PolarDB-SCC: A Cloud-Native Database Ensuring Low Latency for Strongly Consistent Reads

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

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

Hao Chen
Alibaba Group
ch341982@alibaba-inc.com

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

Feifei Li
Alibaba Group
lifeifei@alibaba-inc.com

Wenchao Zhou
Alibaba Group
zwc231487@alibaba-inc.com

ABSTRACT

A classic design of cloud-native databases adopts an architecture that consists of one read/write (RW) node and one or more read-only (RO) nodes. In such a design, the propagation of write-ahead logs (WALs) from the RW node to the RO node(s) is typically performed asynchronously. Consequently, system designers either have to accept a loose consistency guarantee, where a read from the RO node may return stale data, or tolerate significant performance degradation in terms of read latency, as it then needs to wait for the log to be propagated and applied. Most commercial cloud-native databases, such as Amazon Aurora, choose performance over strong consistency. As a result, it makes RO nodes useless for many applications requiring read-after-write consistency (a form of strong consistency), and the support for serverless databases (i.e., allowing the RO nodes to be scaled out automatically) is impossible as they require a single endpoint.

This paper proposes *PolarDB-SCC* (PolarDB-Strongly Consistent Cluster), a cloud-native database architecture that guarantees strongly consistent reads with very low latency. The core idea is to eliminate unnecessary waits and reduce the necessary wait time on RO nodes while still supporting strong consistency. To achieve this, it tracks the RW node's modification timestamp at three progressively finer-grained levels. We further design a Linear Lamport timestamp to reduce the RO node's timestamp fetching operations and leverage the RDMA network for all the data transferring (*e.g.*, timestamp fetching and log shipment) to minimize network overhead and extra CPU usage. Our evaluation shows that PolarDB-SCC does not incur any noticeable overhead for ensuring strongly consistent reads compared with the eventually consistent (stale) read policy. To the best of our knowledge, PolarDB-SCC is the first "read-write splitting" cloud-native database that supports strongly consistent read with negligible overhead. Compared with a straightforward read-wait design, PolarDB-SCC improves throughput by up to 4.51× and reduces median latency by up to 3.66× in SysBench's read-write workload. PolarDB-SCC is already commercially available at Alibaba Cloud.

PVLDB Reference Format:
Xinjun Yang, Yingqiang Zhang, Hao Chen, Chuan Sun, Feifei Li, and Wenchao Zhou. PolarDB-SCC: A Cloud-Native Database Ensuring Low Latency for Strongly Consistent Reads. PVLDB, 16(12): 3754 - 3767, 2023.
doi:10.14778/3611540.3611562

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611562

1 INTRODUCTION

Cloud-native databases are becoming a critical infrastructure supporting the migration of data storage and processing tasks to the cloud environment. Notable examples include Aurora [55], Hyperscale [14, 30], Socrates [6] and PolarDB [32]. Many of them adopt a disaggregated shared storage architecture and usually consist of one primary (RW) node to process the read/write requests and one or more secondary (RO) nodes to handle read requests.

To keep an RO node's buffered data up-to-date, the RW node generates the corresponding log for each update and ships the log to RO nodes. RO nodes apply the log to update their buffered data. Since the log application process is asynchronous, an RO node may be unable to return the latest updates that have already taken place on the RW node and consequently may return "stale" data. Many cloud-native databases claim that RO nodes could improve read performance. However, for the reason outlined above, the service on an RO node can only serve applications that does not require read-after-write consistency. Table 1 shows some databases' [1] stale read ratios (representing how many read requests get the stale data) on the RO node. QueryFresh [56] is a recent research work that targets minimizing staleness on RO nodes. In this test, we first update a record on the RW node, then read it from the RO node after Δt. It shows that only PolarDB-SCC can completely avoid the stale read. In contrast, in the other databases, applications that require strongly consistent reads have to send read requests to the RW node, presenting a severe limitation.

Table 1: Ratios of stale reads in different databases

Δt	DB-A	DB-B	PolarDB	QueryFresh	PolarDB-SCC
1ms	99.8%	99.9%	97.8%	100%	0
7ms	17.1%	89.0%	0.3%	59.9%	0

The strongly consistent read (i.e., a read request always sees the latest committed updates that happen before it, aka *the strict consistency model* [58]) is an essential need in many applications [2]. For example, in Alibaba's e-commerce applications, if it cannot guarantee strong consistency, the customer who has placed an order may soon finds that the order does not exist or is shown to be unpaid after payment. Such need for strongly consistent reads also

[1] Due to the restriction of DeWitt Clause, we anonymize the commercial systems. DB-A and DB-B are two databases from two top cloud providers.

appear in scenarios where databases are used to support interactions among microservices [28]. These microservices usually share the same databases and have some dependencies at the application level. It relies the database to provide strong consistency to ensure the interactions proceed as expected. At Alibaba Cloud, we also receive many requests from users to provide such strongly consistent reads, such as insurance companies and financial institutes.

In many of today's cloud-native databases, to support strong consistency, applications have to send all read requests to the RW node. Consequently, they cannot improve the read throughput by adding more RO nodes and the RW node could quickly become the bottleneck. This dramatically limits the system's ability to process read-dominant workloads. In practice, many applications are read-dominated [9, 12, 51], including Alibaba's trading service workload (50% reads) and inventory management service workload (90% reads). On the other hand, cloud service providers have recently evolved from provisioned to serverless [4, 18, 26, 34, 39, 44]. To support the RO node's auto-scaling-out, a unified endpoint is required for users. The strongly consistent read must be guaranteed by this endpoint to ensure that the writes on this endpoint must be immediately visible to the following reads. Therefore, it's imperative to have a new system design to ensure strongly consistent reads on RO nodes in a cloud-native database cluster to improve system performance and make the system really scalable.

There are two straightforward designs to support consistent reads on RO nodes: *commit-wait* and *read-wait*. In the *commit-wait* approach, the RW node has to wait for the related logs to be applied to all RO nodes before committing a transaction. The *read-wait* requires an RO node to get the RW node's current timestamp when a request arrives and wait for the logs that happened before that timestamp to be applied before processing this request. These two naive solutions induce expensive wait time and sacrifice performance significantly. Some systems do provide these options for ensuring a strongly consistent read, but they are not recommended due to their poor performance. Table 2 shows the performance drop when enabling the strongly consistent reads in different commercial databases [2]. The detailed configurations are shown in Section 5.1. PolarDB-SCC only increases the RO node's query latency by 3.8%, while other systems increase the latency by 1.26×-51.28×. Some works also try to speed up the log shipment with the fast network (*e.g.*, RDMA) or design a faster log application on RO nodes, *e.g.*, Query Fresh [56]. They aim to minimize the data's staleness, but still face the problem that not all the requests on RO nodes can get the latest updates, only guaranteeing eventual consistency.

Table 2: RO node's latency increment when enabling strongly consistent reads on RO nodes

Databases	MGR [37]	PolarDB	DB-C	PolarDB-SCC
Latency increment	5128%	126%	553%	3.8%

Many existing systems depend on the streaming log shipment and linearly apply logs on RO nodes. An RO node can not perform a strongly consistent read before the logs are applied to a specific timestamp, even if the requested data is not changed in those logs. In these conventional schemes, the RO node is unaware if a certain page is changed, causing unnecessary waiting.

[2]DB-C is a database from a famous startup.

Targeting these limitations, this paper proposes PolarDB-SCC (PolarDB-Strongly Consistent Cluster), a cloud-native database that guarantees low latency for strongly consistent reads on RO nodes. PolarDB-SCC is designed based on the read-wait policy while eliminating the aforementioned overheads. To minimize an RO node's wait time on log applications, we proposes a hierarchical modification tracker, which tracks the RW node's modification at global, table, and page levels. Doing this allows an RO node to wait for the log application at a fine-grained level (e.g., page level) when the global timestamp is unsatisfied. This avoids having to wait for the entirety of its in-memory data to be up-to-date. Furthermore, we propose a Linear Lamport timestamp on the RO node to avoid frequent timestamp fetching from the RW node. This significantly reduces the network and communication overhead. We then leverage a fast RDMA network (a commonly available infrastructure for many cloud vendors' data centers) for log shipment and timestamp fetching, which eliminates the RW node's CPU overhead for handling timestamp fetching requests and transferring logs.

Based on these designs, PolarDB-SCC achieves low-latency and strongly consistent reads on RO nodes, enabling a *truly scalable and elastic* "read-write splitting" cloud-native database cluster that *ensures strongly consistent reads on RO nodes*. PolarDB-SCC can provide one endpoint (e.g., via a proxy) with strong consistency guarantees for applications, and distribute read requests to its RO nodes in a load-balancing manner. This brings much better elasticity, scalability, and on-demand usage, as it allows the system to dynamically increase/decrease the number of RO nodes according to application load in a transparent fashion. On the other hand, PolarDB-SCC neither changes the database's internal data structures nor relies on certain databases' internal data structures, making it easy to be integrated with other databases and optimizations.

We summarize our main contributions as follows:

- We propose a hierarchical modification tracker, which tracks the RW node's modification timestamp at three levels (global, table and page levels). This enables an RO node to wait for log applications at a fine-grained level when the global timestamp is unsatisfied, minimizing the wait time overhead.
- We design a Linear Lamport timestamp on RO nodes, which avoids frequent timestamp fetching operations initiated by RO nodes and reduces the network overhead.
- We further leverage the fast RDMA network to guarantee low latency for ensuring strongly consistent reads on RO nodes.
- We thoroughly evaluate PolarDB-SCC with different workloads. PolarDB-SCC does not incur any noticeable overhead compared with one using the stale read policy.

