PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications

Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, Han Liao, Zilin Chen, Xiaojian Fang, Mo Chen, Chenghui Liang, Yanxin Luo, Huanming Wang, Songlei Wang, Zhanfeng Ma, Xinjun Yang, Xiang Peng, Yubin Ruan, Yuhui Wang, Jie Zhou, Jianying Wang, Qingda Hu, Junbin Kang

{mingsong.cw, lifeifei, qushan, jianghang.loujh, jianwei.zhao, dengcheng.hedc, mengshi.sunmengshi, yingqiang.zyq, sh.wang, xueqiang.wxq, changyuan.lh, zilin.zl, xiaojian.fxj, chenmo.cm, chenghui.lch, luoyanxin.pt, huanming.whm, jiujiang.wsl, zhanfeng.mzf, xinjun.y, pengxiang.px, yubin.ryb, yuhui.wyh, jarry.zj, beilou.wjy, qingda.hqd, junbin.kangjb}@alibaba-inc.com

Alibaba Group

***Abstract*—Cloud computing is on the rise, which promotes new breeds of database systems to accommodate the cloud environment. The development of cloud-native databases reveals three trends. One is the adoption of multi-datacenter (DC) deployment to survive the downtime of any single site. Another is the separation of computation and storage resources to achieve higher elasticity and scalability. The last is the support of HTAP to eliminate data redundancy and system complexity from heterogeneous databases.**

To cater to these trends, we design a distributed relational database called *PolarDB-X*, which is built on top of the cloud-native database PolarDB. It hence inherits many cloud-native features, such as multi-datacenter deployment and elasticity. To achieve cross-DC capability, it leverages Paxos and hybrid logical clock to achieve durability and snapshot-isolation consistency with low coordination costs. For resource elasticity, since the underlying PolarDB supports rapid migration of tenants between nodes, *PolarDB-X* can quickly scale the cluster to cope with a sudden traffic increase. For HTAP support, with the help of read replicas and a HTAP executor, *PolarDB-X* can improve the latency and parallelism of analytical queries without impacting concurrently-running TP workloads. Using its MPP engine and an in-memory column index, the efficiency of analytical queries can be further enhanced. *PolarDB-X* is now a cloud database service at Alibaba Cloud. We have learned many useful lessons from its development and operation, and have incorporated those into our design and analysis.

I. Introduction

In order to meet the ever-increasing demand for supporting larger data volume and higher transaction throughput, enterprise applications tend to adopt a shared-nothing architecture for their databases that can distribute data shards on many data stores, such as transactional KV-stores [1], [2], [3], [4] or relational databases like MySQL [5], [6], [7], [8], [9] and PostgreSQL [10], [11], [12], [13]. In such database architectures, the support of fully distributed ACID transactions across shards is essential to simplify the building and reasoning of distributed systems and improve application agility. Meanwhile, with the prevalence of cloud computing, more and more enterprise applications and underlying databases are migrating

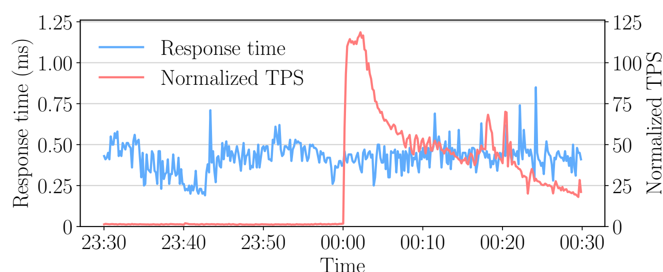


Fig. 1. 122x increase in transactions per second along with stable response time observed in Alibaba Double 11 Shopping Festival 2018. Throughput is normalized by the average TPS for the day before midnight on Nov. 11.

to the cloud. Three challenges arise for operating a distributed relational database service on the cloud.

First, while migrating to the cloud, enterprises will follow the best practices of ready-made multi-DC (data center) deployment. The typical way is to disperse database replicas in different datacenters, where a consensus protocol (known as quorum [14], [15] such as paxos [16], [3], [17], raft [18]) is used to synchronize modifications between replicas to prevent data loss and to offer continuity when a single datacenter fails. However, the leader nodes of different database shards may be located in different datacenters. When a distributed transaction involves multiple database shards (such as updating the primary index and/or global secondary indexes), the cost of two-phase-commit (2PC) and consensus coordination across datacenters can be prohibitive and the performance is significantly compromised due to potentially high network latency. Therefore, *mitigating the impact of cross-DC network latency* and *reducing the cost of cross-DC distributed transactions* are key considerations in designing a distributed database for a cloud-native environment.

Second, large-scale Internet applications, such as in e-commerce and social network, often witness tens or even hundreds of times more traffic during business peaks, e.g., during Alibaba's Double 11 Shopping Festival. Figure 1 shows the total transaction throughput of Alibaba's databases at the midnight of November 11, 2018, which had suddenly

increased by 122 times in just 1 second. In a traditional shared-nothing architecture, operations such as scaling a cluster and adding nodes need migrating data between servers, which is inherently cumbersome and time-consuming. Therefore, companies usually have to scale their database systems weeks or even months in advance before peaks arrive, leading to a waste of resources. The *demand for rapid resource elasticity* from such applications cannot be effectively addressed by traditional shared-nothing databases, and this brings new opportunities to cloud-native distributed databases.

Third, guided by the principle of *one size doesn't fit all* [19], enterprises often use separated database systems to handle OLTP and OLAP workloads. They rely on ETL tools to import data from an OLTP system to an OLAP system, but this increases the redundancy of data storage and the complexity of system maintenance. Processing transactional and analytical workloads simultaneously in a single database system would be ideal, but then resource isolation becomes a main challenge. When running analytical queries, the transaction throughput is often adversely affected, due to the contention of system resources such as CPU, memory and network bandwidth on top of resolving read-write conflicts. For cloud-native distributed databases, there is an opportunity to make use of elastic resources for *processing OLTP and OLAP workloads simultaneously*, such that BI reports can be timely generated without affecting transactions from front-end applications.

To tackle these challenges, this paper describes Alibaba Cloud's distributed relational database named *PolarDB-X*. It is built on top of the cloud-native shared-storage database PolarDB [20], [21], [22] to inherit many cloud-native designs and additionally adopts a novel CN-DN-SN (Computation-Database-Storage Nodes) three layer architecture. The main design goal is to fulfill three aforementioned demands, namely, optimized cross-DC distributed transactions, rapid elasticity for cloud applications, and efficient hybrid OLTP and OLAP workload processing.

1) **Cross-DC Transactions**. *PolarDB-X* uses *HLC-SI*, which relies on the hybrid logic clock [23] to achieve cross-shard data consistency. Since *HLC-SI* complies with snapshot isolation (SI), we compare it to other SI implementations, such as centralized timestamp ordering TSO [24], [1]. More specifically, the network delay of accessing TSO across datacenters will increase transaction latency and affect throughput substantially. Our experiment shows that *HLC-SI* outperforms TSO under a 3-datacenter deployment, and its peak write throughput is 19% times higher.
2) **Elasticity**. Cloud-native relational databases from major cloud vendors all use the separation of computation and storage architecture [14], [25], [20], [26], which renders a significant advantage of elasticity, i.e., add RO (Read-Only) nodes to scale read throughput in minutes. Furthermore, Alibaba's PolarDB supports multi-tenancy (PolarDB-MT), where each tenant represents a collection of schemas/-databases or tables. PolarDB-MT allows scaling of writes by binding different tenants to multiple RW (Read-Write) nodes and distributing write requests among these nodes accordingly. By leveraging these features, a *PolarDB-X* cluster can quickly scale out. Our evaluation shows that, with 160 million rows and 40 GB data volume, *PolarDB-X* can double its size within 4-5 seconds, which is more than a hundred times faster than the traditional data transfer method widely used in a traditional shared-nothing architecture.
3) **HTAP**. *PolarDB-X* 's optimizer can identify whether a query belongs to TP or AP workload based on its estimated cost. AP queries will be isolated and executed in a separate thread pool, and queries in execution can be preempted to prevent others from starvation. When CPU and network resources used by a query reaches its quota, it will be put into the waiting queue for rescheduling. Since PolarDB can quickly add read-only replicas without copying data, we can alternatively run analytical queries on newly extended RO nodes, completely isolated from TP workloads in RW nodes. In our experiment of running TPC-C and TPC-H benchmarks in a mix, after enabling the HTAP mode, the throughput of TPC-C becomes very stable and almost unaffected by concurrent TPC-H tests. The average TPC-H query latency is reduced by 2.7, 5.0 and 5.7 times, corresponding to adding one to three additional read replicas. With the MPP execution engine and PolarDB's in-memory column index, the performance can be further improved.

We have been operating *PolarDB-X* as a cloud service for enterprise customers, and we have learned from our experiences and added a series of DBA- and developer-friendly features in *PolarDB-X* to enhance usability, e.g., anti-hotspot, automated traffic control, and index recommendation.

