concurrent SMOs and enforce a top-down latching order during SMO execution. In MySQL's implementation, during an SMO, an shared-exclusive (SX) latch is applied to the entire B+ tree, allowing only shared latches but blocking all modifications. This ensures that no other SMOs can proceed and that the ongoing SMO can acquire latches on all affected pages in a top-down order. This solution prevents concurrent SMOs, even if they operate on unrelated subtrees, consequently, significantly limits database throughput in high-concurrency update workloads. The situation worsens when dealing with large tables, where data pages often cannot fit into the buffer pool, forcing the SMO process to perform multiple I/O operations to fetch the relevant pages. This further prolongs SMO execution time, reducing system throughput. The issue becomes even more severe in cloud storage environments, where high I/O latency amplifies the performance impact.

To address the above issue, and inspired by the Blink Tree design [34], PolarDB introduces PolarIndex, a new B+ tree concurrency control mechanism that allows concurrent SMO execution. This design significantly improves concurrency and alleviates latch contention, allowing more efficient index modifications under high-concurrency workloads.

Transaction management. Multi-Version Concurrency Control (MVCC) is a widely used solution for managing concurrent data access while ensuring transaction isolation and consistency. A common requirement in MVCC-based databases is that each transaction needs a consistent snapshot of the database to maintain isolation.

[1]In this paper, we use the term *latch* to refer to short-duration synchronization mechanisms for physical consistency (e.g., protecting in-memory data structures), and *lock* to refer to long-duration synchronization mechanisms for ensuring transactional correctness (e.g., enforcing isolation levels).

To achieve this, MySQL-like databases typically maintain an active transaction list, implemented as an array. When a transaction begins, its transaction ID is added to the list, and it is removed when the transaction commits. In this design, if an update is made by a transaction in the active transaction list, this update is not visible to any other transactions except the one that performed the update. This design introduces the read view concept. When a transaction or query starts, a read view is generated by copying the active transaction list along with two variables: the current maximum transaction ID (trx_id_{max}) and the minimum active transaction ID (trx_id_{min}) at the time the transaction/query starts. When a transaction reads a record, it first retrieves the transaction ID that modified the record. If this transaction ID is greater than trx_id_{max} in the read view, the record is not visible to the transaction, as it was modified by a transaction that began after the concurrent transaction. On the other hand, if the transaction ID is less than trx_id_{min}, the record is visible, as its update was committed before the concurrent transaction began. In cases where the transaction ID falls between trx_id_{min} and trx_id_{max}, it checks the read view's active transaction list to determine whether the corresponding transaction is still active. While the read view plays a crucial role in transaction management, it introduces a performance bottleneck. Specifically, each transaction/query generates a read view by copying the active transaction list from a global variable to a local copy. The global active transaction list is protected by a global lock, which can cause significant lock contention in high-concurrency scenarios. Given that modern systems typically have many cores, this bottleneck can severely limit the database's scalability.

To overcome this issue, PolarDB proposes a new transaction management scheme called PolarTrans, which eliminates the need for the active transaction list. PolarTrans is implemented with a variety of lock-free designs, allowing the read view to be generated without locking. This significantly improves database performance in many-core systems, enabling better scaling capabilities.

2.2 Database scale-out

Multi-primary databases. The multi-primary database is a solution designed for scaling out to meet the demands of applications that require high scalability and large-scale clusters. The shared-storage-based multi-primary databases [5, 9, 17, 28, 37, 63] usually struggle primarily with managing conflicts between nodes and coordinating transactions across them. They typically require extra resources (such as the disaggregated shared memory in PolarDB-MP) to handle the conflicts and manage global transactions.

Alternatively, data-sharding-based distributed databases provide another approach to database scaling. Notable examples include Spanner [14], CockroachDB [56], TiDB [25], OceanBase [65] and TDSQL [12]. In these systems, data is partitioned across multiple nodes, with each node having exclusive write/read access to its respective data partition. This architecture offers excellent scalability when application traffic is well-partitioned. However, if a transaction spans multiple partitions, distributed transaction processing becomes necessary to ensure the ACID properties of transactions.

2PC. The Two-Phase Commit (2PC) protocol is a widely adopted solution for distributed transactions, and nearly all distributed databases rely on it or its variants for transaction processing. The protocol consists of two phases: *prepare* and *commit*. In the prepare phase, the coordinator node sends a "prepare" request to all participant nodes. Each participant then checks whether it can commit the transaction, responding with either a "yes" (commit) or "no" (abort). In the commit phase, the coordinator sends either a "commit" or "abort" command based on the responses received, and all nodes finalize the transaction. In most implementations, the coordinator communicates with participants over a TCP/IP-based network. This approach can put significant strain on the CPU, which must handle network messages, often limiting transaction throughput due to network bandwidth constraints [20, 66].

The network bottleneck in 2PC motivates integrating RDMA for message passing in PolarDB's distributed version. We integrated RDMA with the 2PC protocol and proposed the Polar2PC protocol, which alleviates network bottlenecks and significantly enhances distributed transaction performance. Several research efforts, such as Tell [38], FaRM [19], FORD [68], DrTM[61] and DrTM-H [60] have explored optimizing distributed transaction processing using RDMA, particularly in main-memory databases. However, these main-memory databases are often impractical for many OLTP workloads, and while these proposals are promising, they remain in the research phase and require further development to ensure production-level stability.

2.3 Opportunities with RDMA

Recent advancements in network technology, including ultra-low latency (typically just a few microseconds) and 400Gb/s throughput from devices like the ConnectX-7 InfiniBand adapter [46], along with the growing presence of RDMA in standard commodity clusters [18, 72], are reducing the impact of network bottlenecks. Numerous studies have leveraged RDMA to enhance database systems [48], including accelerating data synchronization [73], reducing locking overhead [4, 21], optimizing transaction processing [13, 19, 40, 53, 60, 66, 68], and improving index structures [39, 59, 74]. At Alibaba, RDMA is an integral part of the infrastructure, and PolarDB is specifically designed to leverage it [63, 64, 70]. Every PolarDB cluster in the Alibaba public cloud is equipped with RDMA support, offering a significant advantage. This integration enables PolarDB's distributed version to be deployed without the need for additional hardware costs. In PolarDB, we leverage the RDMA to improve the distributed transaction processing. While many prior systems leverage RDMA to overhaul the entire distributed transaction processing workflow [13, 19, 40, 53, 60, 66, 68], our design adopts a more lightweight approach: RDMA is used exclusively for data synchronization during the distributed transaction processing. This selective integration preserves full compatibility with the conventional two-phase commit protocol and requires minimal changes to the existing codebase. Despite its simplicity, this optimization yields significant performance improvements.

3 POLARDB OVERVIEW

Figure 3 provides an overview of PolarDB. A PolarDB cluster consists of multiple database nodes, each capable of handling application connections and providing equal read/write access. All nodes are interconnected through a high-speed RDMA network for fast

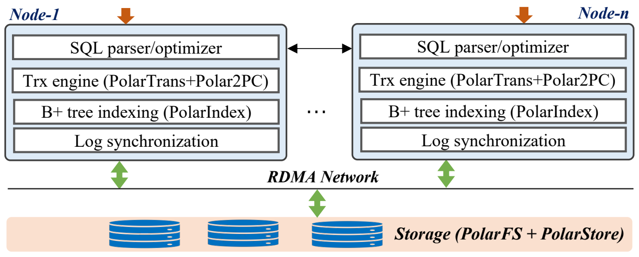
Node-1
SQL parser/optimizer
Trx engine (PolarTrans+Polar2PC)
B+ tree indexing (PolarIndex)
Log synchronization
Node-n
SQL parser/optimizer
Trx engine (PolarTrans+Polar2PC)
B+ tree indexing (PolarIndex)
Log synchronization
RDMA Network
Storage (PolarFS + PolarStore)

Figure 3: The overview of PolarDB

communication. PolarDB employs a disaggregated storage architecture, where each database node is connected to the disaggregated storage layer. The storage system is implemented using PolarFS and PolarStore [7], which ensures low-latency, high-throughput, and high-availability storage. Meanwhile, PolarDB maintains compatibility with standard file system interfaces.