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

2 BACKGROUND AND MOTIVATION

2.1 Cloud-native relational databases

Increasingly more on-premise databases are being migrated to the cloud for reasons such as high availability, elasticity, and lower costs. There have been a number of cloud-native databases designed

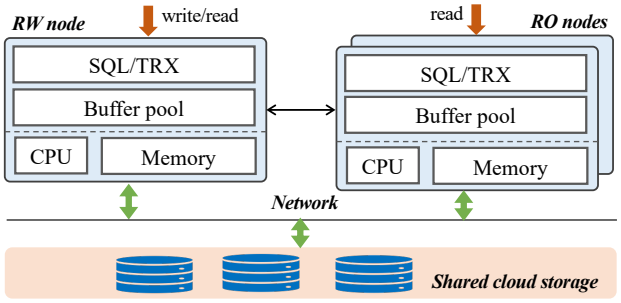

Figure 1: The architecture of a typical cloud-native relational database

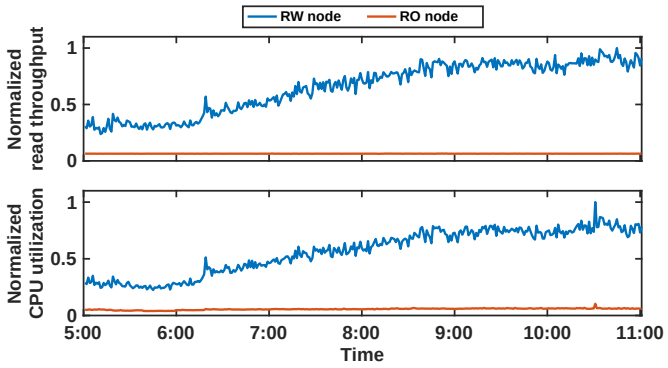

Figure 2: Read throughput and CPU utilization on RW and RO nodes

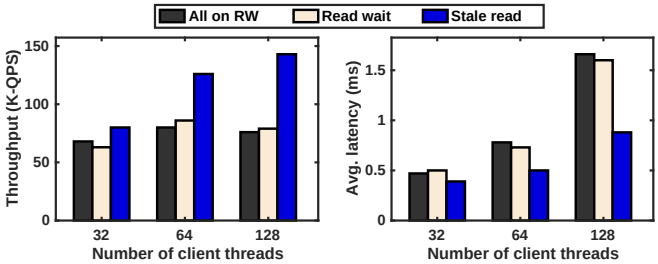

Figure 3: Performance comparison of different read policies

by different cloud providers, *e.g.*, AWS Aurora [55], Azure Hyperscale [14, 30], Azure Socrates [6] and Alibaba PolarDB [32]. They often have compatible APIs with conventional relational databases, but target to fully utilize the underlying cloud resources and infrastructure to provide better performance, elasticity, scalability, and availability [11, 32, 47]. Cloud-native databases play an essential role in connecting the underlying resources (IaaS) to various applications (SaaS), making it a critical system for the cloud [32].

Figure 1 illustrates the typical architecture of a cloud-native relational database. It usually consists of one primary (RW) node and one or more secondary (RO) nodes, in which the RW node serves both read and write requests while the RO nodes only process read requests. Each node is a monolithic database instance with conventional components such as SQL parsing, transaction processing, buffer pool, etc. However, unlike the conventional monolithic databases, the RW and RO nodes share disaggregated cloud storage that promises fault-tolerance and consistency. Thus, adding more RO nodes will not require extra storage overhead. That is different from the conventional primary/secondary database cluster, in which each node has its own storage. In a cloud-native database, the RO node can respond to read requests to increase performance, and provide high availability by promoting one of the RO nodes to be the new RW node when the existing RW node fails. Some cloud-native relational databases often deploy a proxy node on top of the RW and RO nodes to achieve load balancing, handle failover, provide access control, and other functionality.

All database systems buffer data in the memory to improve performance. To avoid losing buffered data, they usually generates the redo log (which records the data changes) for each update and synchronizes the redo log to its persistent storage before committing the transactions. In a cloud-native database, to keep the buffered data up-to-date, RO nodes read the redo logs from the disaggregated shared cloud storage and apply them to their local buffer.

2.2 Limitations on RO nodes

Stale read. As illustrated above, an RO node asynchronously reads redo logs from the cloud storage (or receives redo logs from the RW node via the network), and applies them to keep its buffered data up-to-date. Due to the network delay, storage I/O, and CPU overhead, the log application process on RO nodes may be unable to keep up with the RW node's foreground transaction processing. As a result, the RO node's buffered data may fall behind the RW node's data updates. In this case, a read request on an RO node can not read the latest updates that happen before it. For example, if a transaction successfully updates a value on the RW node, and reads it on an RO node slightly later. It may find the stale value instead of the latest version. Existing cloud-native databases (*e.g.*, Aurora [55], Azure SQL [6], and PolarDB [32]) do not provide strongly consistent reads on RO nodes or the latency of strongly consistent reads is too high to be accepted by applications [5, 35].

Underutilized RO nodes. As a result, in practice, when strong consistency is required, users usually send all such read requests to the RW node. In this scenario, RO nodes are only used for handling the failover but not for scalability, and the resources on RO nodes are not fully utilized. To illustrate this case, we sample the CPU utilization and read throughput of the RW and RO nodes from a standard PolarDB instance at Alibaba, shown in Figure 2. Because the application in this case can not tolerate stale reads, it has to send all its read requests to the RW node for strongly consistent reads. Therefore, the RW node's throughput and CPU utilization are up to 15.61× and 14.40× higher than the RO node (taking the average). Due to the lack of ensuring strong consistency, the RO nodes are idle most of the time, making them underutilized.

Overhead of ensuring strongly consistent read. There are two straightforward ways to achieve strongly consistent reads on the RO nodes: *commit-wait* and *read-wait*. The *commit-wait* solution forces the RW node to wait until the corresponding redo logs have been applied to all RO nodes before committing the transaction. This design induces considerable wait overhead to the critical path of transactions with writes, significantly decreasing the write performance. In the *read-wait* scheme, before an RO node processes a read request, it fetches the RW node's current timestamp and waits for its local applied timestamp to precede it. Compared to the

commit-wait scheme, the read-wait scheme induces less negative impact on processing write requests on the RW node. But it still has some limitations: First, it will slow down the RW node if there are many requests reading timestamps from the RW node. Additionally, the RO node always waits for all the in-memory data to be up-to-date, regardless how much data it will access. To demonstrate the overheads induced by *read-wait*, we tested Alibaba Cloud's cloud-native database PolarDB with three configurations: *all on RW* (all requests are handled on the RW node), *read-wait* (guarantee strongly consistent read via read-wait design), and *stale-read* (RO nodes return the stale read). Figure 3 shows their performance in Sysbench's read-write workloads (a detailed setup is shown in Section 5.1). As expected, the stale-read policy has the best performance (both throughput and latency) because RO nodes could help handle read requests without extra overhead, but they fail to satisfy strong consistency as required by many applications. Processing all requests (read and write) on the RW node decreases throughput by up to 48.85% and increases latency by up to 88.64% compared with the stale-read policy. The read-wait configuration provides only slight improvement over the "all on RW" configuration and its performance even drops slightly under light workloads because of its heavy overhead. Each read request on the RO node will initiate a timestamp fetching request on the RW node in the read-wait design. This will slow down the RW node's performance significantly, especially when the RW node has a heavy load. Furthermore, RO nodes have to wait for log transfer and application after fetching the timestamp, making good performance a difficulty.

2.3 Opportunities with RDMA

In the read-wait design, the TCP/IP-based log shipment and timestamp fetching suffer from high network latency and high CPU usage. Fortunately, as the networks become faster and technologies such as RDMA [15, 63] are readily available at commodity clusters, the remote memory access latency is inching closer to the native DRAM latency [1]. On the other hand, the one-sided RDMA interface does not require involving the server's CPU when remotely reading the server's data. The low network latency and CPU overhead features of RDMA motivate us to track the RW node's timestamp at finer-grained levels and also speed up log shipment and timestamp fetching.

2.4 Stale reads on other database systems

Having stale reads on RO nodes is not the specific problem in cloud-native databases (with disaggregated shared cloud storage). It also exists in the other databases, *e.g.*, the primary-secondary and shared-nothing distributed clusters. MGR cluster [37] is one of the most popular primary-secondary database clusters [3]. It uses the statement log to synchronize the changes from the RW (primary) node to the RO (secondary) nodes. But RO nodes apply the log asynchronously, and could fall behind the RW nodes leading to stale reads. The ProxySQL [45], a high-performance MySQL proxy, does not fix this problem. It may also return stale data when reading from secondary nodes [46]. TiDB [22] also suggests responding to read requests on leader nodes because ensuring strongly consistent reads on follower nodes would have to wait for log application, incurring significant overhead. Amazon DynamoDB [49] also requires performing all of its strongly consistent reads and writes in the same region. Otherwise, it only supports eventual consistency [7]. CockroachDB [54] also claims that the global strong consistent read will cause high latency. As a tradeoff, CockroachDB [54] and Spanner [12] both provide the option of bounded-staleness. However, it cannot completely avoid stale reads.