Next, Section II introduces the architectures of both *PolarDB-X* and the underlying PolarDB[1]. Section III talks about *PolarDB-X*'s Paxos and Multi-DC deployment. Section IV presents our cross-DC distributed transaction processing. Section V explains how to leverage PolarDB-MT to quickly scale a *PolarDB-X* cluster. Section VI discusses the HTAP executor, MPP execution engine and in-memory column index. Section VII gives the experimental results. Section IX reviews the related work and Section X concludes the paper.

II. System Overview

A. Architecture

PolarDB-X is a distributed database compatible with MySQL. As shown in Figure 2, *PolarDB-X* has the following components: Global Meta Service (GMS), Load Balancer, Computation Node (CN), Database Node (DN), and Storage Node (SN). By disassembling the relational database kernel into three layers (CN-DN-SN), this architecture has achieved excellent elasticity and scalability: CN handles distributed transactions and queries; DN solves single-shard transactions and cross-DC replication; SN persists data inside a single DC; and each layer can be scaled independently.

GMS. The GMS is the control plane of *PolarDB-X*. It manages the system's metadata, such as cluster membership,

[1]Since *PolarDB-X* is built on top of PolarDB, when we describe *PolarDB-X*, related designs in PolarDB are also discussed as part of it.

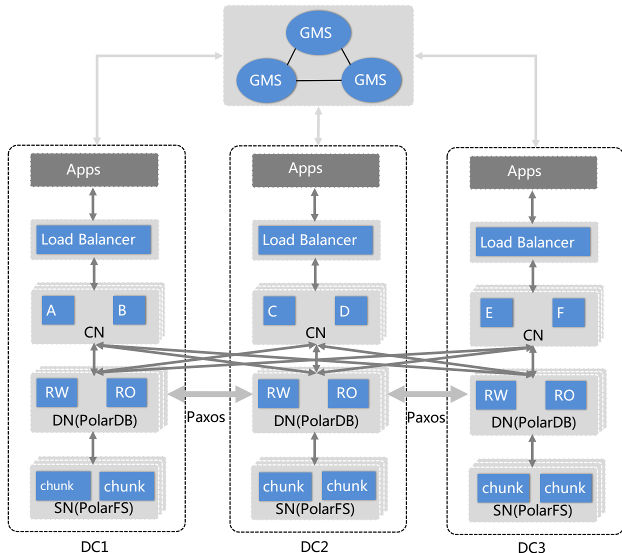


Fig. 2. System architecture of *PolarDB-X*

catalog tables, table/index partition rules, locations of shards, and statistics. It also maintains system tables, such as sequences, stored SQL plan baselines, accounts and permissions. In addition, some background tasks are running on GMS. For example, it schedules data redistribution according to the load, and also manages node registration and failover. In production, we deploy a 3AZ PolarDB as the GMS.

Load Balancer. The load balancer is a cloud network infrastructure [27]. It exposes a single entry point (i.e., virtual IP) to SQL clients for each *PolarDB-X* instance. It is location aware, which tends to disperse client connections to nearby backends (e.g., CN servers co-located in the same datacenter). For example, in Figure 2, the load balancer of DC_1 will first redirect connections to CN servers A and B. Only when they are unavailable, will the request be routed to CN servers in DC_2 and DC_3.

Computation Node. A CN is always deployed in the same datacenter with underlying DN (PolarDB) and SN (PolarFS). Tables are divided into shards and stored in DN. Each shard is replicated to at least three DNs through the Paxos protocol with leader lease. The leader DN of each shard may be located in different datacenters. CN will forward a shard's write requests to datacenters where the leader is located, and forward read requests to the local data center (when it has a fresh enough snapshot) to eliminate cross-DC network latency. CN contains components including a distributed transaction coordinator, a cost-based optimizer and a query executor (with features like isolation of TP and AP traffic, MPP executor for AP queries). Clients establish connections with CN servers and send SQL queries to them. A CN server parses SQL statements, analyzes data locations, and acts as a transaction coordinator. For a distributed transaction, it first starts the transaction and forwards statements to corresponding DN servers. After that, it merges the results, commits the transaction, and returns the final result to the client. Since CN servers are stateless, they can be rapidly scaled to handle more transactions and queries concurrently.

Database Node and Storage Node. DN and SN in *PolarDB-X* are implemented by the same counterparts in PolarDB, a cloud-native database with shared storage architecture [22], [21]. DN is equivalent to the database nodes (RW and RO nodes) in PolarDB [22], [21], and SN corresponds to the chunk server in PolarFS [20]. For the basic PolarDB (as detailed in Section II-C), an instance includes one primary (i.e., RW node) and multiple read replicas (i.e., RO nodes). RO replicas can be scaled horizontally to improve the capacity of processing analytical queries, while having little impact on transactional workloads. Later in Section V, we will introduce PolarDB-MT, a variation of PolarDB where an instance can have multiple RW nodes to improve the aggregated write throughput. Like a traditional database kernel, a RW/RO node contains a SQL processor, transaction engines (like InnoDB [28]), and a buffer pool to serve local transactions and queries.

PolarFS. All data is persisted in PolarFS [20]. PolarFS is a durable, atomic and horizontally scalable distributed storage service. It provides virtual volumes that are partitioned into chunks of 10GB size, which are distributed in multiple storage nodes. Each DN has one volume, where a *PolarDB-X* can have multiple volumes. Each volume contains up to 10K chunks and can provide a maximum capacity of 100TB. Chunks are provisioned on demand so that volume space grows dynamically. Each chunk has three replicas in each datacenter and linear serializable is guaranteed through *Parallel Raft*, which is a consensus protocol derived from Raft.

The throughput of the entire system can be scaled by adding more CN and DN servers. Since CN does not have persistent state, no data movement is involved. For DN, PolarDB increases the read throughput near linearly by adding RO nodes, and PolarDB-MT allows the use of multiple RW nodes to scale write throughput. Neither action above moves the data. Therefore, compared with traditional shared-nothing databases that always require copying data, scaling a *PolarDB-X* cluster is much faster. The task of extending storage capacity and I/O throughput is decoupled to SN, which is transparent to the upper layers and can be achieved by adding more SN nodes.

B. Data Partition

PolarDB-X uses hash partitioning on the primary key to slice tables/indexes into shards. If the table of interest does not specify a primary key, an implicit primary key will be added, which is an auto-increment `BIGINT` type and is invisible to users. An advantage of the hash partitioning is that it can distribute data evenly across shards and reduce the chance of having hotspots. For example, when the primary key is an auto-increment integer (or an ever-increasing timestamp) and the range partitioning is used, bulk data insertions cause most writes to fall on the last shard, making it a hotspot.

Similar to F1 [29], *PolarDB-X* supports both local and global indexes. The local index is also partitioned by the partition key, so that updating the index does not lead to a distributed transaction. The global index is partitioned by the indexed columns and stored as a hidden table, where both clustered indexes and non-clustered index are supported. When

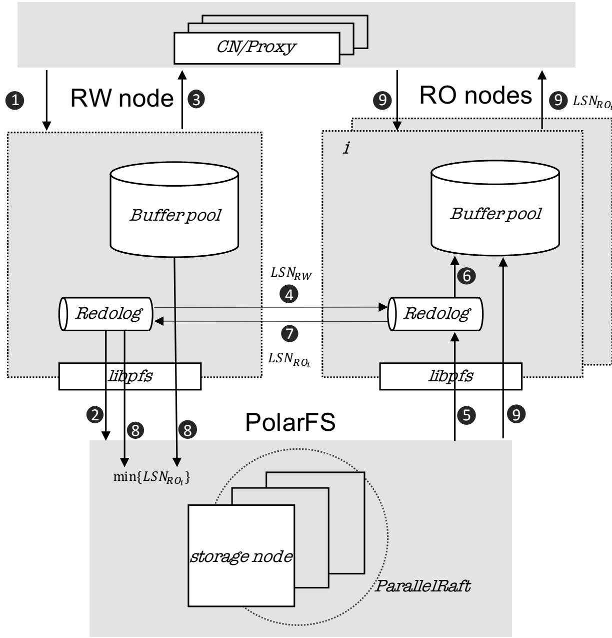


Fig. 3. PolarDB Architecture

updating records, the primary key index and related secondary indexes are updated in a single distributed transaction to ensure ACID. The clustered index can reduce cross-shard data retrievals. For example, after a query retrieves a set of primary keys from the global secondary index, it needs to read the corresponding rows from the primary index on different shards. With a clustered index, we can efficiently read all required columns from the index to avoid scattered reads.

A set of tables could have identical partition key. Queries to these tables usually specify a value on partition key, either explicitly (through "where equal" condition) or implicitly (equi-join on the partition key). In *PolarDB-X*, these tables can be declared as a *table group*, and follow exactly the same partition rule and shard placement policy. In a table group, the set of table shards from the same partition are defined as a *partition group*. The shards in a partition group are always located on the same DN. During data redistribution, all tablets in a partition group will be re-sharded (split or merged) or migrated as a whole. Table group helps reduce the latency introduced from having remote data. For example, in a table group, equi-join on the partition key can be optimized by performing partition-wise join, without reading remote data or redistributing data between shards. Spanner [3] and F1 [29] introduce a new data model named hierarchical schema, which is different from the relational model, to solve similar problems. In contrast, *PolarDB-X* chooses to stick to the MySQL-compatible schema and adds table group as a syntax extension.