Each database node consists of an SQL parser/optimizer, transaction engine and B+ tree indexing and log synchronization. PolarDB has significantly improved these components to enhance scalability on many-core systems. In particular, PolarTrans was introduced to optimize transaction management and alleviate bottlenecks under intensive workloads. In conjunction with PolarTrans, we designed Polar2PC, an optimized two-phase commit (2PC) protocol, to enhance distributed transaction processing. For efficient indexing under high concurrency, PolarIndex was developed to eliminate bottlenecks in B+ tree structure modifications (SMOs). Additionally, log synchronization ensures that the primary node's logs are replicated to standby nodes for high availability.

To implement distributed transaction processing, PolarDB partitions user data and assigns different partitions to different nodes. Each node can read and write only the data belonging to its own partitions. However, to provide a transparent interface for applications, any node can receive queries, regardless of data partitioning. When a query references data owned by other nodes, the receiving node forwards the query to the appropriate node. This necessitates a distributed transaction protocol. To address this, PolarDB introduces Polar2PC, which co-designs the conventional 2PC protocol with RDMA to achieve high-performance distributed transaction processing, providing high-scalability.

4 IMPLEMENTATION AND OPTIMIZATION

4.1 PolarIndex

PolarDB also adopts the B+ tree for indexing. Although the B+ tree has been extensively optimized over decades, mainstream implementations (such as in MySQL) still suffer from scalability bottlenecks under write-intensive workloads, particularly on modern many-core systems (see Section 2.1). Inspired by the B-link tree [34], PolarDB introduces PolarIndex, a new concurrency control scheme for the B+ tree, alleviating the SMO bottleneck and significantly improving performance in write-intensive workloads. While inspired by the B-link tree, PolarIndex makes several key adaptations for compatibility with the existing PolarDB codebase. It enables read latches to support shared buffer pool access (required in many databases) and avoids modifying page formats (for the consideration

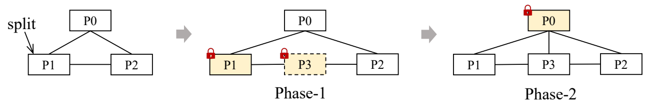

Figure 4: SMO in PolarIndex

of production compatibility) by using an optimistic right-traversal strategy instead of high keys. It also incorporates other engineering optimizations to better support production deployment. The core idea of PolarIndex is to avoid latching the entire subtree during an SMO and instead latch each page only when necessary. Additionally, it eliminates the global tree latch, enabling concurrent SMOs. These improvements significantly enhance B+ tree concurrency in SMO-intensive workloads, leading to better overall performance. However, implementing this design presents challenges. It requires careful handling to prevent potential deadlocks and inconsistent reads while ensuring compatibility with mainstream implementations for product stability and adoption.

Workflow. The search operation in a B+ tree follows a top-down traversal, acquiring latches as it descends through the tree. In contrast, SMOs are triggered at lower levels and propagate upwards, creating an inverse latching order that can lead to potential deadlocks and increased latch contention. To address this, PolarIndex eliminates the traditional latch coupling scheme, which involves holding a latch at one level while acquiring a latch at another level. Instead, PolarIndex acquires latches only at the level where concurrent access occurs. Figure 13 provides an example to demonstrate how PolarIndex's concurrency control scheme works. In this example, an incoming insertion leads to the splitting of page *P1*. The parent of *P1* is *P0*, and its sibling is *P2*. Instead of holding exclusive (X) latches on all pages in the entire sub-tree during the entire SMO operation, PolarIndex divides the SMO processing into two phases, latching only a subset of pages in each phase to improve concurrency. During the first phase, latches are acquired only on *P1* and the newly generated page *P3*. Once *P3* is ready and the corresponding data has been migrated to it, *P1* updates its sibling pointer to *P3*, completing the first phase. After this phase, the latches on *P1* and *P3* are released. In the second phase, *P0* is exclusively latched, and a pointer is added to *P0* to reference the new page *P3*. During this phase, only *P0* is latched, while its child pages remain unlatched. Once the second phase is completed, the SMO operation finishes, and the B+ tree structure is intact. It is important to note that between these two phases, no page is latched, and the B+ tree is in an incomplete state. However, the tree is still able to serve searches, albeit with some additional effort to ensure correctness. Between these two phases, *P3* has not yet been attached to its parent page *P0*, so *P0* is not aware of *P3*. If a query attempts to find a record on *P3*, the search will process *P0* and assume the record is located on *P1*, returning *P1*. Therefore, during the SMO operation, if a search reaches *P1*, it may need to traverse to the next page if the target record is not found in *P1*.

Advantages. The PolarIndex eliminates the need for a global shared-exclusive (SX) latch on the entire tree during structural modification operations (SMOs). This enables concurrent SMOs in write-heavy workloads, significantly improving performance. Furthermore, during an SMO, PolarIndex acquires the necessary latches only at each phase, allowing intermediate states of the B-tree to be visible to other threads while still ensuring data consistency. This further enhances the concurrency of B+ tree access.

4.2 PolarTrans

In PolarDB, we designed a novel transaction management scheme, PolarTrans, that alleviates bottlenecks in the existing transaction management subsystem of mainstream databases, such as MySQL and its variants. The primary goal of PolarTrans is to efficiently maintain transaction information and improve high performance.

Transaction information table. The core data structure of PolarTrans is the *transaction information table* (TIT), which stores transaction details, as shown in Figure 5. The TIT is organized as an array, with each entry consisting of a transaction pointer and the transaction's commit timestamp (CTS). The transaction pointer refers to an active transaction object during the runtime, while the CTS represents the commit timestamp of the transaction if it is committed. When a transaction begins, it is assigned an incremental transaction ID. Based on this transaction ID, a corresponding TIT slot is allocated to the transaction. Typically, the allocation of TIT slots is performed by taking the modulus of the slot count with the transaction ID. Upon committing, the transaction stores its CTS in the allocated TIT slot. Most operations on the TIT are lock-free, ensuring high performance even under heavy workloads.

Transaction ordering. PolarTrans adopts the Lamport timestamp for transaction ordering. It maintains a global variable, CTS_max, as the logical timestamp, which is incremented to allocate a transaction's CTS. To provide a snapshot of all committed transactions, it also maintains another global variable, $CTS\_max\_committed$, recording the maximum CTS of all transactions. When committing a transaction, it first allocates a CTS from CTS_max by incrementing it, then writes the CTS to the transaction's TIT slot, and finally updates $CTS\_max\_committed$, if necessary, after the transaction is committed. To ensure that a committed transaction's TIT slot is properly filled, the steps of CTS allocation and writing must be lock-protected to guarantee atomic execution. Consequently, when allocating a read view from $CTS\_max\_committed$ to a transaction, any transaction whose CTS is less than $CTS\_max\_committed$ is considered visible because its TIT slot must be filled. This design enables efficient visibility checks by simply comparing $CTS\_max\_committed$ with the CTS stored in the TIT slot.