[3] MGR also supports the multi-primary mode, but the primary-secondary mode is more commonly used.

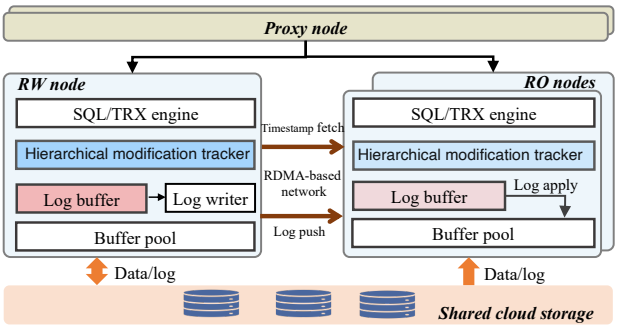

Figure 4: The architecture of PolarDB-SCC

3 POLARDB-SCC OVERVIEW

Design rationale. PolarDB-SCC aims to provide a low-latency, strongly consistent cloud-native database cluster, in which the RO nodes could always return the latest updates that are committed ahead of the request's/transaction's start timestamp. This enables the system to distribute read requests to the RO nodes and split the read/write requests while ensuring strong consistency. As a result, the cluster can provide a unified, strongly consistent endpoint for applications (*e.g.*, via a proxy), and adjust the number of RO nodes on-demand elastically. Resource utilization on RO nodes is also improved, rather than deploying RO nodes only for handling failover. The main challenge is to keep the in-memory data consistent between the RW and RO nodes while ensuring low latency. The key idea of PolarDB-SCC is to eliminate unnecessary waits and reduce the necessary wait time on the RO node.

Architecture overview. Figure 4 shows the architecture of PolarDB-SCC, which consists of one primary (RW) node, one or more secondary (RO) nodes, and a proxy node on top of the RW and RO nodes. The RW node could serve both read and write requests, while RO nodes only respond to read requests. The proxy node could provide transparent load balancing by separating read and write traffic via distributing read requests to RO nodes and forwarding write requests to the RW node. It also provides high availability by promoting one of the RO nodes to the new RW node if the current RW node fails. The proxy node has multiple (usually 2) members for high-availability and high-performance. The RW and RO nodes share a disaggregated cloud storage, similar to many cloud-native databases. The RW and RO nodes are connected by an RDMA-based network for fast log shipment and timestamp fetching.

The core components of PolarDB-SCC are the hierarchical modification tracker, Linear Lamport timestamp, and the RDMA-based log shipment protocol. The hierarchical modification tracker maintains the RW node's modification at three levels: The global level maintains the whole database's latest modification timestamp and

table/page levels record some tables'/pages' newest modification timestamps. To perform a strongly consistent read on the RO node, it will first check the RW node's global level timestamp, then the table and page level timestamps, Once one level is satisfied, it will directly process the request and will not check the next one. It only needs to wait for the log application on the requested pages when the last level (page level) is unsatisfied.

Since the latest modification timestamps are maintained on the RW node, the RO node has to fetch it from the RW node for each request. Although the RDMA network is fast, the overhead is still significant if there is a heavy load on the RO node. To overcome the overhead on the timestamp fetching, we propose the Linear Lamport timestamp. Based on the Linear Lamport timestamp, the RO node can store the timestamp locally after fetching it from the RW node. Any request arriving at the RO node earlier than TS_{ro} can directly use the locally stored timestamp instead of fetching a new one from the RW node. This can save many fetching requests when the load is heavy on RO nodes.

At last, to further minimize the network overhead, we adopt the one-sided RDMA for the log shipment and timestamp fetching. We designed a one-sided RDMA-based log shipment protocol to write the RW node's log to the RO nodes. The one-sided RDMA also saves a lot of CPU cycles during remote writing.

4 THE DESIGN OF POLARDB-SCC

4.1 Linear Lamport timestamp

Serving a strongly consistent read requires that a read request could see all the committed data that happens before its start time. In PolarDB-SCC, only the RW node can update data. Thus, the RW node plays the role of the timestamp oracle (TSO). To guarantee a strongly consistent read, the RO node has to fetch the RW node's latest timestamp and wait for the logs to be applied before handling this request. If the loads are heavy on RO nodes, there will be many concurrent timestamp fetching requests on the RW node, which has a negative impact on the performance. Moreover, timestamp fetching will induce extra overhead for read requests on RO nodes. To avoid having intensive concurrent timestamp fetching operations, PolarDB-SCC designs a Linear Lamport timestamp, in which one-time timestamp fetching from the RW node could conditionally serve a batch of read requests on the initiating RO node.

The vanilla read-wait scheme requires each read request on the RO node to fetch the RW node's timestamp before the actual execution. Actually, if a request finds that someone already fetched a timestamp after its arrival time, it can directly reuse it instead of fetching a new one, which can still guarantee strong consistency. So one request can reuse other request's timestamp if that timestamp satisfies the above relations. We prove this with an illustration shown in Figure 5. There is a request ($r2$) on the RO node that start fetching (e_2) the RW node's timestamp at t_2 and get the response (e_3) with the RW node's timestamp (TS^3_{rw}) at t_3. Then we can get the happen-before relation of ($e_2 \rightarrow TS^3_{rw} \rightarrow e_3$) [4]. There is another request r_1 that arrives at the same RO node (e_1) at t_1. By assigning a local timestamp for each event that happens on the RO node, if r_1 satisfies the relation of ($e_1 \rightarrow e_2$), we can infer that $e_1 \rightarrow e_2 \rightarrow$

[4]The symbol $\rightarrow$ indicates the happen-before relation.

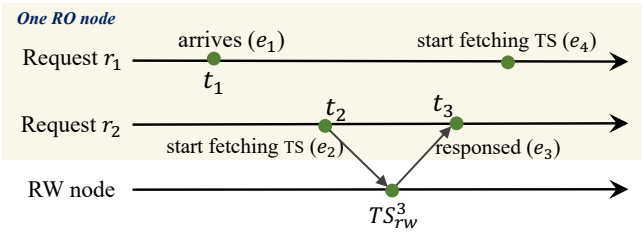

Figure 5: The example of Linear Lamport timestamp

TS^3_{rw}. So TS^3_{rw} could reflect all the updates that happen before e_1. As a result, if r_1 tries to fetch the RW node's timestamp after t_3 ($e_3 \rightarrow e_4$), it can directly use TS^3_{rw}, and consequently, save one timestamp fetching operation for r_1. Actually, in this case, any request arriving before t_2 could all reuse the timestamp TS^3_{rw} if it requires the RW node's timestamp after t_3. But there is a special case in that one request arrives before t_2 but tries to fetch the RW node's timestamp between t_2 and t_3. This request can be aware that an in-flight fetching request was sent before it. So it will wait for the response of the last fetching request and reuse its timestamp later. If there are many such concurrent read requests, this design can save a significant amount of overhead.

Based on this theory, we can cache the RW node's timestamp on the RO node and conditionally reuse it for some requests. To implement this in PolarDB-SCC, whenever an RO node fetches the RW node's timestamp, it will cache this timestamp (TS_{rw}) locally with the fetch operation's start timestamp (TS_{ro}) in a tuple $< TS_{rw}, TS_{ro} >$. With this design, if a request on an RO node requires the RW node's timestamp, it will first check the tuple $< TS_{rw}, TS_{ro} >$ on this RO node. If the request arrives before TS_{ro}, the locally cached TS_{rw} could be directly used by that request to ensure strong consistency, i.e., it does not need to fetch a new timestamp from the RW node.

For transactions in *repeatable read* or higher isolation levels, it only acquires the RW node's timestamp once for one transaction at the beginning of that transaction. All requests in the transaction will use this timestamp for its strongly consistent reads. Subsequently, a transaction arriving on an RO node first compares the transaction's start time with this RO node's TS_{ro} to check if the cached TS_{rw} is valid for the purpose of the transaction's strongly consistent reads.

4.2 Hierarchical modification tracker

In the vanilla read-wait scheme, before handling a read request on an RO node, it always has to wait for the logs that happen before a specific timestamp (that is fetched from the RW node) are applied. That means it always waits for all its local in-memory data to be up-to-date, even if this request only accesses a small subset of the data which may be already up-to-date. To avoid the unnecessary wait for unrelated log applications with respect to a read request, we propose a novel protocol for modification tracking. It tracks the RW node's latest modification timestamp on different levels, which enables the RO node to check the timestamp at different levels and only needs to wait for the requested data to be up-to-date.

Overview. Relational databases usually organize data into tables at the logical level and manage the physical data at page units. Therefore, we track the RW node's latest modification at three levels: the

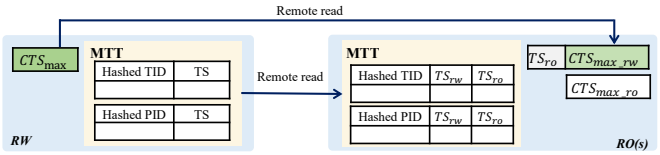

Figure 6: The design of hierarchical modification tracker

top level maintains the global database's latest modification timestamp, and the second/third levels maintain the table/page's latest modification timestamps. Since consistency is usually considered at the transaction level, we use the global committed timestamp as the global level's timestamp. But tracking a table/page's commit timestamp may induce much extra overhead because it requires tracking all the modified pages/tables for each transaction and updating their commit timestamp during committing. The mainstream databases always generate the corresponding log for each update on a table/page. We thus piggyback on the existing log sequence number (LSN) as the table/page's modification timestamp and it will not incur extra overhead.