C. PolarDB

For simplicity and clarity, we only depict the setting of a basic PolarDB deployed in a single datacenter. We demonstrate how writes to the RW node are synchronized to RO nodes, providing snapshot reads and session consistency.

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

The latest snapshot version of a RO node usually lags behind that of RW for a few milliseconds. For clients that require session consistency [30] across RW and RO nodes, CN tracks the latest timestamp LSN_{RW} of RW and piggybacks the timestamp when forwarding read requests to RO nodes. The RO will wait until its snapshot version number is no less than LSN_{RW} before processing and responding to the query. In this way, without increasing storage cost and bearing data copy overhead, RO replicas can be scaled horizontally to improve the system's read throughput, while having little impact on the performance of the RW node.

III. Replication

To support multi-DC deployment for *PolarDB-X*, we enhance PolarDB (used as the DN) using Paxos with a leader lease, which uses optimizations like asynchronous commit, batching and pipelining to efficiently synchronize changes between multiple datacenters. It contains the following roles:

- *Leader* — all write operations are completed here.
- *Follower* — it receives redo log records from the leader and replays the log. A follower could be elected as the new leader if the original one fails.
- *Logger* — it is a special follower node that only documents redo log records and has no data, i.e., it cannot provide database services. It can participate in leader election but cannot be selected as the leader.

Unlike in Aurora, to achieve extremely low storage I/O latency (via RDMA), our cross-datacenter data replication is not achieved at the SN layer, but at the DN layer. PolarDB instances inside *PolarDB-X* transfer redo logs across datacenters. RO nodes can be created on both leader and follower instances.

Next, we show how Paxos collaborates with redo logs in this setup. The leader is responsible for executing transactions. Before a transaction commits, the redo log entries are flushed to PolarFS, which will also be sent to followers using Paxos. Once a follower receives the log entries, it copies them to the

log buffer and writes them to the log file. After the logs are persisted in PolarFS, the follower acknowledges the leader. When majority nodes have retained changes, the leader will advance the DLSN (Durable LSN). The log entries before DLSN will not be lost, even when a datacenter disaster occurs. Hence, the leader can safely flush dirty pages modified before DLSN from the buffer pool to PolarFS, and inform followers that DLSN has been advanced in the next transmission. After receiving the latest DLSN from the leader, the follower applies all redo log entries before that. Changes after DLSN cannot be applied by RO, because if a single datacenter fails and the leader is re-elected, the redo log entries after DLSN may be truncated by the new leader. When a follower recovers from a crash, it must ensure that it does not apply any redo log behind (larger than) DLSN.

Asynchronous Commit. Transaction commit is also driven by the advancement of DLSN. A transaction is divided into multiple mini-transactions (MTR), which are a group of contiguous redo log entries. When the DLSN exceeds the largest LSN of a transaction's last MTR, the transaction is considered to be committed. The leader then returns the success of the `COMMIT` statement to its SQL client. Note that the persistence of redo logs in multiple datacenters requires round-trip communication. If a foreground thread is blocked to wait for ACKs from followers, the leader could have a large number of threads blocked in the waiting state, which will severely affect transaction throughput. To resolve this issue, asynchronous transaction commit is adopted. More specifically, after the foreground thread invokes Paxos to send redo log entries to the followers, it stores the transaction's context in a map data structure and then proceeds to process other transactions. A new `async_log_committer` thread is added to monitor DLSN's changes. When the followers return acknowledges and DLSN is advanced, `async_log_committer` iterates the map to find a list of transactions whose last MTR's LSN exceeds DLSN. It restores their contexts, commits them and returns the results to the client.

Pipelining and Batching. Asynchronous commit relies on Paxos to support pipelined transmissions of redo logs. The leader continuously sends redo log entries to the follower, and it can send a new batch of logs without waiting for the acknowledgement of previous batches. When the transmission delay is far lower than the propagation delay (e.g., in a high-latency and high-throughput network environment), pipelining can effectively improve the throughput. Synchronizing data through Paxos requires extra control information. In order to integrate Paxos in the redo log stream, we have added a special redo log entry type called `MLOG_PAXOS` that manages Paxos metadata in a batched manner. This entry is 64 bytes and contains metadata like epoch, index, LSN range of redo log entries, and checksum. Since each MTR only contains a small amount of changes (up to a few hundreds of bytes), it is expensive to add a `MLOG_PAXOS` log entry for each MTR. Therefore, multiple MTRs are batched in a single `MLOG_PAXOS` (maximum 16KB) to enlarge the payload, which greatly improves the log replication throughput.

Leader Election. Paxos is responsible for liveness detection and new leader election. When a leader failure is detected and more than half of the nodes are active, Paxos will start a leader election. Paxos guarantees that the newly chosen leader has complete log entries before DLSN. However, there may be some redo log entries with LSN larger than DLSN on the old leader. They may have not been persisted to other datacenters, but have already been applied by the leader. This will cause the dirty pages in the old leader's buffer pool to conflict with new leader's. Hence, after the leader election and the old leader rejoins the Paxos group, some additional memory state cleaning is required on the old leader. It needs to synchronize with the new leader, determine the range of redo log entries that are not submitted, evict dirty pages related to them, and reload clean pages from PolarFS. In contrast, the situation on an old follower node is much simpler, it only needs to discard all redo log entries behind (larger than) DLSN and connect to the new leader to resync log entries.

IV. TRANSACTIONS

The key to distributed transactions is to determine the order between transactions and enforce correct visibility. Using centralized clock in a distributed database is conventional to achieve snapshot isolation. TSO-SI is such a solution used by Percolator [24] and TiDB [1], which offers ascending clock as snapshot timestamp and commit timestamp. The former determines appropriate record version to read, while the latter orders transactions globally across all nodes. Atomic updates on multiple data nodes are achieved through two-phase commit (2PC). However, the centralized clock server may become a single point of failure and a potential performance bottleneck. Frequent access to TSO increases the latency of transaction processing, especially in the case of cross-datacenter deployment. To alleviate above problems, Clock-SI [31] instead relies on loosely synchronized physical clock on each node, but it may suffer from the delay caused by clock skew. In *PolarDB-X*, we propose ***HLC-SI***, which utilizes hybrid logical clock (HLC) to track event causality in a distributed database.

HLC Primitives. HLC contains both physical clock and logical clock in a single timestamp. It can not only track the causal relationship of across-node events, but also keep the logical clock value close to the actual physical clock value. We implement the HLC timestamp (denoted as hlc) as a 64-bit integer $\{reserved : 2, pt : 46, lc : 16\}$. The lower 16 bits ($lc$) represent the logical clock, while the upper 46 bits (pt) store the physical time. The finest granularity of the physical clock is one millisecond, i.e., it counts 65,535 times per millisecond and supports more than tens of million transactions per second, which is sufficient for extremely large-scale databases. More bits can be allocated to $hlc.lc$ if needed.

Each node in the cluster has a local physical clock $node.pt$ in millisecond, and also maintains its own HLC timestamp $node.hlc$. The HLC clock has three primitives:

- *ClockUpdate($e.hlc$).* After an event e transmitting the HLC timestamp has occured (e.g., in 2PC), a participant

receives the *snapshot_ts* and *commit_ts* from the coordinator. If the incoming HLC timestamp $e.hlc$ is higher than the node's own $node.hlc$, it will advance $node.hlc$.

- *ClockAdvance()*. It increments the logic clock part of the HLC by one to obtain the next HLC timestamp. If the local physical clock $node.pt$ is higher than $node.hlc$, it overwrites the HLC timestamp. This function ensures that the HLC value is close to the node's physical clock, and the difference between the two is bounded.
- *ClockNow()*. It obtains the latest HLC timestamp similar to *ClockAdvance*, except not incrementing the logic clock.

The original HLC algorithm proposed by Sandeep et al. [23] increments the logical part of the HLC timestamp by one each time a message is exchanged between nodes. In *HLC-SI*, we introduce optimizations for the maintenance of the HLC timestamp. *First*, the logic clock part is not incremented in *ClockUpdate* and *ClockNow*, which can prevent the 16-bits logic clock space from running out too fast. *Second*, we minimize the calls to *ClockUpdate*. For example, in 2PC, the coordinator will not call *ClockUpdate* to update the local HLC timestamp when it receives responses from participants. Instead, after receiving all *prepare_ts* timestamps, *ClockUpdate* is called only once with the maximum timestamp seen from all participants. Since modifications of the global variable $node.hlc$ could become a bottleneck in multi-core systems, less updates reduces lock contentions significantly. At last, we prove below that, after applying aforementioned optimizations, *HLC-SI* still preserves the properties of snapshot isolation.