Data visibility. In PolarTrans, when reading a data record, it begins by retrieving the transaction ID of the transaction that last updated the record. Using this transaction ID, it queries the TIT to obtain the transaction's CTS. If the TIT slot associated with the transaction has been reassigned to a new transaction, it indicates that the previous transaction has been committed and its updates are now visible to all transactions. Conversely, if the TIT slot remains unassigned, PolarTrans can directly retrieve the CTS from that slot. A CTS value still set to its initial value signifies that the transaction is still active, meaning its updates have not yet become visible to other transactions. Finally, if the CTS is less than or equal to the read view's timestamp, it ensures that the current record is visible to the current transaction.

slot	0	1	2	3	4	5	6	7
trx pointer	nullptr	0x1d00	0x1350	0x1828	nullptr	nullptr	nullptr	nullptr
CSN	0	99	INIT	0	0	0	0	0

Figure 5: Design of the Transaction Information Table (TIT) in PolarTrans

TIT recycle. To optimize memory usage, the TIT size is limited, and PolarTrans employs a background thread to periodically recycle TIT slots. If a transaction's updates are already visible to all transactions, there is no need to retain its CTS for visibility checks. The recycling process first determines the minimum timestamp among all active read views, then recycles TIT slots where the CTS is earlier than this minimum timestamp. During recycling, the transaction pointer and CTS field of the TIT slot are reset, allowing future transactions to reuse the slot. When a visibility check encounters a freed TIT slot or one that has been reassigned to a new transaction, the previous transaction that occupied this slot is considered committed, and its updates are visible to all transactions.

TIT slot offloading. A transaction's TIT slot is allocated based on the modulus of the slot count with the transaction ID. If a TIT slot is not recycled, the next transaction mapped to this slot cannot use it and has to wait. If a long-running transaction causes the minimal read view timestamp to remain unchanged, many transactions may be forced to wait for the TIT slot to become available, significantly reducing performance. To address this issue, we offload TIT slots that block new transactions to a hash map. The hash map records the IDs and CTS of these transactions. The background TIT recycle thread also removes records from the hash map based on the smallest timestamp of all active read views. When checking data visibility, the hash map is also consulted to get a transaction's CTS. In most cases, TIT slots can be successfully recycled, and long-running transactions are infrequent. As a result, the hash map is typically empty and does not need to be checked often, ensuring that its impact on performance is minimal.

Benefits. In PolarTrans, only the corresponding TIT slot is updated during transaction begin and commit, and different slots can be modified independently. This design avoids the contention on the global active transaction list found in conventional approaches (as discussed in Section 2.1). In particular, when generating a read view, PolarTrans bypasses the costly copy of the global list by directly checking data visibility via TIT slot access. As a result, PolarTrans achieves high concurrency and performance, as shown in Figure 14.

4.3 RDMA-enabled distributed transaction

The two-phase commit (2PC) protocol is widely used in most commercial databases to implement distributed transactions [6, 12, 25, 56, 65]. PolarDB also employs the 2PC protocol, but is co-designed with RDMA to improvement performance. This section will present our new 2PC protocol, Polar2PC, which is used in PolarDB.

Overview. As discussed earlier, many existing implementations of the Two-Phase Commit (2PC) protocol rely on TCP/IP-based networks, which often face significant network bottlenecks. To overcome this limitation, we integrate RDMA into the Polar2PC protocol to improve performance. Additionally, Polar2PC leverages

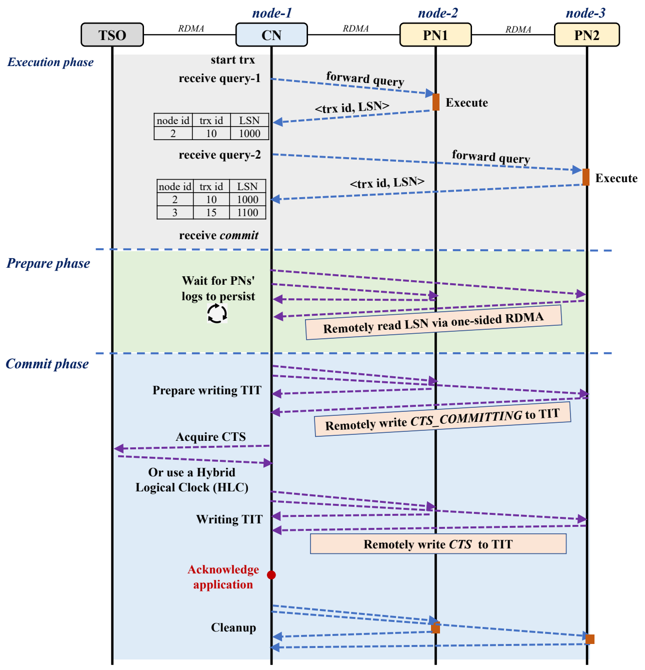

Figure 6: The design of Polar2PC

the capabilities of PolarTrans to make the commit phase asynchronous, while maintaining the same ACID guarantees.

Like most databases, PolarDB employs ARIES-style logging [43], where each update generates corresponding redo logs and follows a steal/no-force policy. Under this model, if a transaction's redo logs are fully persisted to storage, its updates can be considered persistent. During the prepare phase of Polar2PC, RDMA is used to check whether all redo logs of the transaction have been persisted, determining if the transaction is ready to be committed. This operation only requires one-sided RDMA operations on the CN, without involving the PNs, thus eliminating the need for message transfers between the CN and PNs. In the commit phase, the CN obtains a commit timestamp (CTS) and remotely writes it to the PN's TIT slot (explained in Section 4.2) via RDMA. As detailed in Section 4.2, a transaction is considered committed when the CTS is filled in the TIT slot. Once the CTS is filled on all PNs, the CN can acknowledge the application that the transaction has been successfully committed. Afterward, the CN sends a commit message to the PNs to release the corresponding resources, with this step occurring in the background, not in the critical path of the distributed transaction processing. The commit phase also requires only one-sided RDMA operations before sending the acknowledgment, which significantly enhances performance.

Workflow. In the PolarDB cluster, all nodes are primary and can equally serve applications. The terms *CN* and *PN* are logical roles in PolarDB. When a transaction is initiated on a node, that node acts as the CN for the transaction, while any other nodes involved in the transaction are considered PNs for that specific transaction. Figure 6 illustrates the workflow of a distributed transaction in PolarDB. The cluster has three database nodes and one timestamp oracle (TSO) server. Suppose an application connects to node-1. When this connection starts a transaction on node-1, node-1 assumes the CN role for the transaction. Given that this transaction has two queries that access data stored on node-2 and node-3, node-1 (the CN) forwards these queries to node-2 and node-3, where they are executed. In this case, node-2 and node-3 act as PNs for this transaction. For simplicity, we refer to node-1, node-2, and node-3 as CN, PN-1, and PN-2, respectively, in this example.

In the ***execution phase***, the CN receives queries from applications and forwards them to the PNs for execution. When a PN receives a query from the CN, it starts a local transaction (if no transaction is already in progress) and executes the query locally. After execution, the PN responds to the CN with its local transaction ID and the current log sequence number (LSN) generated by the query. Upon receiving the response, the CN records this information within the current transaction, maintaining the PN's node ID, local transaction ID, and corresponding LSN. When the CN receives a commit request from the application, it proceeds with committing the transaction by following the Polar2PC protocol.