Data structures. Figure 6 illustrates the architecture of the hierarchical modification tracker. The top level only needs to maintain one timestamp, but the second/third levels have to maintain many timestamps for different tables/pages. So we design the *modification tracking table* (MTT) for the second/third levels to record the page's and table's latest modification timestamp. The MTT is organized as hash tables. The hash table's key is the hashed value of the table ID (TID) or page ID (PID), and the value is the corresponding latest modification timestamp. The RW node will update the top-level timestamp when a transaction commits and update the MTT accordingly when a related page/table is updated. The RO node could fetch these three levels' timestamps from the RW node and cache them locally with its local timestamp (TS_{ro}) for reusing (as introduced in Section 4.1). So the RO node has the same data structures of the three levels' timestamps as the RW node, but it always has one more field to store its local timestamp at each level.

Workflow. To perform a strongly consistent read on the RO node, it will first check the global level's timestamp, then the requested table's and page's timestamps. Once one level is satisfied, it will directly process the request and will not check the next one. It only needs to wait for the log application when the last level (page level) is unsatisfied. But there is some difference between the global level and table/page level's checking. Once a request is satisfied at the global level, it will not check the timestamp during its following data access because the whole database is up-to-date for the current request. If a request is only satisfied on one requested table/page, it has to check the timestamp when accessing the different tables/pages. If the RO node's log application is fast enough, many requests can be satisfied at the global level. So these requests only require one-time timestamp checking. This can save timestamp-checking operations compared to the page/table level, which requires checking the timestamp for each requested page. But when the RO node's log application can not keep up with the updates on the RW node, the RO node may need to wait for the log application for the most read requests. In this case, the table/page level's timestamp checking can help to avoid waiting on unrelated data. Therefore, these three levels' timestamps work together to achieve low latency on RO nodes in different situations. Also, it can benefit from the Linear Lamport timestamp (Section 4.1). For each level's timestamp checking, it can directly use the locally cached timestamp if it is valid for the current request. This further saves many tiemstamp fetching operations.

MTT's implementation. To avoid having a large memory footprint, it's not practical to store all pages'/tables' newest modification timestamps to MTT. Hence we organize MTT in a hash table. On the RW node, multiple pages or tables will be hashed to the same MTT slot. When an RO node fetches one timestamp from the RW node's MTT according to a hashed PID/TID, this timestamp may belong to a different page/table due to the potential of hash collision. To avoid getting an elder timestamp, RO nodes only update an MTT record's timestamp when the newer one is larger than the former value. In this case, the timestamp in an MTT's slot is always the largest one among all timestamps that map to this slot. Hence, waiting for a larger timestamp to be applied still satisfies the strong consistency requirement if a hash collision occurs in this process. Typically, MTT's size is only hundreds of megabytes, much smaller than its buffer pool size (usually dozens of gigabytes).

Considerations for one-sided RDMA. Since one-sided RDMA does not require the remote machine's CPU's involvement and usually has lower latency compared to two-sided RDMA operations [33], we fully utilize the one-sided RDMA in the timestamp fetching. One of the considerations for implementing MTT via a hash table is that the hash table is more friendly to the one-sided RDMA than other data structures (*e.g.*, LRU or its variants). When accessing data from a remote machine's memory via one-sided RDMA, it must know its memory address on the remote machine in advance. So this requires that the RW node should not dynamically change the data structure's size or remove/add elements during run time. Otherwise, the RO node can not know if an element is in the RW node or its remote address. With the hash table design, the RW node can allocate the memory space for the hash table when the database starts and send its address to the RO node when it registers to the RW node. The hash table's size is fixed at startup. When an RO node tries to read a page/table's timestamp, the target MTT record's offset can be computed from the page/table ID. Then it can remotely read the timestamp by combining the hash table address and the offset without asking the RW node for the target remote memory address. Thus the timestamp fetching can be done via the one-sided RDMA to save the RW node's CPU resource and reduce network overhead for timestamp fetching. To overcome the problem caused by hash collision, we only update the timestamp when the incoming value is larger than the existing one. This can still satisfy the strong consistency read as introduced above. On the contrary, the LRU (or its variants) will dynamically insert/evict elements, making one-sided RDMA unusable in this case. There are also some RDMA-optimized hash table designs [15, 64]. However, they are often more complex and have more functionalities. In PolarDB-SCC, MTT's size is usually fixed at the start or is not changed too often. Most operations are updates and reads. So we use the vanilla hash table for MTT's implementation.

The hierarchical modification tracker design requires more operations for different levels' timestamp fetching compared to the vanilla single-level timestamp, but it does avoid the unnecessary

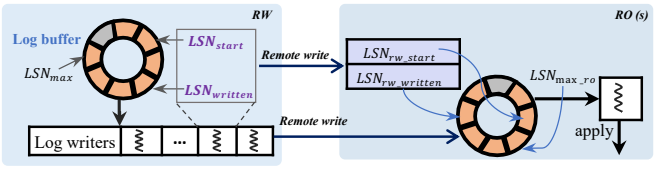

Figure 7: The design of the RDMA-based data shipment

wait for the log application. The timestamp could also be cached on the RO nodes for reuse. This extra overhead can be amortized on multiple requests. Moreover, the timestamp fetching is done via the fast one-sided RDMA, usually in several microseconds.

4.3 RDMA-based log shipment

PolarDB-SCC utilizes the one-sided RDMA for the log shipment to reduce network overhead and save CPU cycles. As shown in Figure 7, each RO node also has a log buffer, and the RW node remotely fills it in. The RO node's log buffer size is the same as the RW node. The RW node's log data will always be remotely written to the same offset in the RO node's log buffer. The RW node allocates one log writer for each RO node, responsible for writing logs to the specific RO node via RDMA. Once a RO node is registered to the RW node (*e.g.*, for serving elastic workload), its corresponding log writer is launched by the RW node.

If the log writers can not keep up with the log generation on the RW node, the log buffer will be overwritten before being written to RO nodes. Meanwhile, the RO node's log buffer could also be overwritten if its log application could not keep up with the RW node's remote writing. Thus, careful designs are required for the remote log writing on the RW node and the log reading on the RO node. Figure 7 shows the design for such log writing and reading. On the RW node, each log writer has two thread-local variables, LSN_{start} and $LSN_{written}$, representing that the logs between LSN_{start} and $LSN_{written}$ have been successfully written to the corresponding RO node. The global LSN_{max} is the maximum LSN that has been written to the local buffer. So, on the RW node, log data between $LSN_{written}$ and LSN_{max} should be remotely written to the corresponding RO node. Once LSN_{start} and $LSN_{written}$ are updated, they will be remotely written to the RO node's LSN_{rw_start} and $LSN_{rw\_written}$ respectively. Therefore, on the RO node, only the log data between LSN_{rw_start} and $LSN_{rw\_written}$ are valid for applying. Meanwhile, the RO node maintains LSN_{max_ro} (maximum LSN that has been applied), and then compares $LSN_{rw\_written}$ with LSN_{max_ro} to derive if some new logs are written to its log buffer.

Algorithm 1 shows how the RW node remotely writes logs to an RO node's log buffer. The log writer firstly initializes LSN_{start} and $LSN_{written}$ with LSN_{max}, and remotely writes them to the corresponding RO node's LSN_{rw_start} and $LSN_{rw_written}$ (line 2). Once there are some new logs are appended to its log buffer (LSN_{max} becomes larger than $LSN_{written}$), it starts to write the logs between $LSN_{written}$ and LSN_{max} to the corresponding RO node (lines 9-10). Both before and after the log writing, the log writer has to check if the log buffer is overwritten (line 11-22). Since it's a ring buffer, it can check if the difference between LSN_{max} and $LSN_{written}$ is larger than the buffer size to determine if the log buffer is overwritten. If yes, it has to reset LSN_{start} and $LSN_{written}$ and restart the write processing (line 14). If the log writing is successfully

Algorithm 1 The log write on the RW node

```
1:  function LogWriteThread( )
2:      Initialize( )
3:      while True do
4:          # wait for the new log
5:          while LSN_written >= LSN_max do
6:              Wait( )
7:          end while
8:          # start to remotely write log
9:          start_lsn ← LSN_written
10:         end_lsn ← Min(LSN_max, start_lsn + WRITE_LEN)
11:         # check overwrite
12:         if IfBufOverWritten(start_lsn) then
13:             ReInitialize( )
14:             continue
15:         end if
16:         # actually write log to RO node
17:         RDMARemoteWriteBuf(start_lsn, end_lsn)
18:         # check overwrite again
19:         if IfBufOverWritten(start_lsn) then
20:             ReInitialize( )
21:             continue
22:         end if
23:         # update LSN_written
24:         LSN_written ← end_lsn
25:         # remotely write LSN_written to LSN_rw_written
26:         RDMARemoteWriteWrittenLSN(start_written)
27:     end while
28: end function
```

Algorithm 2 The log read on the RO node

```
1:  function LogReadThread( )
2:      while True do
3:          # wait for the new log
4:          while LSN_max_ro >= LSN_rw_written do
5:              Wait( )
6:          end while
7:          # start to read log from start_lsn to end_lsn
8:          start_lsn ← LSN_max_ro
9:          end_lsn ← LSN_rw_written
10:         if IfBufValid(start_lsn, end_lsn) == False then
11:             log_data ← ReadFromStorage(start_lsn, end_lsn)
12:         else if IfBufOverWritten(start_lsn) then
13:             log_data ← ReadFromStorage(start_lsn, end_lsn)
14:         else
15:             log_data ← ReadLogBuffer(start_lsn, end_lsn)
16:             if IfBufOverWritten(start_lsn) then
17:                 log_data ← ReadFromStorage(start_lsn, end_lsn)
18:             end if
19:         end if
20:         # parse the log data and put the parsed data to queue
21:         ParseLog(log_data)
22:         LSN_max_ro ← end_lsn
23:     end while
24: end function
```

completed, $LSN_{written}$ will be updated and remotely written to the corresponding RO's $LSN_{rw\_written}$ (lines 24-26). Therefore, the corresponding RO node could know if some new logs have been written to its log buffer by checking its $LSN_{rw\_written}$. If the RW node fails during the RDMA write, there may be some partial data on the RO node. But these log data will not be used by the RO node

because its $LSN_{rw\_written}$ is not updated by the RW node. The RO node will read the corresponding log data from the shared storage.