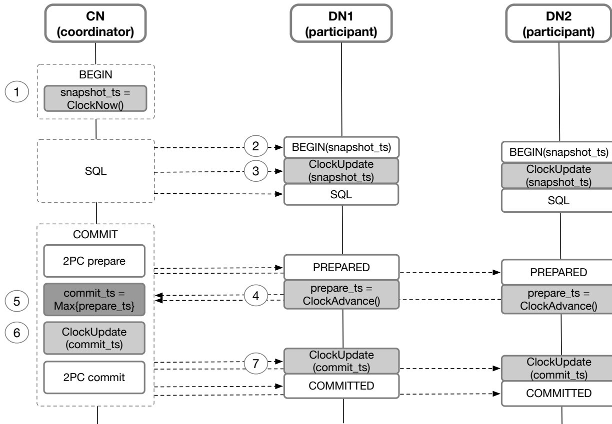


Fig. 4. HLC-SI and two-phase commit

HLC-SI. Figure 4 shows how *HLC-SI* obtains timestamps for a distributed transaction, synchronizes and advances the HLC timestamp among nodes. When a transaction starts, the CN (coordinator) $node_i$ calls *ClockNow* to acquire the transaction's snapshot timestamp *snapshot_ts* ①. CN then sends the transaction together with *snapshot_ts* to all participating DN nodes ②. Suppose one of them is $node_j$, and it calls *ClockUpdate(snapshot_ts)* to update $node_j.hlc$. This ensures that afterwards $node_j.hlc$ is larger than or equal to *snapshot_ts* ③. When the participant encounters a transaction in *PREPARED* state that modifies the data in its read set, it needs to wait for the transaction to complete. Because the commit timestamp of a transaction in *PREPARED* state is uncertain, it could be either greater than *snapshot_ts* (i.e., invisible) or not (i.e., visible).

In the first phase of 2PC, after the DN completes the conflict validation of the transaction's write set, the DN changes the status of transaction to *PREPARED*, calls *ClockAdvance* to obtain a prepare timestamp $node_j.prepare_ts$, and then returns it to the CN ④. In the second phase of 2PC, after the CN receives *prepare_ts* from all participants, it chooses the maximum one as *commit_ts* like in Clock-SI ⑤. Then it calls *ClockUpdate(commit_ts)* to synchronize its local HLC timestamp, and sends *commit_ts* to all DNs ⑥. After DNs receive *commit_ts*, they also call *ClockUpdate(commit_ts)* to update their local HLC timestamps ⑦.

Proof of Correctness. Here we prove how HLC-SI obeys properties of SI. That is, for any pair of transactions, say T_1, T_2, if $T_1.commit_ts$ is less than or equal to $T_2.snapshot_ts$, T_1 must be visible to T_2, and otherwise T_1 is invisible to T_2. When T_2 accesses the records that have been modified by T_1, there can be three cases: (1) T_1 is already committed. the visibility is determined by the *committed_ts* of T_1. (2) T_1 is in *PREPARED* state. T_2 will wait until T_1 completes and then determine the visibility of records modified by T_1 as in Case 1. (3) If T_1 has not entered the *PREPARED* state yet (i.e., in *ACTIVE* state), T_1 must be invisible to T_2.

Proof. Suppose on $node_k$, T_2 observes that T_1 is in *ACTIVE* state. When the coordinator sends T_2 to $node_k$, $node_k.hlc$ is updated to be equal to or greater than $T_2.snapshot_ts$:

$$T_2.snapshot_ts <= node_k.hlc$$

Since T_1 is in *ACTIVE* state, we must have:

$$node_k.hlc < node_k.prepare_ts^{T_1}$$

By definition of commit_ts, we must have:

$$node_k.prepare_ts^{T_1} <= T_1.commit_ts$$

Finally, according to transitivity's law, we have:

$$T_2.snapshot_ts < T_1.commit_ts \qquad \square$$

In fact, T_1 is impossible to be visible to T_2 unless T_1 has already entered the *PREPARED* state on every node in $participants(T_1) \cap participants(T_2)$. Because if there is any node on which T_1 is still in the *ACTIVE* state, according to above inferences, $T_1.commit_ts$ must be greater than $T_2.snapshot_ts$, i.e., T_1 is invisible to T_2.

V. ELASTICITY

Multi-Tenancy. Some customers operate software-as-a-service (SaaS) businesses and advocate the need for a multi-tenant database. A typical SaaS application has a large number of subscribers. To ease the support for many subscribers, the SaaS application provides distinct schemas/databases (logically) for each subscriber. The data of each subscriber can be regarded as a tenant in the database. Since each tenant is logically isolated and independent, there is no cross-tenant

transaction. Meanwhile, in order to reduce costs, multiple tenants are usually consolidated and stored in one database instance. However, when some tenant's access becomes a hotspot where more resources are needed, the tenant shall be migrated to other database instances. This process is expensive and takes time proportional to the data volume involved.

In *PolarDB-X*, cross-shard transactions are implemented in CN through distributed transactions. There is no cross-shard transaction in DN, so that a shard or a partition group can be regarded as the unit of tenant. Multiple tenants are stored in one DN, and when database is expanded horizontally, tenants can be migrated to new available DNs for load balance. A tenant is defined as a collection of schemas/databases or tables. In both scenarios, the PolarDB instance inside *PolarDB-X* encounters a bottleneck, that is, only one RW node can handle write requests for all tenants. Therefore, we extend PolarDB inside *PolarDB-X* to support multi-tenant, which allows the use of multiple RW nodes for scalable writes, but at the cost of not supporting cross-tenant transactions. These RW nodes still share the storage, but different RW nodes operate on disjoint portions of the data (divided by tenants). The strict constraint is that DML between RW nodes will not conflict with each other. To enforce this, each tenant must be bound to only one specific RW node at any given time. For elasticity, the number of RW nodes and tenant-RW bindings can be adjusted at run-time, and tenants can quickly migrate between RW nodes.

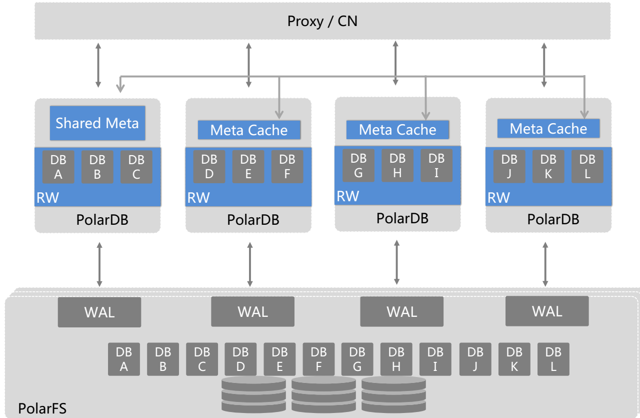


Fig. 5. PolarDB Multi-Tenant Architecture

Design of PolarDB-MT. For each data table, only one RW node can write to it, and hence there is no conflict during the modification of B+Tree pages and rows. As shown in Figure 5, each RW node has its own private redo log, i.e., no write contention in redo log. Besides, there is no global ordering sequence or dependency between these logs, since they record modifications to different tenants, which are logically uncorrelated. Therefore, redo logs belonging to different tenants can be concurrently replayed to recover database states in parallel. In fact, if one RW node fails, one or more other RW nodes can take over its redo log. They divide log entries according to the tenant, replay them, complete the recovery process and restore services. Similar to PolarDB (Section II-C), in PolarDB-MT, each RW node can also have a configurable number of RO nodes, who serve read transactions at the Repeatable Read or Read Committed isolation level.

All RW nodes share a global data dictionary instead of maintaining a distinct private one for each node. Only one RW node can grab a lease. The leaseholder (i.e., the master RW node) manages the data dictionary and is delegated to make all modifications to the dictionary. Other RW nodes maintain a read cache of the dictionary, and only cache the metadata of tables they open. Since a table is bound to and written by a single RW node, the metadata of a table will be cached by at most one RW node. At the beginning and end of executing a DDL statement, the owner RW node needs to acquire an exclusive MDL (metadata lock), which will block all subsequent DML/DDL statements for the table. It then modify the data dictionary and forward the metadata modification request to the master RW. The master RW checks whether the modification request is valid, that is, only the owner of a tenant has the right to modify the metadata of its internal table/database. Once the check is passed and the write succeeds, the in-memory table metadata cached in the RW node who initiates the write request will also be updated. After that, the MDL is released, and all blocked DML/DDL statements continue to execute. The above process ensures the consistency of the table metadata among the RW nodes. The table metadata is also cached in RO nodes, and it is synchronized with the RW node through redo log replication.

Tenant Transfer. The binding information of RW nodes and tenants is stored in an internal system table, which is shared with upper-level components such as proxy or CN. Hence, they know to which RW node a DML statement should be routed. Each RW node subscribes to the updates of the binding info and obtains a lease from the master RW node, in order to ensure the correctness and freshness of the binding info. When a RW node receives a transaction, it first checks whether all related tables are bound to the node and retains the lease, otherwise it immediately returns an error. When the RW node finds that the lease is lost, it will suspend the submission of all outstanding transactions and try to re-acquire the lease. When time out or after obtaining the lease, if it refreshes the binding info and finds that some tenants have migrated to other RW nodes, it will immediately abort all affected transactions.