In the ***prepare phase***, the CN checks its locally stored information about the transaction, retrieves all involved PNs, and their corresponding LSNs. The CN waits until the redo logs for the transaction are persistent on all PNs. To achieve this, each node maintains a variable (LSN_{flush}) that indicates the current maximum LSN that has been persisted on the node. A background thread flushes the redo logs to storage in the order of LSNs and updates the LSN_{flush} value accordingly. The CN can remotely read the LSN_{flush} values of all PNs using a one-sided RDMA interface and wait until all PN's LSN_{flush} values exceed the LSN recorded by the CN for the current transaction. Once this condition is met, all updates on all PNs are persistent, indicating that the transaction is ready to be committed.

Although Polar2PC removes the traditional prepare round to reduce latency, it implicitly assumes the PN is always ready to commit unless explicitly aborted by the CN. While this sacrifices the ability of the PN to unilaterally abort, we found in practice that such aborts are rare. Therefore, this trade-off is justified by the significant performance benefits it brings.

In the ***commit phase***, the CN first remotely writes a flag (a predefined special value, *e.g.*, CSN_COMMITTING) to all PNs' TIT slots via RDMA to indicate that the transaction is committing. It then acquires a commit timestamp (CTS) from the TSO and remotely writes the CTS to all PNs' TIT slots. Once the CTS is filled on all PNs, all changes made by this transaction are visible to other transactions. Following this, the CN can immediately notify the application that the transaction has been successfully committed. After acknowledging the application, the CN sends a commit message to the PNs, prompting them to release transaction-related resources. Once the PNs acknowledge receipt of the commit message, the CN releases its corresponding resources. Since the application acknowledgment does not need to wait for the PNs' commit (as the commit is asynchronous), this design improves performance.

Commit timestamp. The commit timestamp (CTS) is a fundamental concept in distributed systems for ensuring consistency and isolation. PolarDB supports two widely used approaches: timestamp oracle (TSO) and hybrid logical clock (HLC) [31]. PolarDB's TSO is implemented with RDMA, allowing database nodes to acquire a CTS via one-sided RDMA in just a few microseconds. This design guarantees strong consistency, but as the number of nodes increases, the latency of CTS acquisition may also grow. To address this, PolarDB offers HLC as an alternative for high performance in large-scale clusters, providing lower latency at the cost of relaxed consistency guarantees, as sequential consistency is not guaranteed in HLC. However, different from the standard HLC that relies on the TCP/IP network for timestamp synchronization across nodes, PolarDB employs RDMA for cross-node communication, significantly improving performance.

Logging. During the execution phase, the CN adopts the ARIES-style logging [43] to record both undo and redo logs for crash recovery. In addition, we introduce extra log entries to ensure durability and consistency for distributed transaction processing. After the execution phase, when the CN receives the COMMIT command from the client, it first appends the distributed transaction metadata, such as the distributed transaction ID and the identifiers of all PNs, to its redo log buffer. Once the CN successfully obtains the global commit timestamp, it appends a prepare OK log entry and forces a flush, ensuring that all buffered log records are persisted to disk. Upon receiving the commit command from the CN, each PN commits its corresponding local transaction and appends a local commit log entry to its own redo log to indicate successful local commit. This logging protocol guarantees the durability and consistency of distributed transactions across all participating nodes.

Integration with PolarTrans. PolarTrans also plays a key role in distributed transaction processing. In PolarDB, each node's TIT is accessible to all other nodes via RDMA. When a node acts as the CN, it can directly write the CTS to the PN's TIT via RDMA. Since the TIT slot is allocated based on the modulus of the slot count with the transaction ID, and because the CN knows the local transaction ID on each PN, it can easily calculate the transaction's TIT slot. However, there is a difference in the process of filling the CTS in TIT between single-node and distributed setups. In a distributed setup, where more than one PN may be involved in a transaction, the CN cannot write the CTS to all PNs' TIT slots at the exact same time. This introduces a potential issue: if another transaction accesses data across different PNs, it may observe inconsistent visibility—some updates may already be visible (where the TIT slots are filled with the CTS), while others may remain invisible (where the TIT slots are still empty), violating ACID properties. To address this issue, we implement a two-step CTS filling process during the commit phase. First, the CN writes a predefined flag, *CSN_COMMITTING*, to the TIT slots on all PNs. Once this step is successfully completed, the CN acquires the CTS from TSO and finally writes it to all PNs' TIT slots. If a transaction encounters the *CSN_COMMITTING* flag during the visibility check, it must retry until the TIT slot is fully populated with the final CTS. This ensures that a transaction's updates are either fully visible or fully invisible across all nodes, maintaining consistency in the distributed system.

4.4 High availability and crash recovery

High availability. To ensure high availability, PolarDB deploys a standby node for each primary node. When an update occurs on the primary node, it generates redo logs, which are then transferred to the standby node, allowing it to apply the logs and keep its data up to date. The standby node can be deployed within the same availability zone or across regions, providing different levels of high availability. Users can configure log synchronization policies based on fault tolerance requirements. For example, to prevent data loss, the primary node can be configured to wait until the logs are persisted on the standby node before committing the transaction. Alternatively, if some data loss is acceptable, the system can be set to asynchronous log shipping, improving performance by eliminating commit-time delays. The failover process from a primary node to its standby is managed by the PolarDB control plane. PolarDB includes a resource manager that monitors the health of all instances, typically using a heartbeat mechanism. If a node failure is detected, the manager instructs the standby node to take over, ensuring seamless failover with minimal downtime. Additionally, PolarDB employs a TCP keepalive mechanism between primary nodes to rapidly detect failures. This prevents unnecessary delays in handling connections to a crashed node, allowing the system to quickly recognize failures and recover efficiently.

Recovery for distributed transactions. Similar to most databases, PolarDB employs a redo-based recovery mechanism to ensure transaction consistency and durability. The CN, acting as the transaction coordinator, records redo logs for all transaction state changes and logs each phase of execution. During recovery, the CN replays the redo logs to restore transactions to their pre-crash state. Meanwhile, PNs maintain redo logs for both local transactions and data modifications. Additionally, to support rollback operations, PNs also record undo logs for data changes. Since the CN coordinates transaction execution, the recovery process is centrally managed by the CN, with PNs following the CN's instructions to maintain consistency during recovery. When a primary node fails, the standby node takes over and applies redo logs to recover all active transactions. For transactions where the failed node acted as the CN, the new primary determines the transaction's final state based on its last recorded status, deciding whether to commit or roll back. For transactions where the failed node previously served as a PN, it awaits instructions from the corresponding CN on whether to commit or roll back its local transaction.

PolarDB adopts the Presumed Abort [45] protocol, where a transaction is considered aborted if no corresponding commit log entry is found. During recovery, the final state of an active transaction, whether to commit or roll back, is determined by its pre-crash status. The CN first restores all involved PNs' information. If the transaction had already received a commit request from the client, the CN checks whether all PNs have persisted their redo logs to storage. If they have, the CN commits the transaction and sends a commit command to all PNs; otherwise, it rolls back the transaction and instructs all PNs to do the same.

In cases where a PN crashes while executing a query, the CN detects the failure when it does not receive a response and notifies all other PNs to roll back the transaction, returning an error to the client. When that PN recovers, it checks for active transactions where it previously acted as a PN and communicates with the CN. If the CN no longer holds any information about the transaction, it indicates that the transaction has already been rolled back, and the PN follows suit. However, if the PN crashed after the CN received a commit request from the client, the CN waits for the PN to recover before continuing the commit process, ensuring transaction consistency and durability across the distributed system.

