SunStorm: Geographically distributed transactions over Aurora-style systems

Cuong D. T. Nguyen*
University of Maryland, College Park
ctring@umd.edu

Pooja Nilangekar*
University of Maryland, College Park
poojan@umd.edu

Heikki Linnakangas
Neon, Inc.
heikki.linnakangas@iki.fi

Daniel J. Abadi
University of Maryland, College Park
abadi@umd.edu

ABSTRACT

There are two main approaches to scaling transactional database workloads: (1) a shared-nothing architecture with distributed transaction processing, or (2) an Aurora-style shared-storage architecture with separate compute and storage layers that scale independently. In option (2), the compute layer typically contains a single writer node and all other compute nodes are read-only. This may lead to scalability limits for write-intensive workloads, and introduces communication latency for write transactions that initiate far from the writer node. However, shared-nothing systems must pay the overhead of distributed coordination and commit protocols. In this paper, we discuss the design of a more scalable version of Aurora-style systems which supports multiple writer nodes managing geographically partitioned data. It yields many of the efficiency benefits of Aurora-style systems while removing the scalability bottleneck. Furthermore, geographic partitioning improves latency by over an order of magnitude for global applications in which clients from across the world can experience local write performance.

PVLDB Reference Format:
Cuong D. T. Nguyen, Pooja Nilangekar, Heikki Linnakangas, and Daniel J. Abadi. SunStorm: Geographically distributed transactions over Aurora-style systems. PVLDB, 18(13): 5555 - 5568, 2025.
doi:10.14778/3773731.3773733

PVLDB Artifact Availability:
The source code, data, and other artifacts have been made available at https://github.com/umd-dslam/sunstorm.

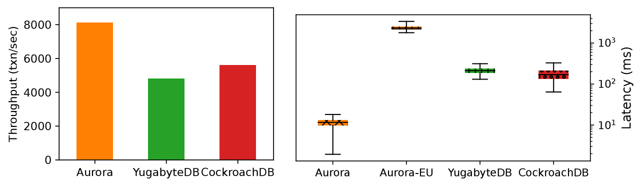
1 INTRODUCTION

There are two major competing schools of thought on scaling transactional processing to over tens of thousands of concurrent transactions. The first is to partition the database across a shared-nothing cluster of machines in one or more geographic regions. Every machine may contribute to transaction processing that accesses data within its partition. The larger the required scale, the more machines are added to the cluster to share load and meet these scale requirements. Examples of such systems include Calvin [46], Spanner [18], TAPIR [55], Carousel [50], CockroachDB [45, 47], YugabyteDB [2], OceanBase [53], Chardonnay [22], and Detock [38].

The other school of thought is to use a shared-storage architecture in which compute and storage are decoupled. Multiple compute nodes exist to handle the client workload, accessing the same shared storage layer (which typically consists of data partitioned and replicated across a cluster of servers, potentially also replicated across geographic regions). This design in which compute nodes have minimal long-term state enables extremely high elasticity as new compute nodes can be added or dropped on the fly, without having to copy or move data from other nodes. However, by avoiding state, these compute nodes cannot support concurrency control protocols across multiple nodes. Therefore, they typically designate only a single compute node to support requests that update the data stored, and all other compute nodes only support read-access. Examples of such systems are Amazon Aurora [48], Google AlloyDB [37], Microsoft Socrates [9], PolarDB [13, 14, 51, 52], and Neon [5].

The first school of thought is fundamentally more scalable since it does not limit write access to a single instance [57]. However, it struggles with multi-partition transactions, which are notoriously challenging to perform at high levels of performance and correctness, and introduce much complexity into the system. These transactions typically require advanced distributed transaction functionality, which may include distributed concurrency control, coordination, deadlock resolution, and consensus protocols, which can significantly increase the amount of code executed per transaction.

To illustrate this tradeoff, we ran an experiment that compares the performance of Aurora (second school) vs. two state-of-the-art examples from the first school: CockroachDB and YugabyteDB in a two region deployment (us-east-1 and eu-west-1) using the same



(a) Throughput **(b) Latency**

Figure 1: Single-writer vs. shared-nothing database systems.

*Both authors contributed equally to this work.