During migrating or transferring a tenant from the *source* RW to the *destination* RW, the proxy or CN is responsible for keeping client connections alive. They pause new transactions to the tenant and stop forwarding them to the source RW. Then, they wait for the source RW node to complete all ongoing DML/DDL statements gracefully. After that, the source RW will flush all dirty pages associated with the tenant to PolarFS, clean tables' cached metadata and close resources such as files belonging to the tenant. Finally, it asks the system table to update the binding info. Meanwhile, The destination RW opens tenant's files, fetches necessary metadata from the master RW node, initializes itself and starts serving new transactions to this tenant. After all above steps are completed, the proxy or CN servers connect to the destination RW node, restore session states, and then forward paused transactions for

execution. The whole process is transparent to the application, just as the upcoming transactions are blocked by the migration DDL statement for a while.

Scale *PolarDB-X* cluster. Using PolarDB-MT, we can quickly add a DN node to increase both read and write throughput of *PolarDB-X*: (1) An empty RW node is created. The elapsed time depends on the time to find available resources (e.g., virtual machines); (2) The node is registered to GMS. GMS then counts the load distribution and generates a migration plan containing those tenants to be moved to the new node; (3) The plan is executed. Tenant migrations with different pairs of source and destination RWs can be scheduled in parallel. As described previously, during the migration, new transactions would be suspended for seconds, but read requests can be forwarded to the RO node to mitigate the impact.

VI. HTAP

With the decoupled storage architecture, *PolarDB-X* can tackle both OLTP and OLAP workloads within a single system. In this section, we introduce *PolarDB-X*'s major designs for Hybrid Transactional and Analytical Processing (HTAP).

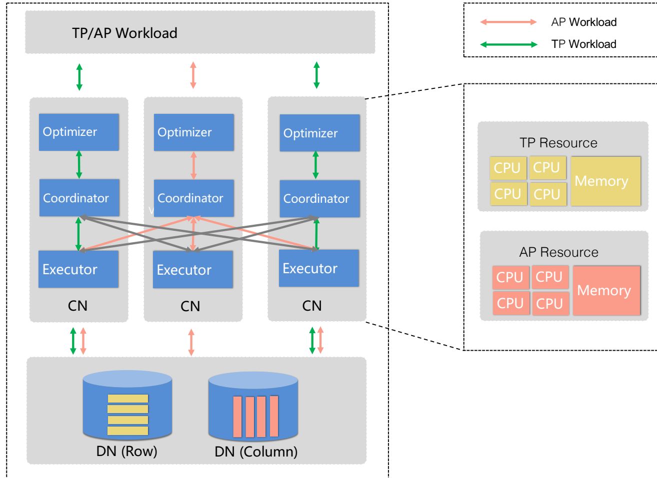


Fig. 6. *PolarDB-X* HTAP framework

A. Overview

Figure 6 shows the overall framework of *PolarDB-X*'s HTAP design. *PolarDB-X* provides a single access endpoint to the application, and all OLTP and OLAP traffics are handled through this endpoint. The HTAP-oriented optimizer at the endpoint automatically differentiates OLTP and OLAP requests and dispatches them to different compute nodes. More specifically, the concept of RW and RO nodes in *PolarDB-X* leads to a natural approach of separating different workloads, i.e., OLTP and OLAP requests are processed by RW and RO nodes, respectively. Such a physical separation of resources retains high stability to OLTP workloads. At the same time, distributed transaction and replica consistency inherently take care of data visibility for OLAP workloads. There are two major advantages from the above HTAP framework. First, different workloads are separated and consistent, where OLTP workloads will not be delayed from log replication. Second, distributed parallel computing techniques can be utilized, where OLAP workloads can obtain good scalability.

B. HTAP Optimizer

Request classification and routing. Existing HTAP solutions are mainly built on top of separated OLAP and OLTP database systems, where read-write separation or ETL pipeline is applied. They suffer from many challenges, such as poor analytical query freshness and high multi-system maintenance overhead. In contrast, *PolarDB-X*'s optimizer is equipped with query classification which is able to route workloads appropriately to help process HTAP workloads within a single system. When a request arrives, the optimizer will first estimate the cost of core resource (e.g., CPU, memory, I/O, network) consumption required by the request. Based on this cost and an empirical threshold, each request is classified as either an OLTP or an OLAP request. Afterwards, all OLTP requests are routed to the primary RW node, while OLAP requests are further fed into a MPP optimization stage that generates distributed execution plans to run in multiple RO nodes. As a result, applications can rely on *PolarDB-X* to handle HTAP workloads transparently.

Operator push-down. The separation of computing and storage brings in excellent scale out capability, but also introduces noticeable network traffic between computing and storage nodes. To mitigate this overhead, pushing compute operation closer to data storage is a promising optimization. Hence, *PolarDB-X* supports operator push-down. We fully consider characteristics of both storage (e.g., row store or column store) and data (e.g., whether the column has an index) to estimate the execution cost. This guides us to push specific portions of the query (such as Join, Agg, Sort operators) to corresponding storage nodes for near-data computing.

C. HTAP Executor

MPP model. *PolarDB-X* processes OLAP queries using a MPP model, which involves many CN nodes accessing data from many DN nodes. The overall procedure is as follows: (1) The user connects to a CN node, which subsequently acts as Query Coordinator. (2) A query is sent to Query Coordinator, who generates a execution plan from its optimizer. The plan is split into multiple fragments (i.e. sub-plans), each of which further contains multiple operators (e.g. Scan, Agg, Join). (3) Task Scheduler in Query Coordinator encapsulates each fragment as a Task, and then schedules all tasks to appropriate CN nodes for execution. (4) Each involved CN node applies for required resources. It constructs the context, starts the execution task, and periodically reports its status to Query Coordinator. (5) Each executed task exchanges necessary data with others. When all tasks complete, partial results are sent back to Query Coordinator, who assembles the final result and returns it to the user. (6) Query Coordinator and all involved CN nodes are cleaned up and all resources are released.

Since *PolarDB-X* shards data to multiple DN nodes, the execution plan considers the underlying data locality. Subplans will be pushed down to corresponding DN nodes when

possible, and the rest of the plan is translated to fragments executed by CN nodes.

Timesharing Scheduler. The generated execution plan has to be carefully scheduled on CN nodes to make better use of resources. A CN node in *PolarDB-X* has two schedulers, namely *Task Scheduler* and *Local Scheduler*. The *Task Scheduler* is responsible for task scheduling among different CN nodes. The *Local Scheduler* is responsible for task scheduling within a CN node. The task scheduling and execution resources are not strongly tied. A scheduled job will not exclusively occupy all resources of corresponding execution thread. To make better use of thread resources, *PolarDB-X* adopts *time-slicing execution* model. Each CN nodes has a thread pool for executing scheduled jobs, where each job queues up to enter the thread pool for execution. When a job is blocked, it will be dropped to a blocking queue and wait to be awakened. There are mainly three reasons that cause a job to be blocked: derived from the operator dependency graph; lacking execution resources (e.g. memory); or waiting for the response from DN node. In addition, we borrow from the time-slicing policy of Linux kernel, where we will suspend a job after it runs long enough (e.g. 500ms) in a single round.

D. Resource isolation

Apart from above queuing and time-slicing mechanisms to facilitate the parallel execution of many in-progress queries, the resource management is also important for concurrently running queries. Especially, when a running query can aggressively consume huge amount of resources (such as CPU and memory), it will seriously affect the progression of other concurrent queries. *PolarDB-X* isolates resources for different workloads in a preemptive manner.

For the CPU resource, We classify its usage into two groups, i.e., AP Group and TP Group, and use `cgroups` for resource isolation. The CPU resource of TP Group is unrestricted, while the resource of AP Group is strictly controlled by `cgroups` (using *cpu.min.cfs_quota* and *cpu.max.cfs_quota*). Moreover, the query tasks are assigned to different thread pools: TP Core Pool, AP Core Pool, and Slow Query AP Core Pool. The latter two pools belong to AP Group with strict CPU restrictions. It is possible that a AP query might have been mistakenly recognized as a TP query, and we should ensure that TP queries are unaffected. Once a query in TP Core Pool has been running for an unexpectedly long time, it will terminate its current time slice and be re-assigned to AP Core Pool for subsequent execution. Similarly, when a query in AP Core Pool runs unexpectedly long, it will be re-assigned to Slow Query Pool, which has a lower share of time slices.

For the memory resource, the heap memory in a CN node is divided into four major regions: *TP Memory* to store temporary data for TP queries; *AP Memory* to store temporary data for AP queries; *Other* to store data structures, temporary objects, metadata, etc.; and *System Reserved* for privileged usage. Meanwhile, both *TP Memory* and *AP Memory* have corresponding maximum and minimum usage limits, and they can preempt each other's resources when needed. More specifically, TP Memory will only release the preempted memory (from AP Memory) until the query completion, while AP Memory must immediately release the preempted memory when TP Memory is requesting for it.