4.5 Practical experience with RDMA

PolarDB is highly co-designed with RDMA network. However, deploying RDMA at such a scale while maintaining high performance is non-trivial and requires careful optimizations.

The first key optimization focuses on connection establishment during initialization. In a large-scale RDMA-based cluster, every node must establish connections with all other nodes. In our evaluation with 2340 nodes, each node needs to create at least 2339 connections during initialization. To accelerate this process, we parallelize connection establishment across multiple threads, which is especially beneficial during failover scenarios where rapid reconnection is necessary. During cluster startup, each node retrieves cluster topology information from the manager node and attempts to connect to others. However, since nodes become available at different times, an attempted connection may fail if the target node is not yet ready. Instead of repeatedly retrying, which wastes time, the system simply skips that node and waits. When the target node eventually initiates its own connection to the current node, the current node also verifies whether it already has a connection to the target node. If not, it will establish one. This strategy ensures efficient and non-blocking connection establishment.

The second optimization focuses on connection sharing. Due to the limited memory buffer in the network interface controller (NIC), an excessive number of RDMA connections can degrade performance. To mitigate this, we limit each node pair to only two RDMA connections (one in each direction), significantly reducing the total number of connections. A single connection in one direction is shared across multiple threads within a node to maximize efficiency. Since most RDMA communication in PolarDB relies on fast one-sided operations, which typically complete within a few microseconds and occur at a moderate frequency, our evaluation confirms that a single connection per direction between two nodes is sufficient and does not introduce any performance bottlenecks.

5 EVALUATION

5.1 Setup

Test platform. Our evaluation is conducted on machines, each equipped with 2 Intel Xeon Platinum 8575C CPUs and 2TB DDR5 DRAM, running Alibaba Group Enterprise Linux Server 7.2 (Paladin). These physical machines are connected via two 100Gbps Mellanox ConnectX-6 NICs.

Workload. In our evaluation, we primarily use the TPC-C [15] workload, a long-standing industry standard for measuring OLTP (Online Transaction Processing) performance. Despite being introduced decades ago, TPC-C has undergone multiple revisions to remain relevant as computing power has increased by several orders of magnitude. It remains a valuable benchmark today, as demonstrated by its widespread adoption in evaluating modern database systems [12, 25, 30, 58, 65]. The benchmark results are measured in new order transactions per minute (tpmC), providing a standardized metric for comparing OLTP performance across different systems. Additionally, submission of a TPC-C result also requires the disclosure of the total system cost. The system must also include sufficient storage for data generated at the quoted tpmC

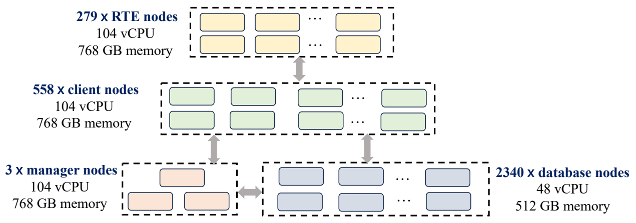

Figure 7: System configuration in TPC-C benchmark

rate over 60 days. The total cost is then combined with tpmC to compute a price/performance metric, enabling fair cost-effectiveness comparisons across database solutions.

In addition to TPC-C, we also use Sysbench [29] , another widely adopted benchmark, to evaluate specific aspects of PolarDB's performance, particularly in scale-up scenarios. This allows us to assess the impact of our optimizations on transaction processing efficiency under varying workloads, providing a more comprehensive evaluation of PolarDB's scalability and performance.

5.2 TPC-C testing

We first present PolarDB's results on the TPC-C benchmark, with the testing supervised by auditors from the TPC-C committee, and the results have been validated by them. The system configuration and detailed results are available on the TPC-C official website [2].

System configurations. Figure 7 illustrates the system architecture used during the TPC-C testing. To emulate a large-scale real-world workload, we deployed 279 remote terminal emulator (RTE) servers to simulate 1.6 billion users. The RTE servers generate HTTP requests, which are sent to the client nodes. These client nodes handle the incoming requests and communicate with the database cluster. We deployed 558 client nodes to efficiently distribute the load. For the PolarDB database cluster, we deploy 2340 nodes, each equipped with 48 vCPUs and 512 GB of memory. These 2340 database nodes run on 1170 physical servers, each of which is powered by two Intel Xeon Platinum 8575C CPUs @3.2GHz. To ensure high availability, each database node is paired with a standby node, providing fault tolerance in case of failures. Additionally, we deploy three manager nodes to oversee database cluster operations. For the RTE, client, and manager nodes, each node runs on a dedicated physical server, equipped with two Intel Xeon Platinum 8269CY CPUs @2.5GHz.

Since the TPC-C benchmark does not require storage scalability, we simplify the system configuration by using a local native file system instead of cloud storage for testing. To ensure that this choice does not affect performance results, we also compare PolarDB 's throughput on cloud storage and native file system under the same workload. As shown in Figure 12, both configurations deliver similar throughput, validating that our test setup accurately reflects PolarDB 's real-world performance. For local storage, each physical server is equipped with 14 SSDs, each with a capacity of 8.8TB, configured into two RAID-0 arrays. This setup provides sufficient storage capacity and performance to sustain the 60-day peak throughput requirement, as mandated by the TPC-C benchmark.

TPC-C configuration. For the TPC-C testing, we have followed the standard TPC-C specifications [15] for the accuracy and reliability of our evaluation. This also allows for fair and reproducible

Table 1: TPC-C results

	Total tpmC	tpmC/core	Price/tpmC
Oceanbase	707 M	10.8 K	0.54 $
TDSQL	814 M	20.6 K	0.17 $
PolarDB	2055 M	36.6 K	0.11 $

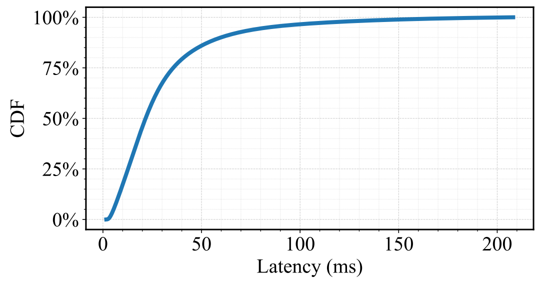

Figure 8: The distribution of latency

performance evaluations. We initialize our test with 163.2 M warehouses, totally 13.43 PB data. To accurately simulate human behavior, we incorporated some thinking time and keying time before each transaction, by following the TPC-C specification. Additionally, an additional 100 ms delay per transaction was introduced, as required by TPC-C, to better reflect real-world OLTP workloads.

Overall performance. Among all publicly available test results to date, PolarDB has achieved the highest performance. Detailed benchmark results are available on the official TPC-C website [2]. Table 1 summarizes key performance metrics and compares PolarDB against TDSQL [12] and Oceanbase [65], which rank as the second and third highest-performing databases in the public TPC-C results. The results indicate that PolarDB achieves over 2 billion tpmC, which is 2.91× and 2.52× that of Oceanbase and TDSQL, respectively. This substantial performance advantage highlights PolarDB 's ability to handle extremely high transaction volumes efficiently. Given that these systems operate with different node configurations, we also evaluate the tpmC per CPU core to provide a fair comparison. As shown in Table 1, PolarDB delivers 1.38× and 77.8% higher tpmC per core than Oceanbase and TDSQL [2].