Algorithm 2 illustrates how an RO node reads logs from its log buffer. The RO node maintains the global maximum LSN (LSN_{max_ro}) that has been read from the log buffer or the shared storage). Once $LSN_{rw\_written}$ is larger than LSN_{max_ro}, the log reader starts to read the logs (between LSN_{max_ro} and $LSN_{rw\_written}$) from the log buffer. But, before initiating this read process, it has to check if the corresponding logs are valid (line 10). This is because the logs before LSN_{rw_start} will be considered invalid. It further checks if these logs are overwritten before and after reading the log (line 12-19). If the log buffer is invalid or overwritten, it has to read the redo log from the shared cloud storage instead. After reading, the RO node can parse the logs being read and register the parsed log entries to another queue for the log application (line 21). Finally, it updates LSN_{max_ro} to the latest LSN that has been read (line 22).

4.4 Read-your-writes consistency

In a read-write transaction, however, there is a challenge that we have to guarantee that the read requests on the RO nodes must read the updates in the same transaction that happened on the RW node, which is called *read-your-writes consistency*. This is a general problem for a database or storage cluster and widely addressed by many deployed systems and proposed prototypes [20, 41, 43, 46, 48, 50]. PolarDB-SCC follows a similar design to guarantee read-your-writes consistency. Each write in the PolarDB-SCC would generate the corresponding redo log with a unique incremental LSN. The RW node returns the LSN to the proxy node for each write request. Before sending a following read request in the same transaction to the RO node, the proxy node has to check the maximum applied LSN on the RO nodes to determine which RO nodes could serve this read request. If some RO nodes have satisfied this requirement, it will send the read request to one of these RO nodes by the load balancer. If none is available, it will be blocked to wait for the LSN to be applied on one of the RO nodes or eventually forwarded to the RW node after a timeout period. PolarDB-SCC provides different options for applications to handle this situation. One is that the proxy could directly forward the read request to the RW node without waiting if no RO node satisfies the above condition. Another option is to send all read requests to the RW node without checking if there is an update before these reads in the same transaction.

4.5 High availability and recovery

PolarDB-SCC only piggybacks on existing redo logs for data synchronization without any changes to the logging scheme. The RDMA-based log shipment also does not change the existing log buffer's management. Before committing a transaction, PolarDB-SCC still synchronizes the corresponding logs to the shared cloud storage as usual to ensure the transaction's durability with WAL. Therefore, PolarDB-SCC has no impact on the existing recovery policy. If an RO node fails, the corresponding log writer on the RW node will find the RDMA network to this RO node is disconnected and stop writing logs to this RO node. Once this RO node restarts and registers to the RW node, that log writer will resume to work. The RO node will read the required logs from the shared cloud storage if they are not in its log buffer. If the RW node fails, one of the RO nodes will be promoted to be the new RW node, following the same procedure as that in a cloud-native database.

4.6 Compatibility

The hierarchical modification tracker could also be applied to other databases, such as Aurora, Socrates, and MySQL. These systems also organize the physical data at the page unit. So, they can also maintain the table/page's latest modification timestamp on the RW node or even directly apply MTT design to their RW nodes. This design could also be applied to key-value stores. The RW node could record the KV pairs' modification timestamps and could be fetched by RO nodes. Linear Lamport timestamp and RDMA-based log shipment are more general designs. They are not database-specific. These theories and implementation could also be used in other databases or storage systems.

5 EVALUATION

5.1 Experimental setup

Test platform. PolarDB-SCC is implemented with a commercial cloud-native database (PolarDB). It's in production and commercially available at Alibaba Cloud. Our evaluations are all conducted in a cloud environment. In our test, the underlying physical machines are equipped with 2 Intel Xeon Platinum 8269CY CPUs and 755GB DDR4 DRAM, running CentOS-7 OS. These physical machines are connected by a 50Gbps Mellanox ConnectX-4 network.

System configurations. At Alibaba Cloud, the most popular type of PolarDB instances is 8 vCPUs and 32GB memory (*8c32g*), and it's usually deployed with one RW node and one RO node. So we test PolarDB-SCC with the same configuration in most test cases. We further evaluate PolarDB-SCC's performance with larger instances with 88 vCPUs and 710GB memory (*88c710g*) and more RO nodes (up to 8). The buffer pool size is 24GB for the 8c32g instance and 533GB for the 88c710g instance. The log buffer size is 64MB and 2GB on 8c32g and 88c710g instances, respectively. The MTT's size is 128MB on 8c32g instances and 2GB on 88c710g instances. Finally, we adopt the *read committed* isolation in all systems during the evaluation. This is the default isolation level for many databases and is widely used for many user applications. Note that this is also the isolation level that may present the most overheads for ensuring strong consistency (i.e., most challenging) because it requires the RW node's latest timestamp for each query.

Baseline setup. Since PolarDB-SCC is implemented in PolarDB, PolarDB is the natural baseline. We configure PolarDB in three ways to compare with PolarDB-SCC:

- PolarDB$_{default}$: This is the default configuration in PolarDB. It handles all the read and write requests on the RW node. The RO node is only used for failover (providing high availability).
- PolarDB$_{read-wait}$: This is the current configuration used to provide strongly consistent reads when needed (e.g., for e-commerce and trading transactions in Alibaba). The RO nodes use the vanilla read-wait scheme to achieve a strongly consistent read.
- PolarDB$_{stale-read}$: RO nodes directly process the read requests without waiting for any log application. It may return stale data.

Furthermore, we also compare PolarDB-SCC with MGR (MySQL Group Replication) and two popular commercial databases, DB-A

and DB-C. Since DB-A does not support a strongly consistent read on RO nodes, we use its multi-master version in our evaluation.

Workloads. We evaluate PolarDB-SCC with three standard OLTP benchmarks (Sysbench [25], TPC-C [13] and TATP [38]) and a real production workload from Alibaba (with a profiled mix of 3:2:5 insert:update:select ratio). For Sysbench, we set up a database with 100 tables, each with 0.5 million records on *8c32g* and 10 million records on *88c710g*. Unless otherwise stated, requests are issued with uniform distribution and adopt the default values for other configurations. We configure TPC-C with 60 warehouses and run its read-only transactions on RO nodes (by following the previous work [56]). For TATP, we configure it with 100K subscribers.

5.2 Overall performance

Sysbench's read-write workload. As PolarDB-SCC primarily targets strongly consistent reads in the read-write workloads, we start with Sysbench's read-write workloads. Figure 8(a) shows the performance with uniform distribution. It indicates that PolarDB-SCC always has similar performance (both throughput and latency) to PolarDB$_{stale-read}$ under different workload pressures for the following reasons: First, most of the RO node's potential timestamp fetching requests hit its cached CTS_{rw_max}, saving a lot of network overhead. Second, PolarDB-SCC avoids much unnecessary wait time for the log application. Third, the one-sided RDMA-based log shipment and timestamp fetching speed up the data transfer and eliminate the extra CPU overhead. These enable PolarDB-SCC to achieve strongly consistent reads with little extra overhead.

When the workload pressure is light (*e.g.*, 16 and 32 threads), PolarDB$_{default}$'s performance is nearly identical to others. But it becomes saturated at 64 threads and performs worse than PolarDB-SCC and PolarDB$_{stale-read}$ because their RO node could help process read requests. PolarDB$_{read-wait}$'s throughput drops by 11.96% compared with PolarDB$_{default}$ at 16 threads because its RW node is not saturated while its RO node induces extra overhead. When increasing the number of threads, PolarDB$_{read-wait}$ can come up with PolarDB$_{default}$ because it can benefit from processing reads on the RO node. But its throughput is still not better than that of PolarDB$_{default}$. This is because all the read requests on the RO node will incur the same amount of timestamp query requests on the RW node. This makes the RW node the bottleneck. PolarDB$_{read-wait}$'s median latency is much better than that of PolarDB$_{default}$ at 256 threads. This is because PolarDB$_{default}$ is saturated under such heavy pressure. However, compared with PolarDB$_{default}$ and PolarDB$_{read-wait}$, PolarDB-SCC increases the throughput by up to 1.82× and 1.70×, and reduces the median latency by up to 3.03× and 1.42×, respectively.