E. In-Memory Column Index

PolarDB-X supports an *in-memory column index* on its DN to benefit from column stores, which can significantly improve the performance over row store when dealing with complex queries. This index is implemented as an in-memory columnar representation of the selected or indexed columns in row store. The logical operations (e.g., insert, update, delete) on the indexed column are captured from the log and converted to the corresponding operations on the index. Since the log will be transmitted to RO nodes, we can build column indexes on selected RO nodes responsible for AP queries. To avoid memory expense from maintaining column indexes on the RW node, it only captures the log but will not materialize column indexes. A record in column index has its `trx_id` being consistent with that in InnoDB. This facilitates the reuse of InnoDB *read_view* to implement a hybrid execution plan on a consistent snapshot of both row and column stores. To further mitigate the maintenance overhead of the column index, its updates can be delayed and batched. In this case, its version lags behind the row store's, and AP queries run on the version of snapshot subject to the column index.

Recall that the optimizer needs to enumerate various feasible execution plans in the CBO phase. Inside *PolarDB-X*'s optimizer, the execution plan is represented as an expression tree in relational algebra. In this case, the storage layer is decoupled and different storage types (i.e., row store and column store) can be leveraged. More specifically, the optimizer converts the logical expression plan into a physical execution plan, by adopting the cost model according to the storage characteristics. For example, in a row store, when the index is hit, the qualified row can be quickly fetched; but when the index is missing or a large amount of data is scanned, the IO cost tends to be quite high. Similarly, in a column store, the storage is more compact where the I/O and computation are more efficient when processing large amounts of data; the execution of certain operations such as filter, join, aggregation becomes much faster.

After a comprehensive comparison of physical execution plans on both row store and column store (i.e., in-memory column index in *PolarDB-X*), the optimizer will finally select the one with the lowest cost. In practice, large data scans and push-down plans with join or aggregation prefer in-memory column index, while point queries choose InnoDB row store.

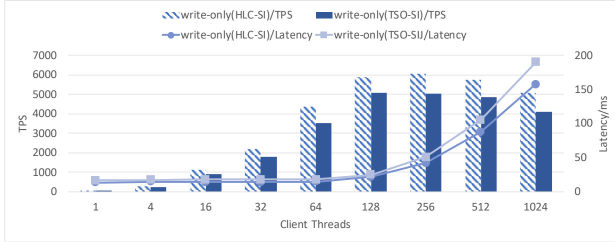
VII. Evaluation

All experiments were conducted on Alibaba Cloud. The CN is deployed on ECS virtual machines, configured with 16-core Intel(R) Xeon(R) Platinum 8269CY CPU @ 2.50GHz and 64GB memory. The DN used is PolarDB with 8-core 64GB memory instance specification (both RW and RO). The Linux kernel version is 4.19.91-19.1.al7.x86_64.

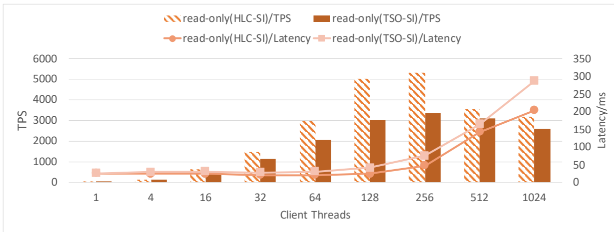
A. Cross-DC Transaction

In order to compare the impact of deploying HLC-SI and TSO-SI in multiple datacenters on transaction (read and write) performance, we used the Sysbench oltp-write-only and oltp-read-only benchmarks. A transaction in oltp-write-only includes deletes, inserts and index updates to different rows. While the transaction in oltp-read-only consists of ten point reads and another four range queries. Data access follows a random distribution, and leads to distributed transactions.

The *PolarDB-X* cluster in this experiment is deployed in three datacenters, and the network round-trip time between datacenters is about one millisecond. It consists of six CN servers and three PolarDB instances. Each datacenter has two CN servers and one PolarDB instance. For TSO-SI deployment, the TSO server is placed at one of the datacenters.



(a) Sysbench Write-Only Transactions



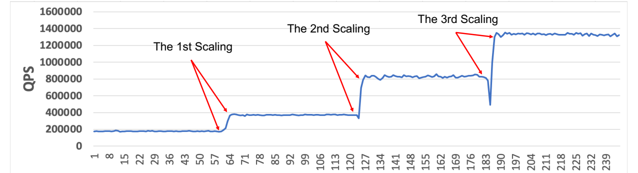
(b) Sysbench Read-Only Transactions

Fig. 7. Comparison of TSO-SI and HLC-SI with Sysbench when deployed across DC

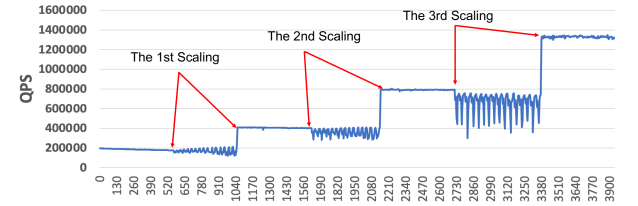
Figure 7(a) and 7(b) compare the total transaction throughput and latency of HLC-SI and TSO-SI under various workloads. In Figure 7(a), we observe that the average transaction latency of TSO-SI is higher than that of HLC-SI, due to the additional delay in obtaining timestamps from the TSO server perhaps in another datacenter. As the number of client threads increases, the gap between them also increases. This is because under higher concurrency, more threads are waiting for responses from TSO, which results in excessive context switches or even thread-thrashing. The peak write throughput of TSO-SI is also 19% higher than that of TSO-SI. A similar situation is also observed in Figure 7(b).

B. Elasticity

This experiment expands the size of a *PolarDB-X* cluster from eight CN nodes and four PolarDB instances gradually to 8x of the original size. We compare the scaling speed of two methods, i.e., one using the rapid tenant migration of PolarDB-MT and the other with traditional data transfer method. The total data volume is 160 million rows and 40 GB in size (about 250 bytes per row), evenly distributed on all shards. During the scaling process, a Sysbench oltp-read-write test (with 3000 clients threads) runs continuously to simulate the background load. Figure 8(a) shows that, when using PolarDB-MT, the three scaling operations (including the warm up time) are completed in 4.2, 4.5 and 4.6 seconds, and the sysbench throughput has increased by 113%, 94% and 68% times, respectively. In contrast, Figure 8(b) shows that it takes 489, 527 and 660 seconds to scale the cluster using data transfer, which is 116-143 times longer than the above approach. Clearly, the rapid scaling based on tenant migration is very helpful to deal with traffic surges.



(a) Scale Using Fast Tenant Migration Mechanism Of PolarDB-MT



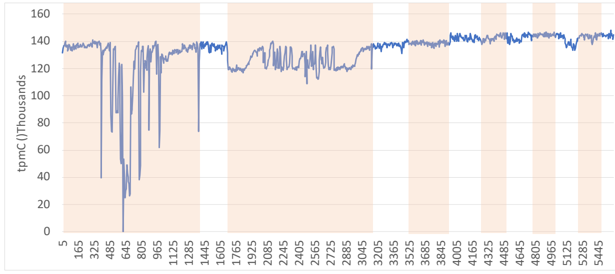
(b) Scale By Copying Data Between Source And Target

Fig. 8. Comparison of Scaling PolarDB-X using different approaches

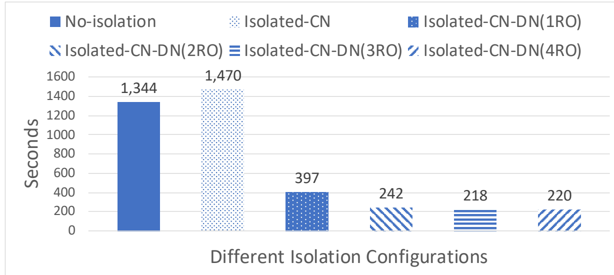
C. HTAP

Resource Isolation and Scalable RO. This experiment aims to evaluate resource isolation in CN server under mixed loads, and the benefits of using dedicated and scalable RO nodes to serve analytical queries. We simulate mixed TP and AP workloads by simultaneously running TPC-C and TPC-H tests on one PolarDB-X cluster. The cluster has two CN servers and four PolarDB DN instances. Each PolarDB DN instance has four RO nodes. TPC-C uses 1000 warehouses and 800 terminals, and the scale factor is 100. The TPC-C test runs continuously in the background, and the load of TPC-C is bound to RW nodes. When TPC-H is not running, tpmC is stable at 130K-140K. We run TPC-H six times with a different configuration each time. In the first configuration, the resource isolation switch of CN is turned off, and it is turned on in all the following configurations. In the first two configurations, the load of TPC-H is sent to the RW node, and TPC-C and TPC-H share CN and DN resources. While in the last four configurations, we use one to four dedicated RO nodes respectively, and reroute the reads in TPC-H to them.