In addition to throughput, cost-effectiveness is a critical factor in database system evaluations. Table 1 also presents a cost comparison using price per tpmC as a metric. PolarDB demonstrates the best cost-efficiency among the three systems, with its per-tpmC cost being 79.5% lower than Oceanbase and 37% lower than TDSQL. This is largely attributed to PolarDB 's novel architecture, which not only improves raw performance but also optimizes resource consumption, reducing overall costs.

Beyond throughput and cost, latency is another critical factor in OLTP workloads, as low transaction latency is crucial for delivering a smooth user experience in real-world applications. Figure 8 presents the cumulative latency distribution, illustrating PolarDB 's low-latency characteristics. The P90 latency is only 59.8 ms, significantly surpassing the typical requirement of tens of milliseconds in the TPC-C benchmark. This low-latency performance also ensures

[2]While this performance gap is primarily attributed to PolarDB's optimizations, a small portion of the improvement also arises from differences in the CPU models used across these systems.

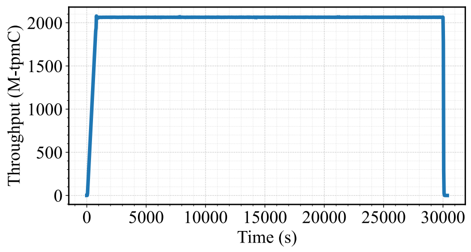

Figure 9: Throughput jitter in an 8-hour stress test

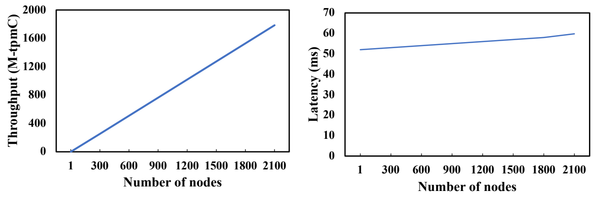

Figure 10: The scalability of PolarDB

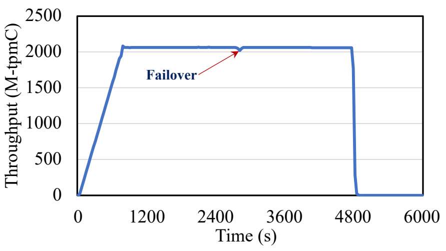

Figure 11: Performance impact of failover in PolarDB

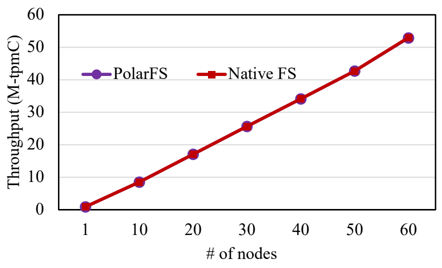

Figure 12: PolarDB performance on PolarFS *vs.* Native-FS

that PolarDB can support high-frequency transactional applications such as financial trading, large-scale e-commerce platforms, and enterprise-level database systems.

Stability. The throughput jitter rate is a critical requirement in the TPC-C benchmark, as it reflects a system's ability to maintain stable performance under sustained load. Figure 9 illustrates PolarDB 's throughput variation over time during an 8-hour stress test. At the beginning of the test, the system undergoes a brief warm-up period, where throughput gradually ramps up as caches are populated and the system optimizes execution paths. This warm-up phase lasts approximately 20 minutes, after which PolarDB reaches a stable state. Following the warm-up, PolarDB maintains consistently high throughput for the remainder of the 8-hour test, with a jitter rate of less than 0.16%, which is significantly lower than the TPC-C benchmark requirement of 2%. This exceptionally low fluctuation indicates that PolarDB can deliver predictable and reliable performance, even under prolonged high-load conditions. Such stability is crucial for large-scale OLTP applications, where fluctuations in transaction processing rates can lead to degraded user experience, increased latencies, or operational inefficiencies. PolarDB 's ability to achieve such low jitter stems from its optimized transaction processing, efficient concurrency control mechanisms, and RDMA-based communication, which collectively minimize performance degradation over time. The results demonstrate not only the robustness of PolarDB 's architecture but also its reliability in handling real-world enterprise workloads that require sustained, high-throughput operations over extended periods.

Scalability. Furthermore, we evaluate PolarDB's scalability by varying the number of nodes from 1 to over 2,000, as shown in Figure 10. The results demonstrate that PolarDB achieves nearly linear scalability as the cluster size increases, maintaining high throughput efficiency even at large scales. Additionally, the P90 latency remains stable throughout the scaling process, with an increase of only 20% when expanding to thousands of nodes. This highlights PolarDB's ability to handle large-scale deployments while preserving both throughput and latency requirement.

Crash recovery. To evaluate the high availability of PolarDB, we simulate a power failure by shutting down a machine during testing. As shown in Figure 11, we randomly turn off a server at the 2790-second mark to trigger a power failure scenario. This causes all instances running on the affected server to fail, prompting their respective standby nodes to take over. For ongoing transactions, the system automatically completes the necessary commit or rollback operations to maintain ACID guarantees. The failover process completes within approximately 2 minutes, with a temporary throughput drop of around 2.5%, demonstrating PolarDB's ability to recover quickly with minimal performance impact.

PolarFS *vs.* Native FS. PolarDB primarily relies on disaggregated storage, particularly in the Alibaba public cloud, to achieve high scalability and availability. The disaggregated storage in PolarDB is implemented using our custom-designed PolarFS and PolarStore. To evaluate the impact of PolarFS, we compare PolarDB's performance using PolarFS and a native file system (native-FS) while scaling the number of nodes from 1 to 60, as shown in Figure 12. The results indicate that PolarFS-based and native-FS-based PolarDB exhibit similar throughput. This similar performance is achieved because PolarFS leverages a lightweight user-space network and I/O stack, which fully exploits RDMA, NVMe, and SPDK technologies.

5.3 Scale-up performance

We then use the Sysbench workload to evaluate the effectiveness of our scale-up optimizations.

PolarIndex optimization. We first evaluate the performance improvement brought by PolarIndex. Since the PolarIndex mainly targets on the scenarios where has high-concurrent B+ tree structure modification operation (SMO), we use the Sysbench's insert workloads which involves the many SMO operations. To evaluate the impact on the scaling up on modern many-core systems, we configure the database instance with 88 vCPU cores. This test is conducted in the Alibaba Cloud environment, as shown in Figure 13 and Table 2. We vary the number of client threads from 4 to 512 to

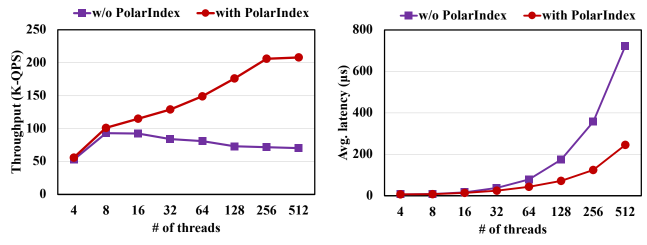

Figure 13: The performance improvement of PolarIndex

Table 2: The P95 latency (μs) with and without PolarIndex

# of threads	4	8	16	32	64	128	256	512
w/o PolarIndex	9	12	51	176	457	1065	2269	4747
with PolarIndex	9	10	37	69	110	182	302	922

simulate different application pressures and measure throughput and latency with and without the PolarIndex optimization. When the workload pressure is light (fewer than 8 threads), the number of concurrent SMOs remains low, meaning B+ tree concurrency is not yet a bottleneck. In this scenario, PolarIndex delivers performance similar to the native solution. However, as the workload pressure increases, the number of concurrent insertions rises, leading to frequent SMOs that make B+ tree concurrency the primary performance bottleneck. In the native solution (without PolarIndex), concurrent SMOs are not allowed, causing many requests to be blocked. As a result, its throughput saturates beyond 8 threads and even declines slightly with more threads due to increased contention on B+ tree access. Correspondingly, its latency continues to rise linearly as workload pressure increases. In contrast, PolarIndex significantly improves performance. Even under high-concurrency workloads (beyond 16 threads), throughput continues to scale, and latency increases much more modestly compared to the native solution. Ultimately, PolarIndex improves throughput by 1.96×, while average latency is reduced by a factor of 2.93. Notably, the P95 latency is reduced by up to 7.5×. This substantial improvement is attributed to PolarIndex's ability to enable concurrent SMOs, enhancing parallelism and minimizing lock contention, which is crucial for scalability in modern many-core database.