Figure 8(b) shows the results under the Zipfian distribution. It has a similar trend to that of the uniform distribution. As our evaluation targets CPU-bound scenarios, neither uniform nor distribution workload is involved in storage I/O (except for the write-ahead log persistence). All of their requests could be finished in memory, which makes nearly no performance difference between a uniform and a Zipfian distribution. There is only a small difference in the 256-thread testing that PolarDB-SCC's performance drops 11.03% compared to PolarDB$_{stale-read}$. This is because some hot data are frequently updated on the RW node, and at the same time, many

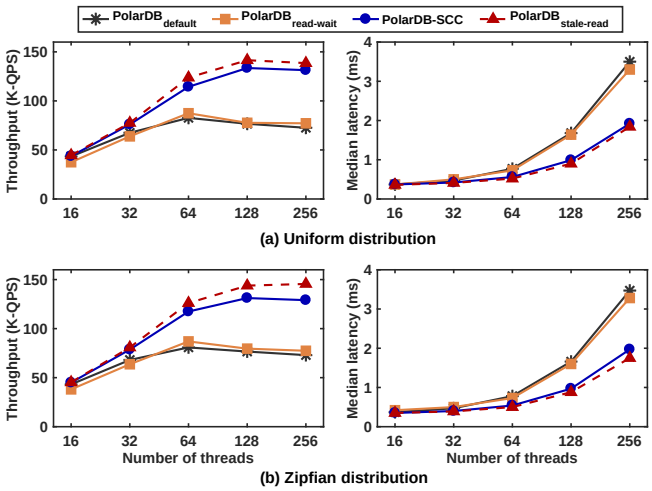

Figure 8: Performance of Sysbench's read-write workloads

Table 3: Tail latency comparison within different systems

Latency (ms)	Uniform			Zipfian		
	P90	P95	P99	P90	P95	P99
PolarDB$_{default}$	3.82	5.09	9.22	3.82	5.18	9.73
PolarDB$_{read-wait}$	3.43	5.67	14.73	3.36	5.77	15.55
PolarDB-SCC	2.11	3.68	8.43	2.07	3.62	8.58
PolarDB$_{stale-read}$	1.82	3.25	7.56	1.76	3.13	7.43

threads are accessing them on the RO node. That causes a longer time to wait for the log application. But PolarDB-SCC still increases the throughput respectively by up to 1.79× and 1.68×, and reduces the median latency respectively by up to 3.08× and 1.42×, compared with PolarDB$_{default}$ and PolarDB$_{read-wait}$.

We report the tail latency in Table 3. Due to the space limitation, we only show the tail latency within 128-threads testing, but other configurations have a similar trend. PolarDB$_{read-wait}$'s P90 latency is lower than PolarDB$_{default}$, because PolarDB$_{read-wait}$'s RO node can process some read requests in this heavy workload, while PolarDB$_{default}$ handle all requests on the RW node, making RW node overloaded. In PolarDB$_{read-wait}$, the request on the RO node has to fetch the timestamp from the RW node and wait for the log application if needed. That makes it have much higher P95 and P99 latency than PolarDB$_{default}$. This higher tail (P95 and P99) latency makes PolarDB$_{read-wait}$ have no improvement in throughput compared with PolarDB$_{default}$, although its median and P90 latencies are lower. However, PolarDB-SCC's P90, P95, and P99 latencies are all lower than PolarDB$_{default}$ and PolarDB$_{read-wait}$ (by up to 1.84× and 1.82× respectively). Although PolarDB-SCC minimizes the extra overhead on RO nodes, it inevitably has some extra overhead induced by RDMA communication and log application. So its tail latency is slightly higher than that of PolarDB$_{stale-read}$.

TPC-C performance. Figure 9 gives TPC-C's throughput and P90 latency. Since the full TPC-C mix is a read-write workload, we run TPC-C's read-only transactions on the RO node (but no load on the RO node in PolarDB$_{default}$) following the setup in QueryFresh [56]. As expected, all systems have a similar performance on the RW node. In the PolarDB$_{read-wait}$, the RO node's

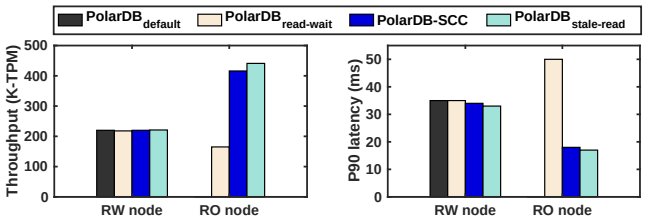

Figure 9: Performance of TPC-C workload

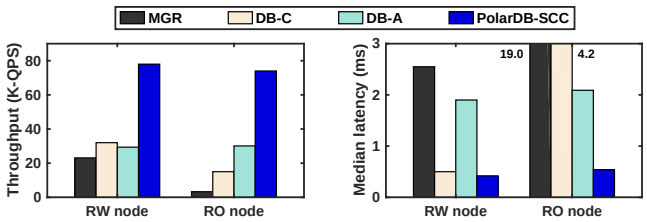

Figure 11: Comparison with commercial databases

throughput is not as that high as in the SysBench workload. So its RO node's timestamp fetching operations have little impact on its RW node's performance. That makes its RW node have a similar performance to other systems. That is different from the results in the SysBench workload. However, its RO node's throughput is only 39.6% of PolarDB-SCC's due to its high overhead, while PolarDB-SCC performs similarly to PolarDB$_{stale-read}$.

TATP performance. Figure 10 shows TATP's performance. When the workload is not heavy (16 or 32 threads), the RW node is not the bottleneck. These four systems (except PolarDB$_{read-wait}$) have a similar performance. But PolarDB$_{read-wait}$ performs worse due to its extra overhead on the RO node. When the workload becomes heavy (128 or 256 threads), PolarDB$_{default}$ is saturated, while PolarDB-SCC and PolarDB$_{stale-read}$'s performances are increasing. PolarDB$_{read-wait}$'s benefits exceed costs in the heavier workload, and it performs slightly better than PolarDB$_{default}$. PolarDB-SCC still shows a nearly identical performance to PolarDB$_{stale-read}$, improving the throughput by up to 1.89× and 1.61×, while decreasing median latency by up to 2.30× and 1.35× comparing with PolarDB$_{default}$ and PolarDB$_{read-wait}$.

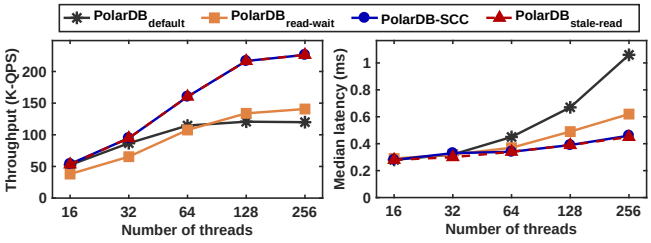

Figure 10: Performance of TATP benchmark

Compared with commercial databases. Figure 11 shows the performance comparison with some commercial databases by running SysBench's read-write workload on the RW node and read-only workload on the RO node with 64 threads. Due to the high cost for strongly consistent reads on the RO node, MGR's RO node's throughput is much slower than others and has a much higher latency. DB-C utilizes the read-wait design on the RO node to guarantee strong consistency and cause a longer latency and lower throughput on the RO node. DB-A shows a much lower latency than MGR and DB-C, but is still much higher than PolarDB-SCC. PolarDB-SCC's RO node's throughput is 2.45× higher than DB-A, and its median latency is only 25.8% of that in DB-A.

Production workload. We then run a trading service workload in Alibaba's production environment, in which 50% of the requests are reads. Figure 12 shows the throughput and RO node's stale read ratio (how many reads get stale data). As expected, PolarDB-SCC performs similarly to PolarDB$_{stale-read}$, but PolarDB-SCC completely avoids stale read while PolarDB$_{stale-read}$'s stale read ratio is about 98% on average. PolarDB$_{read-wait}$ and PolarDB$_{default}$ also eliminate stale read, but their throughput is only 64% of PolarDB-SCC's on average. PolarDB-SCC achieves a strongly consistent read on the RO node without sacrificing performance.

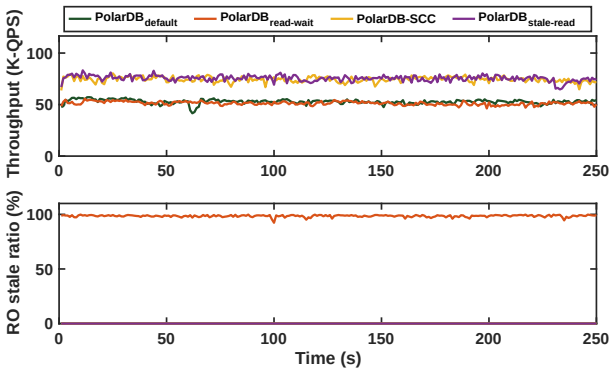

Figure 12: Performance of Alibaba product workload

5.3 Breakdown analysis

To further understand the performance impact of PolarDB-SCC 's individual optimizations, Figure 13 breaks down PolarDB-SCC's improvement by incrementally enabling its individual optimization technique. In this test, it has one RW node and one RO node, and runs Sysbench's read-write workload. We start with the vanilla read-wait scheme and gradually add on our optimizations. We first add the Linear Lamport timestamp (LLT) optimization and then the hierarchical modification tracker (HMT). Finally, we add the RDMA-based log shipment. By reusing timestamps, LLT significantly reduces the number of timestamp fetching. It improves the throughput by 18.37%-39.73%. We further deploy HMT to eliminate the unnecessary waits on unrelated log applications, continuing to improve the throughput by 20.00%-27.50%. Finally, we make further efforts to ship the log via the one-sided RDMA. But it only improves the throughput by about 4%, because the log shipment is no longer the bottleneck when LLT and HMT are both enabled. The median latency reduction shows a similar trend to that of the throughput.