Proceedings of the VLDB Endowment, Vol. 18, No. 13 ISSN 2150-8097.
doi:10.14778/3773731.3773733

per-region set-up described in Section 7.1 and the single-region read-modify-write benchmark from Section 7.2. Within a region, each system used its default partitioning scheme: range partitioning in CockroachDB, hash partitioning in YugabyteDB, and no partitioning in Aurora. Since Aurora stores all its data at the primary, writes from eu-west-1 are forwarded to the primary node in us-east-1. When data is accessed via a uniform distribution, this causes most transactions to be multi-partition in CockroachDB and YugabyteDB, giving Aurora's partition-free architecture a major advantage. Even though none of these multi-partition transactions access data in multiple regions, Aurora processes 1.5x more transactions compared to the distributed systems, despite having 1/6 the write-compute resources. Thus, Aurora is 12 times more efficient on a per-machine basis. These results are not surprising and are well-known in the industry. Distributed transactions are complex; for a workload that can run on a single machine, it is most efficient to avoid the coordination required for distributed execution.

This phenomenon has driven the success of Aurora-style systems that are built according to the second school of thought, avoiding much of the complexity of distributed transactions. The lack of write scalability has only occasionally been problematic, since write transactions are rarer than read transactions in many real-world workloads. Furthermore, Aurora's new multi-writer version, "Limitless", aims to alleviate this scalability bottleneck [54], albeit with poor consistency guarantees in the recommended configuration.

However, a bigger problem (not addressed by Aurora Limitless) has emerged with this approach: poor latency in geo-replicated scenarios. Applications are increasingly global in their design and have users across continents. By enabling only a single instance to write data, all write transactions, no matter where they initiate, must be sent to the primary instance for processing. If a client is located halfway across the world from the primary instance, that user necessarily experiences latencies of hundreds of milliseconds for every write transaction. These types of latencies can be problematic for many applications [47].

In contrast, shared-nothing DBMS can be extended, often in a natural and straightforward way, to perform their partitioning schemes across continents. This can potentially enable read and write transactions initiated from a location close to the partition that stores data accessed by those transactions to be processed at local (less than 10 milliseconds) latency. Both CockroachDB and YugabyteDB support geo-partitioned transactions in this way. For example, Fig. 1b shows that transactions initiated in eu-west-1 are 3-4 times slower in Aurora than the other systems, since they use geographic partitioning to process those transactions locally, while Aurora needs to send them to its primary node in us-east-1.

These results motivate the need for a geo-partitioned version of an Aurora-style system. To that end, we designed SunStorm, a system that guarantees strict serializability while still supporting multiple writers in multi-region settings. Our design follows two guiding principles: (1) minimize assumptions about the DBMS used at the compute layer, and (2) avoid cross-region communication on the critical path of transaction processing where possible. SunStorm uses a shared-storage architecture yet also partitions this storage across regions. SunStorm only incurs the cost of coordination for multi-partition distributed transactions that access data at more than one region. For workloads in which such transactions are rare, the system achieves the throughput of Aurora-style systems along with lower observed latency, since the clients are serviced by write nodes in the same geographic region, as is the case in shared-nothing geo-partitioned systems. In addition, it removes the scalability bottleneck of Aurora-style systems by providing more than one location that can service write transactions.

To evaluate the contributions of SunStorm, we run experiments to compare it to leading examples built according to the two competing schools of thought that we described above. We find that in addition to the expected result (that SunStorm is able to avoid the main disadvantages of each school of thought), SunStorm surprisingly achieves better latency and scalability for geographically dispersed transactions relative to systems from both architectures – even the geographically partitioned shared-nothing system.

2 AURORA-STYLE SYSTEMS

This section gives background on Aurora-style systems with an architecture that separates compute from storage, such as Amazon Aurora [48], Google AlloyDB [37], Microsoft Socrates [9], PolarDB [14, 51, 52], and Neon [5]. While they differ in some details and implementations, their general design principles described in this section are similar. We refer to this architecture as a **single-primary** architecture throughout the paper.