PolarTrans optimization. Next, we evaluate the performance improvements brought by PolarTrans, which is designed to optimize transaction management under high-concurrency scenarios. As database workloads scale up, contention in transaction management can become a significant bottleneck, limiting overall system throughput. PolarTrans addresses these issues by introducing a highly optimized, lock-free transaction management mechanism, reducing contention and improving concurrency efficiency. To evaluate its impact, we conduct tests on an 88-vCPU instance using Sysbench workloads that include read-write, insert, and point-update. We configure the system with 512 client threads, a setting that introduces high contention to stress-test the transaction management subsystem. Figure 14 presents the throughput and latency results under both configurations: with and without PolarTrans. The results demonstrate that PolarTrans increases throughput by 26.47%–50.0% across the tested workloads, while also reducing average latency by 21.3%–33.2% and P95 latency by 31.5%–55.7%.

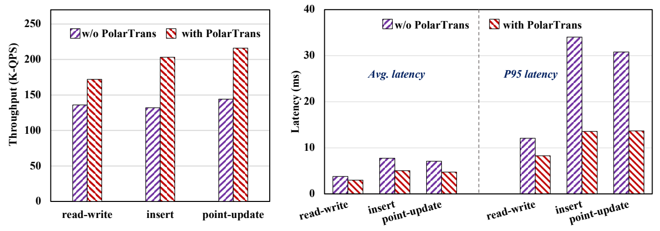

Figure 14: The performance improvement of PolarTrans

These improvements indicate that PolarTrans effectively alleviates transaction processing bottlenecks, allowing the system to sustain higher concurrency levels without performance degradation. Unlike traditional transaction management designs, where contention on a global active transaction list can cause severe performance degradation under high-concurrency conditions, PolarTrans optimizes transaction management and enables the lock-free operations. Moreover, the improvements in P95 latency indicate that PolarTrans not only enhances average performance but also provides a more stable and predictable transaction processing experience, which is crucial for latency-sensitive applications such as financial transactions and high-frequency trading.

6 RELATED WORKS

In addition to prior work detailed in Section 2.1, we discuss other relevant works in the following categories.

Primary-replica-based databases. The primary-replica-based architecture is widely adopted by many databases, such as MySQL [47], Aurora [58], Azure Hyperscale [16, 33], Azure Socrates [3]. This architecture typically consists of a single primary node and one or more replica nodes, where only the primary node handles write operations, while replica nodes are limited to read requests. Although modern servers offer hundreds of CPU cores, a single primary node can become a bottleneck in large-scale applications with write-intensive workloads. Furthermore, some databases struggle with scalability on many-core servers, limiting their ability to fully utilize available hardware resources. To address these challenges, multi-primary architectures have emerged, providing higher scalability and improved performance by distributing operations across multiple nodes. PolarDB not only introduces new designs to enhance scale-up capabilities but has also recently evolved towards a multi-primary architecture, further improving scalability and performance in cloud-native environments.

Multi-primary databases. The multi-primary database architecture is designed to enable scale-out capabilities and generally follows two mainstream approaches: shared-storage and shared-nothing. In the shared-storage-based multi-primary databases, such as IBM DB2 Data Sharing [28], Oracle RAC [9], NonStop SQL [24], Solar [71], Taurus MM [17], GaussDB [37], and PolarDB-MP [63], all primary nodes access the same underlying storage, requiring mechanisms to handle conflicts in data access while ensuring ACID compliance for transactions. These solutions show better performance in high-contention workloads at moderate scale, but face challenges to scale to hundreds or thousands of nodes.

On the other hand, the shared-nothing architecture partitions the entire database, ensuring that each primary node can only access its assigned partition. When a transaction spans multiple partitions, a distributed transaction protocol is required to maintain consistency. This architecture is widely adopted by both key-value stores [32, 49] and relational databases, such as CockroachDB [56], Spanner [14], PolarDB-X [6], TiDB [25], TDSQL [12], and OceanBase [65]. These systems are particularly well-suited for workloads that are naturally partitioned with minimal data access conflicts across nodes. While this approach provides high scalability, it often struggles with the overhead introduced by distributed transaction processing. PolarDB follows this architecture but incorporates highly optimized distributed transaction mechanisms to significantly enhance both scalability and performance.

Distributed transactions. Distributed transaction processing has been extensively optimized over the decades. More recently, several academic works have explored the use of RDMA to enhance performance in distributed transactions. Examples include Tell [38], FaRM [19], FORD [68], DrTM[61] and DrTM-H [60], all of which leverage RDMA to accelerate transaction execution in main-memory databases. However, these approaches are primarily designed for in-memory databases, which are not practical for many OLTP workloads that typically involve terabytes or petabytes of data and require persistence. On the other hand, solutions such as SLOG [50], Calvin [57] primarily focus on cross-region database clusters, where network latency is the primary bottleneck.

B-tree concurrency control. Efficient and robust concurrency control mechanisms for B-tree indexes have long been a central focus in both academia and industry, resulting in a variety of techniques widely adopted in modern database systems. Notably, ARIES/KV [42] and ARIES/IM [44] are pioneering works that support highly concurrent B-tree access. The B-link tree [34] is a classic solution that enables concurrent access through right-sibling pointers and latch coupling, making it well-suited for high-concurrency environments. More recently, researchers have proposed B-tree variants optimized for emerging hardware technologies, such as SSDs [35, 51] and persistent memory [11, 26, 67], as well as for new deployment environments like disaggregated memory [59, 74]. Inspired by the B-link tree, PolarDB introduces PolarIndex, a new concurrency control mechanism designed to enhance B tree performance in general-purpose scenarios by supporting concurrent structure modification operations (SMOs) with minimal contention. However, PolarIndex's implementation diverges from the original B-link tree design due to the constraints of the existing InnoDB-based codebase and practical considerations for production deployment, as discussed earlier.

7 DISCUSSION

Since its launch in 2017, PolarDB has continuously evolved, transitioning from a primary-replica architecture to a multi-primary distributed database. Along the way, we have made significant performance improvements through numerous optimizations. Due to space limitations, this paper focuses on some key new designs, such as PolarIndex, PolarTrans, and the Polar2PC protocol. Throughout PolarDB's development, we have also explored various optimizations inspired by both academic and industrial solutions. However, as a commercial product, we must consider important factors like forward compatibility and stability. This means that while we aim to incorporate the latest academic advancements, we also make trade-offs to ensure minimal changes to the existing codebase and maintain stability. While many academic solutions demonstrate significant improvements in specific scenarios, they often come with compatibility challenges or require invasive modifications to the existing architecture, making direct adoption impractical. Adapting such innovations for real-world deployment requires substantial effort to ensure reliability, robustness, and integration with existing systems. Given these challenges, we carefully balance performance improvements with practical constraints, leading to the design of new optimizations that maintain compatibility with previous implementations while delivering substantial performance gains. These enhancements have been deployed in production for several years, supporting hundreds of thousands of users in various applications. Looking ahead, we will continue to push the boundaries of database performance and scalability, not only by designing new solutions but also by exploring how forward-thinking academic ideas can be adapted for large-scale industrial applications, bridging the gap between theory and real-world deployment in cloud-native databases.