Figure 9(a) shows that in the first configuration, TP workloads are severely disturbed. There are ten obvious performance degradation jitters (over 40%). When resources are nearly exhausted due to contention, the lowest tpmC drops to 57. In the second configuration, the resource isolation switch of CN is turned on, where the interference to TP workload is reduced. TpmC is higher than 120K for most of the time, with only two jitters down to 110K. It can also be seen from Figure 9(b) that the second run of TPC-H is slightly longer



(a) Performance variation of TPC-C while TPC-H runs six times



(b) Latency of each run of TPC-H

Fig. 9. TPC-C and TPC-H Performance when running mixed workloads with different resource isolation and available resources configurations

(9%) than the first run. This is because the resource isolation is used and AP workload takes up less resources.

In the last four configurations, after using dedicated RO nodes to serve read requests, TPC-C is almost unaffected. The last four groups show the effect of dedicated and scalable RO nodes on analytical workloads. It can be seen from Figure 9(b) that increasing the RO node from one to two has a significant effect, reducing the latency by 39%, and further expanding from two to three nodes reduces by another 10%. However, the continued increase of RO nodes has almost no effect, which indicates that the bottleneck of the system at this moment lies in the CN and backend row store (based on InnoDB).

MPP and Column-Index. This test aims to show the impact of the MPP execution engine and the in-memory column index on the TPC-H load. MPP uses four CN servers. Figure 10 shows that after using MPP, almost all queries are greatly improved, and 21 of them are improved by more than 100%. Among them, Q9 has the highest improvement ratio, reaching 263%, which is close to linear because this query involves operations of 6 tables and few filter conditions. A large number of Hash Join and Hash Aggregation are computed in CN, which has become a bottleneck. The ratios of Q11 and Q15 are relatively low, 49% and 79% respectively, because the amount of data involved in these two queries is relatively small where the join algorithm chooses to use index nested loop join. In this case, the CPU of CN is not a bottleneck.

Using column index, the latency of seven queries have been significantly reduced: Q1 (748%), Q6 (1828%), Q8 (243%), Q12 (556%), Q14 (547%), Q15 (463%), Q21 (348%). In Q14, using column index reduces a lot of scan time. For Q1 and Q6, table-scan and filter operators of the lineitem table and the first phase of aggregation are offloaded. For Q12, the partition-wise join of the lineitem and orders is pushed down to the column index, which uses in-memory hash join to speed up. Q21 also uses the built-in hash join of column index to improve performance. Q15 is an example of the combined use of row storage and column index, in which the table-scan and filter of the lineitem are pushed down to the column index, and the primary key of the supplier (row storage) table is looked up in the index nested loop join. Q8 pushes the bloom filter down to the column index, effectively reducing the amount of data transmission between CN and DN.

VIII. Lessons Learned

We have also learned from experiences in providing *PolarDB-X* as a service on Alibaba Cloud and added a series of DBA- and developer-friendly features.

Anti-Hotspots. In mission critical applications, hotspots in the database must be carefully dealt with. The most common case is that the load between DN nodes is unbalanced where some are overloaded. We can migrate shards to achieve a balanced state between DNs. If the data volume or traffic of a single shard is too large, it will become a *hot shard*. When a shard grows larger due to data skew, we will split the shard according to another hash function. Some secondary index keys will become *hot keys*, such as using states as keys. Some states will have a lot of data and become hot keys. The hot key can be placed on one shard alone. If hotspot still exists, more fields can be added to the key of the secondary index to split a hotspot key into multiple keys with the same prefix. Rows that are frequently updated and accessed are called *hot rows*. All operations on a single hot row is actually executed serially, and can be optimized using hotspot-aware in-memory data structures, such as the work in [32], [33].

Automated Traffic Control. When concurrency of certain unusual SQLs increase, these queries will occupy significant system resources and compete for resources with high-priority queries of the core business. For example, the occurrence of cache penetration/breakdown would dramatically increase the concurrency of certain types of SQL. And when a large number of slow SQLs without proper indexes are issued, they will exhaust system resources and disturb normal business. *PolarDB-X* is assisted with machine learning techniques to perform offline training on historical performance data, and uses obtained model to perform anomaly detection on real-time telemetry data. When an anomaly is detected, *PolarDB-X* performs an analysis of running transactions and locks waiting, finds the problematic queries that consume the most resources, and then limits the maximum allowable concurrency of them.

Index Recommendation. Indexes are critical to the performance of SQL queries. However, indexes will also introduce significant update costs, which is true especially in a distributed database, where adding indexes will increase the number of participants in two-phase commit. If the database can suggest indexes for SQL and give quantitative analysis of performance improvement after using the new index, a lot of efforts will be saved. Hence, we introduce a *SQL Advisor* in *PolarDB-X*. This advisor can analyze the SQL to find which columns can use the index (Indexable Column), enumerate the possible index combinations to get the Candidate Index, prune

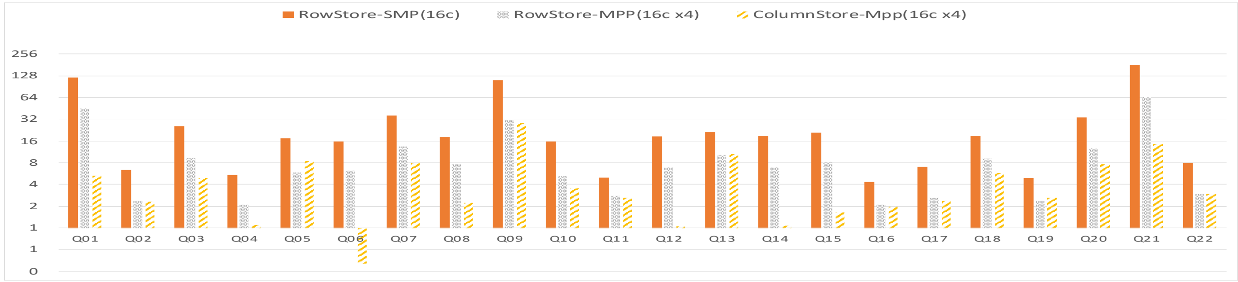


Fig. 10. The effect of vectorized engine, MPP engine and in-memory column index on TPC-H workload

some candidates with low selectivity through heuristic search, use the optimizer to estimate costs with these hypothetical (what-if [34]) indexes, select the index combination with the highest saving and recommend it to the user.

IX. Related Works

Cloud-native databases like Aurora [14], Socrates [25] and Taurus [35] build database services on top of a shared storage pool. Notably, Aurora offloads page materialization downwards to the shared storage. Socrates transforms the on-premise SQL Server into a DBaaS and further separates logging and storage from database kernel to dedicated services. Decoupled storage systems has a significant advantage of elastic scaling-up/down. However, none of these systems have been reported to treat shards as tenants and support seamlessly horizontal scaling-out. *PolarDB-X* makes up for this aspect.

The industry has developed many distributed database systems from scratch based on transactional key-value stores [3], [1], [2]. Spanner [3] uses strict two-phase locking (2PL) and two-phase commit to ensure serializability. The adoption of Truetime API allows Spanner to support linearizability in a geo-distributed cluster. However, Truetime API requires a few milliseconds commit-wait time to avoid uncertainty caused by clock drifting. This increases the latency of read-write transactions in Spanner. *PolarDB-X* supports lower isolation levels and performs better for short-transaction OLTP workloads. CockroachDB [2] provides serializable isolation using Multi-version timestamp ordering. Its timestamp is generated using HLC. Our approach is different since *PolarDB-X* chooses snapshot isolation. TiDB [1] uses a centralized TSO to provide MVCC based snapshot isolation. In contrast, our system adopts a decentralized timestamp service.

The academia has also addressed performance bottlenecks in distributed database systems, such as distributed transaction, concurrency control protocol, and global clock service [36], [37], [31], [38], [39], [40], [41], [42], [43]. Several work [44], [37], [31] adopt local clock to remove the overhead of consulting a centralized clock service. Their approaches depend on optimistic transaction execution or wait delay before starting a transaction due to local clock's time skew. DST [45] designs a distributed time service that is orthogonal to concurrency control protocols. This approach needs to record read and write sets to support serializable isolation for both read-write transactions and read-only transactions.

Live migration of both shared-storage and shared-nothing databases have been well studied. In Albatross [46], similar to PolarDB-MT, the persistent data of a tenant's database is stored in the shared storage and hence does not need migration. In addition, it migrates the database cache and the state of active transactions to minimize service disruption. For shared-nothing architecture, Zephyr [47] introduces a dual mode that allows both source and destination to execute transactions simultaneously to mitigate service interruption, but it still needs to transfer database pages to the target node. To the best of our knowledge, no previous studies have combined the live migration techniques from both shared-storage and shared-nothing architectures. In future work, *PolarDB-X* will explore similar methods in [46], [47], [48], [49], [50] to further shorten the transaction pause time during migration.