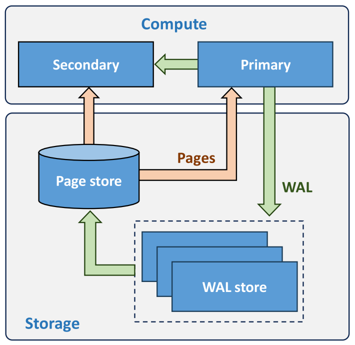


Figure 2: Single-primary architecture

Fig. 2 depicts the compute and storage layers of this architecture. The compute layer has a primary node, and optional secondary nodes, running a relational DBMS like PostgreSQL or MySQL. All transactions involving writes must be sent to the primary node, while secondary nodes only serve read-only transactions. Every data modification triggers the primary compute node to generate a **Write-Ahead Log** (WAL) record containing information for redoing the change on a specific page. The position of a record in the log is its **Log Sequence Numbers** (LSN).

Instead of reading from or writing to local disks, compute nodes forward these requests to the storage layer. The primary compute node sends the WAL to a WAL store within the storage layer, comprising multiple log storage nodes. Together with the compute node, these log storage nodes participate in a consensus or quorum-based protocols to replicate WAL records, ensuring durability and fault-tolerance. Additionally, the WAL store forwards WAL records to the page store, which can reconstruct any page at a specified LSN.

Read requests from any compute node—whether primary or secondary—are directed to the page store with a page identifier and an LSN up to which all relevant WAL records must be applied to that page. Pages retrieved from the page store are cached in the compute node's DBMS and subject to its buffer pool management system. The primary compute node is the only writer, and thus is always aware of the most recently persisted WAL record. Secondary nodes receive log records either directly from the primary node (e.g., Aurora, Socrates) or from the WAL store (e.g., Neon) to update their buffer pool pages and keep them up-to-date.

This architecture offers several advantages. By decoupling compute from storage, the two layers can scale independently and use hardware that suits their requirements. The storage layer can be transparently sharded to eliminate I/O bottlenecks. Failover time is reduced and more predictable since a new compute node can join and catch up quickly without copying data from another node. Data backup and archiving are easier as they can be done at the storage layer without interfering with the compute nodes' operation.

While this architecture can expand to multiple geographic regions to improve the performance for read-only transactions, all write transactions must go to the single region that hosts the primary node, resulting in increased latency of these transactions and a potential bottleneck at the primary region.

3 SUNSTORM ARCHITECTURE

SunStorm consists of multiple deployments of the single-primary system (described in the previous section), located in different geographic regions. The storage layers of these deployments are interconnected to enable cross-region data sharing. SunStorm also adds a thin layer of transaction servers responsible for coordinating transactions accessing data across regions. This architecture, referred to as a **multi-primary** architecture, is illustrated in Fig. 3.

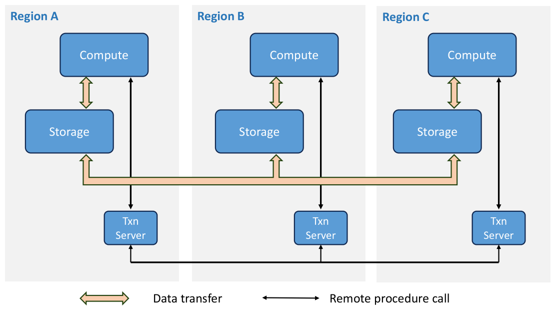


Figure 3: SunStorm architecture

A database consists of multiple partitions, each assigned to exactly one region as its **local data**. Partitions not local to a region constitute its **remote data**. The storage layer of each region continuously and asynchronously replicates its local data updates to all other regions. Thus, each region's storage layer stores both up-to-date local data and slightly stale remote data from other regions.

Compute nodes have complete read/write access to all data as if all data was local. During transaction execution, the compute node sends data page requests to its local storage layer, which immediately returns the most recently known version of each page without communicating with other regions. While local pages are always up-to-date, the possibility of reading stale data in remote pages necessitates a multi-region commit protocol.

Each compute layer node runs an instance of a traditional DBMS. Any DBMS can be used if it meets the following requirements:

(1) SSI [24] is used to guarantee serializable, ACID transactions[1].
(2) When a transaction commits, the DBMS assigns a monotonically increasing **commit sequence number** (CSN) to it and stores the CSN alongside every modified data item[2].
(3) Transactions can transition into a *prepared* state, after which it cannot be aborted by any factor except from the client.