In addition to the scalability optimizations discussed in this paper, PolarDB has also made several architectural advancements that are crucial to its evolution. These include novel solutions for high-performance, low-latency strongly consistent reads on replica nodes [64], memory disaggregation to separate memory from CPU resources [8, 52, 69], and the PolarDB Serverless [70] architecture. These advancements have been detailed in previous publications, but are not the focus of this paper. Nevertheless, they have played a critical role in PolarDB's evolution and represent significant contributions to the development of cloud databases.

8 CONCLUSION

This paper presents the evolution of PolarDB over nearly a decade, showcasing its transformation from a scale-up-optimized system to a scale-out distributed database. To enhance scalability on many-core servers, PolarDB introduces key optimizations for B+ tree indexing and transaction management, achieving up to 1.96× throughput improvement in high-concurrency workloads. As the demand for scale-out capabilities has increased, PolarDB has evolved into a highly scalable distributed database. To further enhance performance, it optimizes the conventional Two-Phase Commit (2PC) protocol with RDMA and co-designs it with an efficient transaction management system. These advancements enable PolarDB to achieve over 2 billion tpmC in the TPC-C benchmark, significantly outperforming the second- and third-highest-ranking systems in publicly available TPC-C results, demonstrating its superiority in both performance, scalability and cost-effectiveness.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and constructive suggestions, which have helped improve the quality of this paper. We are deeply grateful to Jianying Wang, Ze Yang, Ziqian Fan, Yicong Zhu, Ming Zhao, Ninglong Weng, Guoqiang Xu, and Junwei Zhang for their support of this project. We also thank all past and present members of the PolarDB team for their dedication and continued contributions.

REFERENCES

[1] 2024. Huawei cloud-native relational database. https://www.huaweicloud.com/intl/en-us/product/gaussdbformysql.html.

[2] 2025. TPC-C All Results. https://www.tpc.org/tpcc/results/tpcc_results5.asp.

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

[4] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2019. Strong Consistency Is Not Hard to Get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. *Proceedings of the VLDB Endowment* 12, 13 (2019), 2325–2338.

[5] Eric Boutin and Steve Abraham. 2019. 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.

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

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

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

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

[10] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.

[11] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: A Persistent B+-Tree with Low Tail Latency. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2634–2648.

[12] Yuxing Chen, Anqun Pan, Hailin Lei, Anda Ye, Shuo Han, Yan Tang, Wei Lu, Yunpeng Chai, Feng Zhang, and Xiaoyong Du. 2024. TDSQL: Tencent Distributed Database System. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3869–3882.

[13] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–17.

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

[15] Transaction Processing Performance Council. 1992. On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/.

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

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

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

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

[20] CBACA Galakatos and Tim Kraska Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proceedings of the VLDB Endowment* 9, 7 (2016).

[21] Jian Gao, Qing Wang, and Jiwu Shu. 2025. ShiftLock: Mitigate One-sided RDMA Lock Contention via Handover. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 355–372.

[22] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques*. Elsevier.

[23] Rick Greenwald, Robert Stackowiak, and Jonathan Stern. 2013. *Oracle essentials: Oracle Database 12c*. " O'Reilly Media, Inc.".

[24] Tandem Database Group. 1987. NonStop SQL: A Distributed, High-performance, High-availability Implementation of SQL. In *International Workshop on High Performance Transaction Systems*. Springer, 60–104.

[25] 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.

[26] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. 187–200.

[27] IBM. 2025. Table and Index Management for Standard Tables. https://www.ibm.com/docs/en/db2/11.5?topic=tables-table-index-management-standard. "[accessed-March-2025]".

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

[29] Alexey Kopytov. 2004. Sysbench: A System Performance Benchmark. *http://sysbench. sourceforge. net/* (2004).

[30] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Matei Zaharia, and Xiangyao Yu. 2023. Epoxy: ACID Transactions Across Diverse Data Stores. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2742–2754.

[31] Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical Physical Clocks. In *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings 18*. Springer, 17–32.

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

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

[34] Philip L Lehman and S Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-trees. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 650–670.

[35] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 302–313.

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

[37] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. 2024. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3786–3798.

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

[39] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. 2024. DEX: Scalable Range Indexing on Disaggregated Memory. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2603–2616.

[40] Kai Lu, Siqi Zhao, Haikang Shan, Qiang Wei, Guokuan Li, Jiguang Wan, Ting Yao, Huatao Wu, and Daohui Wang. 2024. Scythe: A Low-Latency RDMA-Enabled Distributed Transaction System for Disaggregated Memory. *ACM Transactions on Architecture and Code Optimization* 21, 3 (2024), 1–26.

[41] Ross Mistry and Stacia Misner. 2014. *Introducing Microsoft SQL Server 2014*. Microsoft Press.

[42] C Mohan et al. 1989. *ARIES/KVL: A Key-value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-tree Indexes*. IBM Thomas J. Watson Research Division.

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

[44] C Mohan and Frank Levine. 1992. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-ahead Logging. *ACM Sigmod Record* 21, 2 (1992), 371–380.

[45] C Mohan, Bruce Lindsay, and Ron Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems (TODS)* 11, 4 (1986), 378–396.

[46] NVIDIA. 2022. 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. "[accessed-March-2025]".

[47] Oracle. 2025. MySQL. https://www.mysql.com/. "[accessed-March-2025]".

[48] Kerry Osborne, Randy Johnson, Tanel Põder, and Kevin Closson. 2011. *Expert Oracle Exadata*. Springer.

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

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

[51] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2011. B+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-Based Solid State Drives. *arXiv preprint arXiv:1201.0227* (2011).

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

[53] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data*. 433–448.

[54] Dale Skeen. 1981. Nonblocking Commit Protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 133–142.

[55] CAE Specification. 1991. *Distributed Transaction Processing: the XA Specification*. X/Open.

[56] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The Resilient Geo-distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1493–1509.

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

[58] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[59] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+ Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data*. 1033–1048.

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

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

[62] John Worsley and Joshua D Drake. 2002. *Practical PostgreSQL*. " O'Reilly Media, Inc.".

[63] 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*. 295–308.

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

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

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

[67] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: A Lock-Free PM-Friendly Persistent B+-Tree for eADR-Enabled PM Systems. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1187–1200.

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

[69] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1900–1912. https://doi.org/10.14778/3467861.3467877

[70] Yingqiang Zhang, Xinjun Yang, Hao Chen, Feifei Li, Jiawei Xu, Jie Zhou, Xudong Wu, and Qiang Zhang. 2024. Towards a Shared-Storage-Based Serverless Database Achieving Seamless Scale-Up and Read Scale-Out. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 5119–5131.

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

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

[73] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design Guidelines for Correct, Efficient, and Scalable Synchronization Using One-Sided RDMA. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[74] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks. In *Proceedings of the 2019 International Conference on Management of Data*. 741–758.