Next, we show the performance breakdown with more RO nodes in Figure 14. We run Sysbench's read-write workload on the RW node and read-only workload on each RO node with 64 client threads. We also individually enable the different optimizations, similar to Figure 13. Figure 14 shows that the improvement of the individual optimization is more significant when having more RO

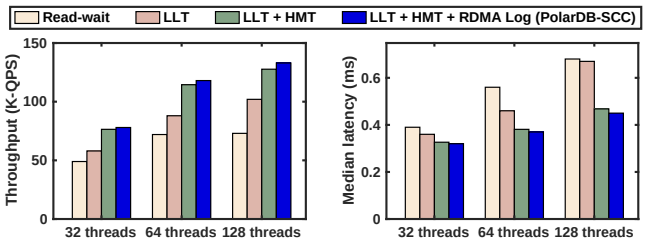

Figure 13: Breakdown analysis: different number of threads

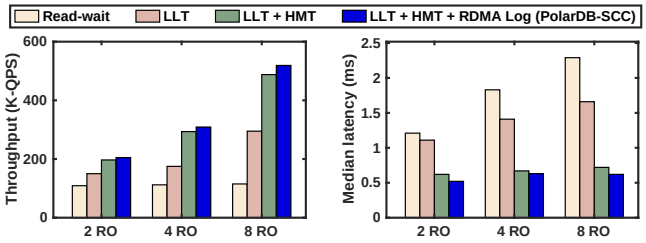

Figure 14: Breakdown analysis: different number RO nodes

nodes. When having 8 RO nodes, LLT improves the throughput by 2.56× compared with the read-wait policy, while only 1.37× and 1.56× at 2 RO nodes and 4 RO nodes. HMT can further improve the throughput by up to 1.67×. Lastly, the RDMA-based log shipment additionally makes an improvement of about 6%.

5.4 High scalability

As PolarDB-SCC could provide strong consistency on RO nodes, it's scalable to improve the performance by processing read requests on RO nodes. Figure 15 shows the scalability of PolarDB-SCC on a small instance (8c32g), by running Sysbench's read-write workload on the RW node and read-only workload on each RO node, both with 64 client threads. But for PolarDB$_{default}$, all workloads have to be processed on the RW node (to ensure strong consistency). It shows that PolarDB$_{default}$'s throughput does not increase with more RO nodes because its RO nodes do not handle any requests. However, the latency increases as there are more read requests on the RW node when adding more RO nodes. PolarDB$_{read-wait}$ only improves the throughput by 21.05% when increasing the number of RO nodes from 1 to 8. When having 8 RO nodes, it may have 512 concurrent requests on the RW node to get the timestamp, inducing too much overhead on the RW node. The RW node will be the bottleneck, making it hard to improve performances even having more RO nodes. However, PolarDB$_{stale-read}$ has no extra overhead for read requests on RO nodes, and PolarDB-SCC minimizes various overhead with its optimizations, achieving similar performance and scalability to PolarDB$_{stale-read}$ does. So both PolarDB-SCC and PolarDB$_{stale-read}$ increase the throughput when increasing the number of RO nodes. With 8 RO nodes, PolarDB-SCC's throughput is 3.73× higher than 1 RO node, while the latency is still comparable. Its throughput only drops 7.16% compared with PolarDB$_{stale-read}$. The median latency shows a similar trend. Furthermore, among the 8 RO nodes, their QPS ranges from 51.1K to 66.0K, with an average value of 56.12K, showing good load balancing.

We then show the scalability on larger instances (88c710g) in Figure 16 by running Sysbench's read-write workload. Every time

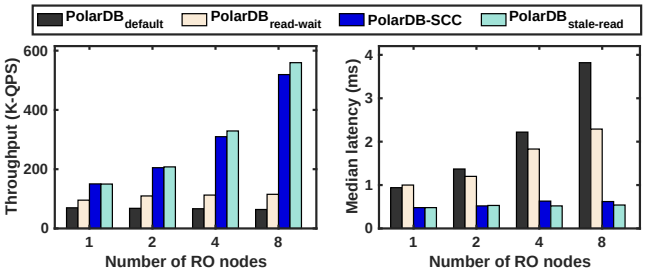

Figure 15: Performance with different number of RO nodes

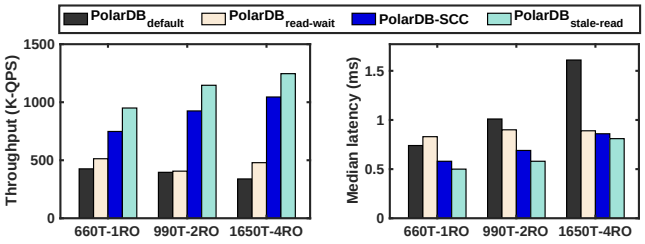

Figure 16: Performance on large database instances with different numbers of RO nodes

add one more RO node, we increase the number of client threads by 330 to saturate them. The x-axis represents the numbers of client threads and RO nodes, *e.g.*, xT-yRO means x client threads and y RO nodes. Similar to Figure 15, PolarDB$_{default}$ and PolarDB$_{read-wait}$ have no improvement on the throughput by increasing the number of RO nodes, but cause a higher latency due to heavier load. However, PolarDB-SCC still provides a comparable throughput and latency with PolarDB$_{stale-read}$. But, when running with 1650 client threads, the RW node could be saturated and become the bottleneck, therefore, the performance improvement of 4 RO nodes is not as significant as that on 8c32g instances. Nevertheless, PolarDB-SCC still improves the throughput by up to 3.08× and 2.18×, compared with PolarDB$_{default}$ and PolarDB$_{read-wait}$.

5.5 Support on serverless

PolarDB-SCC can provide a unified endpoint (e.g., via a proxy) that promises strong consistency for applications. Users can connect applications to that endpoint and increase/decrease the number of RO nodes at the backend without any changes to applications. They only need to pay for the necessary RO node resources to achieve high performance. PolarDB's serverless feature has the ability to dynamically adjust the number of RO nodes for dynamic workloads. Integrated with PolarDB-SCC, the application seems connected to a single RW node with dynamic resources in the background while guaranteeing strong consistency. We test this scenario in Figure 17. In this test, the workload becomes heavier at 300s, 600s, and 900s. The database cluster will dynamically add more RO nodes in the background and the throughput is quickly improved in an almost-linear fashion when more RO nodes are added to the cluster.

5.6 Use-case study on the cloud

Many customers from different vertical industry sectors (*e.g.*, finance, e-commerce, telecom, etc) on Alibaba Cloud have already

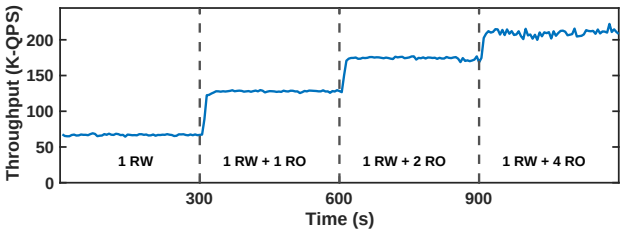

Figure 17: PolarDB-SCC with serverless

enjoyed the benefit of running their mission-critical workloads on PolarDB-SCC, especially for those with strong consistency requirements. In the interest of space, we elaborate one particular customer case here in this paper. This customer deploys their intelligent logistic application in a microservices way. One up-level task will be split into several stages and processed by different microservices in order. Once one stage is finished, it will notify the next one to process. The next one will read the previous updates from the database for the following processing. Since microservices are independent components, they highly rely on the database's strong consistency to guarantee the task can be done correctly. If the later stage cannot read the latest updates, it can not correctly process its own stage and cause problems to up-level applications. Without strongly consistent reads on RO nodes, they always send all read requests to the RW node, leaving the RO node idle (only used for high availability) and making the RW node bottleneck. Although the naive read-wait scheme achieves a strongly consistent read, the overhead is too high to be accepted by them. However, by deploying PolarDB-SCC, they can send all read requests to the RO node. It can guarantee a strong consistency without noticeable extra overhead. As a result, the RO node's resources are highly utilized and improve the total throughput by about 60%.

6 RELATED WORKS

Cloud-native database clusters. Aurora [55], Socrates [6], and PolarDB [32] are the mainstream cloud-native database clusters, which consist of one primary read-write (RW) node and one or more read-only (RO) nodes. The RW node synchronizes the log to the shared storage before committing a transaction. The RO nodes asynchronously read logs from the shared storage and apply them to keep the buffered data up to date. Since the log application is asynchronous, the RO node's buffered data may fall behind that of the RW node, which could cause stale reads on the RO node. PolarDB provides a strongly consistent read option on RO nodes, implemented with the naive read-wait scheme. But it causes a high overhead for the read request on the RO nodes. However, PolarDB-SCC aims to minimize the extra overhead for the strongly consistent reads on the RO nodes.