Requirement (1) ensures strictly serializable, ACID properties for all parts of a transaction executed within the same region.

SunStorm commits multi-region transactions using an optimistic protocol. The CSNs in requirement (2) are used to establish their commit order, and relied upon during OCC validation to verify that the most recent version of a data item has been read. CSN monotonicity needs to be maintained only within each region, which can be achieved with an incrementing local counter. One such counter could be the LSN of the transaction's commit record[3].

This design decision of relying only on local CSNs instead of global CSNs has important consequences for the system's implementation. Using global CSNs is the more obvious choice, and would make a globally consistent snapshot readily available to each compute node, greatly simplifying SunStorm's validation process. Nevertheless, it would incur the overhead of global coordination, potentially affecting the latency and throughput of transactions that only access data within a single region.

Requirement (3) is necessary for SunStorm's multi-region commit protocol, which is described in Section 4.4. It ensures that once a region votes to commit a transaction, it does not inadvertently abort the local transaction running at the compute node. Many popular DBMSs, such as PostgreSQL, MySQL, Oracle, and SQL Server, can satisfy this requirement with their support for X/Open XA transactions [49]. Although these systems' XA transaction implementations include recording the prepared transactions to stable storage for recovery, this is not required to meet requirement (3).

By including one writer-node per region, SunStorm avoids Aurora's write-scalability bottleneck (as long as the write workload is spread across region-partitions). However, to be usable, SunStorm needs to support high throughput multi-region transactions.

4 TRANSACTIONS

Since SunStorm is designed not to be tied to any specific DBMS, we first define an abstract database model and then describe SunStorm's read/write operations and commit protocol based on this model.

4.1 Database Model

We define a *database* as a set of all possible uniquely identifiable *data items*, each associated with a *value*. The value domain includes a special value $\perp$ indicating uninitialized data items. The database

[1]Some parts of SunStorm can be simplified if locking or optimistic approaches are used instead of SSI. The high-level architecture can still be used without SSI.
[2]We added this feature to the DBMS used in our initial implementation. See Section 6.
[3]This is the method used by our current implementation: CSNs are a subset of LSNs.

system supports two operations: *read*, which retrieves a data item's current value, and *write*, which updates a data item's value.

In practice, examples of data items are tuples, index pages, and tables, which can be identified by (table name, page id, offset), (index name, page id), and (table name), respectively. In our model, an insertion, update, and deletion of a data item involves a read followed by a write operation. Specifically, an insertion reads from an uninitialized data item and writes a new value to it, and a deletion reads from an initialized data item and writes $\perp$ to it.

4.2 Read and Write Operations

A client initiates a transaction by connecting to the primary compute node of a nearby region. This node processes the entire transaction locally, even if it accesses remote data. It requests any needed pages (that are not already in cache) from the storage layer.

For local data pages, the compute node always requests the latest version. For remote pages, when a transaction accesses data from region R for the first time, it requests the latest known LSN for R from the storage layer. The transaction then reuses this LSN for all subsequent requests for data from R. The compute node caches pages of both remote and local data in the DBMS's buffer pool.

It is possible for multiple versions of the same remote page to concurrently exist in the buffer pool. For instance, T1 requests a remote data page at LSN 100, while T2 requests it at LSN 200.

A transaction records (1) the identifiers of all remote data items that it reads along with their respective LSNs, and (2) the new values of all writes to remote data items in an in-memory data structure called a **remote read/write set** (RemoteRWSet). Subsequent reads of previously written remote data items within the same executing transaction are read directly from this data structure. On commit, if the RemoteRWSet is not empty, it is sent to other regions involved in the transaction for validation and application of the new values.

Section 6 describes how writes to remote data are performed in a temporary buffer, converted to a **logical representation**, and then sent to the owning remote region for validation and application. By representing writes logically, index modifications need not be recorded, and remote information related to metadata stored inside the tuple header need not be coordinated across regions.