Azure SQL has thorougly investiaged resource isolation in a multi-tenant database system and introduces the SQLVM abstraction [51], [52], [53], which adopts lightweight metering to audit whether the dynamic allocation of resources for tenants achieves the same performance goal as the resource reservation. *PolarDB-X* adopts a similar abstraction in the HTAP scenario to avoid the mutual interference between TP and AP workloads, focusing on dynamic allocation of CPU, memory and network traffic resources. In addition, it also leverages dynamically provisioned cloud resources to allow AP workload to run on physically independent resources.

X. Conclusion

In this paper, we describe the design of *PolarDB-X*, a distributed database co-designed with cloud-native databases. It follows a three-tier CN-DN-SN architecture. At the CN layer, *PolarDB-X* uses *HLC-SI* to avoid coordinating with the central timestamp service to determine the order and visibility between transactions. It also supports HTAP loads through resource isolation and MPP execution engine. At the DN layer, it uses Paxos to replicate redo logs for cross-datacenter deployment. It can quickly increase read throughput through extending RO nodes on top of decoupled storage, rapidly scale a cluster by fast migrating shards through the multi-tenant live migration, and supply the in-memory column index to accelerate AP queries. At the SN layer, we achieve data persistence in a single datacenter. The novel layered architecture of *PolarDB-X* makes it meet the goals of efficient cross-datacenter transaction, rapid scaling-out, and powerful HTAP support in the cloud-native era.

REFERENCES

[1] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang *et al.*, "Tidb: a raft-based htap database," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3072–3084, 2020.

[2] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss *et al.*, "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1493–1509.

[3] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, D. Woodford, Y. Saito, C. Taylor, M. Szymaniak, and R. Wang, "Spanner: Google's globally-distributed database," in *OSDI*, 2012.

[4] D. G. Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh, "Omid: Lock-free transactional support for distributed data stores," in *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, 2014, pp. 676–687.

[5] Apache, "Shardingsphere," https://shardingsphere.apache.org/.

[6] Vitess, "Vitess," https://vitess.io//.

[7] F. Li, "Cloud-native database systems at alibaba: Opportunities and challenges," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2263–2272, 2019.

[8] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li *et al.*, "{TAO}: Facebook's distributed data store for the social graph," in *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, 2013, pp. 49–60.

[9] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan, "Salt: Combining ACID and BASE in a distributed database," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 495–509. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/xie

[10] citusdata, "Citusdb," https://www.citusdata.com/.

[11] Postgres-XC, "Postgres-xc," https://postgresxc.fandom.com/wiki/Postgres-XC_Wiki.

[12] Postgres-XL, "Postgres-xl," https://www.postgres-xl.org/.

[13] P. Chairunnanda, K. Daudjee, and M. T. Özsu, "Confluxdb: Multi-master replication for partitioned snapshot isolation databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 947–958, 2014.

[14] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1041–1052.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[16] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, 2007, pp. 398–407.

[17] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," 2011.

[18] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[19] M. Stonebraker and U. Çetintemel, "" one size fits all" an idea whose time has come and gone," in *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, 2018, pp. 441–462.

[20] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma, "Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1849–1862, 2018.

[21] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan *et al.*, "{POLARDB} meets computational storage: Efficiently support analytical workloads in cloud-native relational database," in *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, 2020, pp. 29–41.

[22] W. Cao, Y. Zhang, X. Yang, F. Li, S. Wang, Q. Hu, X. Cheng, Z. Chen, Z. Liu, J. Fang, B. Wang, Y. Wang, H. Sun, Z. Yang, Z. Cheng, S. Chen, J. Wu, W. Hu, J. Zhao, Y. Gao, S. Cai, Y. Zhang, and J. Tong, "Polardb serverless: A cloud native database for disaggregated data centers," in *ACM SIGMOD*, 2021, pp. 2477–2489.

[23] M. Demirbas, M. Leone, B. Avva, D. Madeppa, and S. Kulkarni, "Logical physical clocks and consistent snapshots in globally distributed databases," The State University of New York at Buffalo, Department of Computer Science and Engineering, Tech. Rep., 2014.

[24] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[25] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash *et al.*, "Socrates: the new sql server in the cloud," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1743–1756.

[26] A. Depoutovitch, C. Chen, J. Chen, P. Larson, S. Lin, J. Ng, W. Cui, Q. Liu, W. Huang, Y. Xiao *et al.*, "Taurus database: How to be fast, available, and frugal in the cloud," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1463–1478.

[27] AlibabaCloud, "Loadbalancer," https://www.alibabacloud.com/product/server-load-balancer.

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

[29] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner *et al.*, "F1: A distributed sql database that scales," 2013.

[30] K. Krikellas, S. Elnikety, Z. Vagena, and O. Hodson, "Strongly consistent replication for a bargain," in *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, 2010, pp. 52–63.

[31] J. Du, S. Elnikety, and W. Zwaenepoel, "Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks," in *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013*. IEEE Computer Society, 2013, pp. 173–184. [Online]. Available: https://doi.org/10.1109/SRDS.2013.26

[32] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li, "Hotring: A hotspot-aware in-memory key-value store," in *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, 2020, pp. 239–252.

[33] H. Jin, Z. Li, H. Liu, X. Liao, and Y. Zhang, "Hotspot-aware hybrid memory management for in-memory key-value stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 779–792, 2019.

[34] S. Chaudhuri and V. Narasayya, "Autoadmin "what-if" index analysis utility," *ACM SIGMOD Record*, vol. 27, no. 2, pp. 367–378, 1998.

[35] A. Depoutovitch, C. Chen, J. Chen, P. Larson, S. Lin, J. Ng, W. Cui, Q. Liu, W. Huang, Y. Xiao *et al.*, "Taurus database: How to be fast, available, and frugal in the cloud," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1463–1478.

[36] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *SIGMOD*, 2012.

[37] X. Yu, Y. Xia, A. Pavlo, D. Sanchez, L. Rudolph, and S. Devadas, "Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System," in *Proceedings of the VLDB Endowment (PVLDB)*, June 2018.

[38] J. Cowling and B. H. Liskov, "Granola: Low-overhead distributed transaction coordination," in *Proc. of USENIX ATC. 2012*, 2010, pp. 223–236.

[39] E. Zamanian, C. Binnig, T. Kraska, and T. Harris, "The end of a myth: Distributed transaction can scale," *PVLDB*, vol. 10, no. 6, pp. 685–696, 2017.

[40] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building consistent transactions with inconsistent replication," in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 263–278. [Online]. Available: https://doi.org/10.1145/2815400.2815404

[41] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. E. Abbadi, "Maat: Effective and scalable coordination of distributed transactions

in the cloud," *Proc. VLDB Endow.*, vol. 7, no. 5, pp. 329–340, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p329-mahmoud.pdf

[42] X. Yu, H. Liu, E. Zou, and S. Devadas, "Tardis 2.0: Optimized time traveling coherence for relaxed consistency models," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*, A. Zaks, B. Mendelson, L. Rauchwerger, and W. W. Hwu, Eds. ACM, 2016, pp. 261–274. [Online]. Available: https://doi.org/10.1145/2967938.2967942

[43] T. Zhu, Z. Zhao, F. Li, W. Qian, A. Zhou, D. Xie, R. Stutsman, H. Li, and H. Hu, "Solardb: Toward a shared-everything database on distributed log-structured storage," *ACM Trans. Storage*, vol. 15, no. 2, pp. 11:1–11:26, 2019. [Online]. Available: https://doi.org/10.1145/3318158

[44] X. Yu, A. Pavlo, D. Sánchez, and S. Devadas, "Tictoc: Time traveling optimistic concurrency control," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 1629–1642. [Online]. Available: https://doi.org/10.1145/2882903.2882935

[45] "Unifying timestamp with transaction ordering for MVCC with decentralized scalar timestamp," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. Boston, MA: USENIX Association, Apr. 2021. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/wei

[46] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration," *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 494–505, 2011.

[47] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: live migration in shared nothing databases for elastic cloud platforms," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 301–312.

[48] S. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, and P. Shenoy, "" cut me some slack" latency-aware live migration for databases," in *Proceedings of the 15th international conference on extending database technology*, 2012, pp. 432–443.

[49] T. Mishima and Y. Fujiwara, "Madeus: database live migration middleware under heavy workloads for cloud environment," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 315–329.

[50] J. Hai, C. Wang, X. Chen, T. O. Li, H. Cui, and S. Wang, "Fulva: Efficient live migration for in-memory key-value stores with zero downtime," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2019, pp. 83–8309.

[51] V. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri, "Sqlvm: Performance isolation in multi-tenant relational database-as-a-service," 2013.

[52] S. Das, V. R. Narasayya, F. Li, and M. Syamala, "Cpu sharing techniques for performance isolation in multi-tenant relational database-as-a-service," *Proceedings of the VLDB Endowment*, vol. 7, no. 1, pp. 37–48, 2013.

[53] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri, "Sharing buffer pool memory in multi-tenant relational database-as-a-service," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 726–737, 2015.