Variants of MySQL clusters. MySQL is a most common and widely deployed monolithic databases clusters [10, 17]. MySQL Group Replication (MGR) [37] enables users to create highly-available and fault-tolerant database clusters. MGR synchronizes the data between different nodes with the statement log. It supports both stale reads and consistent reads on RO nodes. However, the consistent reads on RO nodes have a very poor performance. Percona XtraDB cluster [42] and Galera Cluster [19] are the two popular database clusters based on MySQL. They are both compatible with ProxySQL [45] and HAProxy [21] to achieve high availability and read-write split. But they all do not support the low-latency, strongly consistent reads on the RO nodes [46, 53].

Other databases. Spanner [12] and TiDB [22] are distributed database clusters based on data sharding. They rely on the consensus algorithm (*e.g.*, Paxos [29] and Raft [40]) to keep the data consistent among the nodes, in which the log is shipped with TCP/IP networks. Their follower nodes could still return stale data, while strongly consistent reads sacrifice much performance. Amazon Dynamo [49] is a fully managed NoSQL database. It supports eventually consistent reads by default, while strongly consistent reads have much higher latency and is not supported on global secondary indexes [3]. QueryFresh [56] tries to minimize the replica node's staleness, but it may still return stale data and only guarantee eventual consistency. However, PolarDB-SCC completely avoids stale reads on RO nodes with a strong consistency guarantee, and it's also orthogonal with QueryFresh's design.

Distributed transaction optimizations. There are also some works to improve the distributed transaction protocols for higher performance and scalability, or stronger consistency [16, 23, 31, 36, 52, 57, 59–62]. However, PolarDB-SCC does not rely on the distributed transaction protocol. It only has one RW node that can process updates. The RW node maintains the global transaction ordering. The RW node and RO nodes share the storage that provides consistency and high availability. RO nodes only rely on the RW node's latest commit timestamp for the strongly consistent read.

Shared-memory/cache coherency in database clusters. There are some database clusters that adopt a shared-memory (or shared-cache) for performance or consistency requirements. They also face in-memory data consistency challenges. Oracle RAC's Cache Fusion technique [27] utilizes the high-speed inter-node message to keep data consistent. IBM's DB2 pureScale [8, 24] employs centralized lock management to control global cache access. PolarDB Serverless [11] proposes a cache invalidation mechanism to ensure cache coherency. However, these works are based on the shared-memory/cache design and their policies are all on the critical path of the writes, and may have a negative impact. PolarDB-SCC doesn't have a shared memory/cache. It only needs to keep the RO node's in-memory data up-to-date at the time of accessing it.

7 CONCLUSION

We design and implement PolarDB-SCC, a low-latency, strongly consistent cloud-native database. We first propose the hierarchical modification tracker, enabling the RO node to check timestamps at different levels and eliminating the wait time on unrelated log applications. We then design the Linear Lamport timestamp to save the timestamp fetching operations on RO nodes. At last, we design the RDMA-based log shipment to minimize the network overhead and CPU usage. Based on these designs, PolarDB-SCC achieve strongly consistent reads without noticeable extra overhead.

REFERENCES

[1] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian,

and Michael Wei. 2017. Remote Memory in the Age of Fast Networks. In *Proceedings of the 2017 Symposium on Cloud Computing*. 121–127.

[2] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. 2015. Challenges to Adopting Stronger Consistency at Scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[3] Amazon. 2012. Read Consistency of DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html. "[accessed-April-2022]".

[4] Amazon. 2022. Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless/.

[5] Amazon. 2022. Replication with Amazon Aurora. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Replication.html. "[accessed-April-2022]".

[6] 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.

[7] AWS. 2019. Global Tables: How It Works. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/globaltables_HowItWorks.html. "[accessed-April-2022]".

[8] Vlad Barshai, Yvonne Chan, Hua Lu, Satpal Sohal, et al. 2012. *Delivering Continuity and Extreme Capacity with the IBM DB2 pureScale Feature*. IBM Redbooks.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.

[10] Wei Cao, Feng Yu, and Jiasen Xie. 2014. Realization of the Low Cost and High Performance MySQL Cloud Database. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1742–1747.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 171–172.

[18] Funa. 2022. Funa Serverless. https://fauna.com/serverless.

[19] Galera. 2013. Galera Cluster. https://galeracluster.com/. "[accessed-April-2022]".

[20] GaleraCluster. 2015. Achieving Read-After-Write Semantics With Galera. https://galeracluster.com/2015/06/achieving-read-after-write-semantics-with-galera/. "[accessed-April-2022]".

[21] HAProxy. 2001. The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/. "[accessed-April-2022]".

[22] 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.

[23] Tianyang Jiang, Guangyan Zhang, Zhiyue Li, and Weimin Zheng. 2022. Aurogon: Taming Aborts in All Phases for Distributed In-Memory Transactions. In *20th USENIX Conference on File and Storage Technologies (FAST 21). USENIX Association*.

[24] 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.

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

[26] Cockroach Labs. 2022. CockroachDB Serverless. https://www.cockroachlabs.com/lp/serverless/.

[27] Tirthankar Lahiri, Vinay Srihari, Wilson Chan, Neil Macnaughton, and Sashikanth Chandrasekaran. 2001. Cache Fusion: Extending Shared-disk Clusters with Shared Caches. In *VLDB*, Vol. 1. 683–686.

[28] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *arXiv preprint arXiv:2103.00170* (2021).

[29] Leslie Lamport. 2019. The Part-time Parliament. In *Concurrency: the Works of Leslie Lamport*. 277–317.

[30] 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.

[31] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. 2015. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 71–86.

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

[33] Xinxin Liu, Yu Hua, and Rong Bai. 2021. Consistent RDMA-Friendly Hashing on Remote Persistent Memory. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 174–177.

[34] Microsoft. 2022. Azure SQL Database Serverless. https://learn.microsoft.com/en-us/azure/azure-sql/database/serverless-tier-overview?view=azuresql.

[35] Microsoft. 2022. Use Read-only Replicas to Offload Read-only Query Workloads. https://docs.microsoft.com/en-us/azure/azure-sql/database/read-scale-out?view=azuresql. "[accessed-April-2022]".

[36] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 479–494.

[37] MySQL. 2016. MySQL Group Replication. https://dev.mysql.com/doc/refman/5.7/en/group-replication.html. "[accessed-April-2022]".

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

[39] Jethava Nikhil and Clugage Kevin. 2021. Databricks Serverless SQL. https://www.databricks.com/blog/2021/08/30/announcing-databricks-serverless-sql.html.

[40] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 305–319.

[41] Oracle. 2017. Read-Your-Writes Consistency. https://docs.oracle.com/cd/E17076_05/html/gsg_db_rep/C/rywc.html. "[accessed-April-2022]".

[42] Percona. 2018. Percona XtraDB Cluster. https://www.percona.com/software/mysql-database/percona-xtradb-cluster. "[accessed-April-2022]".

[43] Fabio Picconi, Pierre Sens, et al. 2005. Pastis: A highly-scalable multi-user peer-to-peer file system. In *European Conference on Parallel Processing*. Springer, 1173–1182.

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

[45] ProxySQL. 2013. A High Performance Open Source MySQL Proxy. https://proxysql.com. "[accessed-April-2022]".

[46] ProxySQL. 2018. GTID Consistent Reads. https://proxysql.com/blog/proxysql-gtid-causal-reads/. "[accessed-April-2022]".

[47] Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, et al. 2019. Native Store Extension for SAP HANA. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2047–2058.

[48] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. 2020. FlightTracker: Consistency across Read-Optimized Online Stores at Facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 407–423.

[49] Swaminathan Sivasubramanian. 2012. Amazon DynamoDB: A Seamlessly Scalable Non-relational Database Service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 729–730.

[50] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. 1994. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. IEEE, 140–149.

[51] The Transaction Processing Council. 2007. TPC-E Benchmark. http://tpc.org/tpce/. "[accessed-April-2022]".

[52] 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.

[53] Marco Tusa. 2021. Full Read Consistency Within Percona Operator for MySQL Based on Percona XtraDB Cluster. https://www.percona.com/blog/2021/01/11/full-read-consistency-within-percona-kubernetes-operator-for-percona-xtradb-cluster/. "[accessed-April-2022]".

[54] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, et al. 2022. Enabling the Next Generation of Multi-Region Applications with CockroachDB. In *Proceedings of the 2022 International Conference on Management of Data*. 2312–2325.

[55] 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.

[56] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2017. Query Fresh: Log Shipping on Steroids. *Proceedings of the VLDB Endowment* 11, 4 (2017), 406–419.

[57] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. 2021. Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 357–372.

[58] Wikipedia. 2015. Consistency Model. https://en.wikipedia.org/wiki/Consistency_model.

[59] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. 2014. Salt: Combining ACID and BASE in a Distributed Database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 495–509.

[60] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1289–1302.

[61] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. 2018. Building Consistent Transactions with Inconsistent Replication. *ACM Transactions on Computer Systems (TOCS)* 35, 4 (2018), 1–37.

[62] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. 2013. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 276–291.

[63] 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.

[64] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *USENIX Annual Technical Conference*. 15–29.