4.3 Concurrency Control

Other recent systems that attempt to build a strictly serializable, geographically replicated DBMS have used locking or optimistic types of approaches at each node in the system [18, 38, 42]. Our design choice to build over single-node systems that use MVCC (Multi-Version Concurrency Control) provides a wider variety of systems to be used at the compute layer but significantly complicates the ability of the system to guarantee strict serializability.

Most MVCC systems do not guarantee serializability out-of-the-box. Instead, they use reduced isolation levels – usually snapshot isolation (SI), which is susceptible to the write-skew anomaly [10]. Fekete et al. introduced the Serializable Snapshot Isolation (SSI) technique that enables SI systems to guarantee serializability in single-node (non-distributed) deployments [24]. The SSI paper proves that all violations of serializability in SI systems results from cycles in the read-write (**rw**) anti-dependency graph since

Algorithm 4.1: Modified Serialization check

```
   Input: Transaction to be serialized
   Output: Transaction status is PREPARED or ABORTED
1  Function CheckSerializable (MyXact) is
2      if MyXact.status = ABORTED then
3          return
4      for conflict in MyXact.inConflicts do
5          if conflict.hasInConflicts then
6              if conflict.status in (PREPARED,
                COMMITTED) then
7                  MyXact.status ← ABORTED return
8              else
9                  conflict.status ← ABORTED
10         if (conflict.isMultiRegion or MyXact.isMultiRegion)
            and conflict.status = PREPARED then
11             MyXact.status ← ABORTED return
12     for conflict in MyXact.outConflicts do
13         if conflict.status ≠ ACTIVE and
            (conflict.isMultiRegion or MyXact.isMultiRegion)
            then
14             MyXact.status ← ABORTED return
15     MyXact.status ← PREPARED return
```

snapshot isolation disallows write-write (**ww**) and write-read (**wr**) dependencies among concurrent transactions.

If two concurrent transactions (T_i, T_j) are executed by a database guaranteeing snapshot isolation and $T_i \xrightarrow{rw} T_j$, then the history is serializable as long as $T_i < T_j$ in the serial order. The SSI algorithm works based on the following theorem proved by Fekete et al. [24] and implemented in PostgreSQL by Ports and Grittner [41].

Theorem 4.1. *Every cycle in the serialization history graph contains a sequence of edges* $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$ *where each edge is a rw-anti-dependency. Furthermore,* T_3 *must be the first transaction in the cycle to commit.*

SSI guarantees serializability by preventing this condition from occurring. It tracks the **rw-anti-dependency** graph as the **conflict** graph and deems the $T_1 \rightarrow T_2 \rightarrow T_3$ structure unsafe for concurrent transactions, aborting one of the transactions involved in the structure via serialization checks during the commit process.

The SSI protocol works on single-node systems where the rw-conflict graph can locally track and abort unsafe structures at commit time. It cannot be extended directly to distributed systems in which the conflict graph is spread across multiple nodes, and the unsafe structures are not visible at a single location.

Therefore, SunStorm's concurrency control protocol needs to compensate for this lack of visibility. Unsafe structures involving two or more single-region transactions can still be detected and eliminated by the single-node SSI systems running at each compute node. Other types of unsafe structures (specifically those that involve multi-region transactions) are handled in SunStorm by expanding the serialization checks that occur during commit at each compute node. For every **rw**-anti-dependency of the form: $T_i \xrightarrow{rw} T_j$ where either T_i or T_j are mutli-region transactions, SunStorm imposes the following serialization checks:

(a) Reader rule: If the reader, T_i, attempts to commit and T_j is not *ACTIVE* (*PREPARED* or *COMMITTED*), abort T_i.
(b) Writer Rule: If the writer transaction, T_j attempts to commit and T_i is *PREPARED*, abort T_j.

Algorithm 4.1 gives pseudocode for these serialization checks. The lines in black are imposed are the original checks introduced by SSI to avoid unsafe conflict-graph structures, while the lines in red (lines 10 - 14) are additional checks added by SunStorm to ensure correctness in the face of multi-region transactions.

4.4 Commit Protocol

Single-region transactions are committed by the DBMS at the compute node using the protocol described in Section 2 without modification except for the additional serialization checks described in Section 4.3. Multi-region transactions are committed using a *decentralized 2-phase commit* (2PC) protocol [11] combined with a validation process that prevents write-read and write-write conflicts causing serializability violations. A transaction is considered "multi-region" if it accesses any data from a remote region, even if all accesses are from that same region.

4.4.1 Decentralized 2PC. This protocol is executed by transaction servers (TSs) that communicate independently from the storage layer. Its steps are as follows:

(1) The compute node processes the transaction locally, puts it in a prepared state, and passes its RemoteRWSet to the local TS.
(2) The local TS (**initiator**) logs the RemoteRWSet, and then sends a `PREPARE` message to all other TSs in the regions listed in the RemoteRWSet (**validators**). This message includes the RemoteRWSet and serves as a `YES` vote from the initiator.
(3) Upon receiving a `PREPARE` message, a validator TS initiates a **validating transaction** on its local primary compute node.
(4) The validating transaction runs as described in Section 4.4.2. If it passes, it is prepared, and control is passed to the local TS.
(5) If validation passes, the validator logs the RemoteRWSet, then sends `YES` votes to all participants Otherwise, it sends `NO` votes.

The initiator (6) and validators (7) independently collect these votes and commit the prepared transaction if all yes votes are received.

In contrast to the centralized version of 2PC [11], the decentralized protocol does not require a coordinator to collect the votes from the participants and send out commit messages. This saves one round of message passing from the coordinator to the participants [25], which is important in geo-distributed settings. Nevertheless, every participant sends one round of messages to all other participants, resulting in the number of messages being proportional to the square of the number of participants. This approach is taken because the number of participants (regions) is usually small.

4.4.2 Validation. For a given remote transaction T, each region validates access to its own items in T's RemoteRWSet. The validator retrieves the most recent CSN for each of these data items (requirement (2) from Section 3) and fails if the CSN is greater than the LSN at which the item was read. These two values can be compared since SunStorm uses the LSN of the commit record as the transaction's CSN (Section 3).

Phantom read prevention is integrated into validation by including the index pages that cover the range of a query predicate in the RemoteRWSet. Alternatively, the next index key beyond the end of an index scan could be used. These approaches correspond to index-range locking [11] and next-key locking [35] in lock-based protocols. We use the index-range approach in our prototype since PostgreSQL already supports index-range locking [41].

In theory, validation can be performed by the TS directly or sent to the compute layer to be performed there via a DBMS transaction. SunStorm uses the second approach for three reasons. First, validation requires extracting the CSN from data in the RemoteRWSet. The CSN is stored alongside other data within pages. The storage layer only knows how to return entire pages upon request. A non-trivial amount of parsing work must occur to extract the relevant CSN data from these pages. By performing this extraction effort inside the DBMS, SunStorm can take advantage of existing data deserialization, data traversal, type systems, and other code within the DBMS for this task. Second, by performing this work inside a DBMS transaction, SunStorm can leverage the DBMS's concurrency control system to isolate validation work from other local transactions running in the validator's local compute node, avoiding the need to implement an external mechanism [28]. Third, the validation work can be combined inside the same transaction that performs the tentative local writes for that transaction so that the built-in atomicity guarantees of the DBMS can ensure that the validation and writing happen together atomically.

5 PROOF OF STRICT SERIALIZABILITY

Our proof that SunStorm guarantees strict serializability follows two steps. We first prove that the schedule generated by SunStorm is serializable. Next, we prove that the equivalent serial schedule preserves the temporal ordering of non-concurrent transactions, thereby guaranteeing strict-serializability [12, 26].

Transactions in SunStorm may be single-region (SR), or multi-region (MR). The SSI algorithm at each compute node serializes single-region transactions, called T_s. However, an MR transaction, called T_m, is initiated at an arbitrary region, a, and may operate on both local and remote data. Call the operations to the local data $T_m@a$, which adhere to the SSI rules for SR transactions. Operations on remote data that belongs to a remote region $b \neq a$ execute speculatively on a snapshot of the remote region, CSN_b.

Once the 2PC process begins, the validation transaction at region b, $T_m@b$ validates the speculative read and write operations. As described in Section 4.4.2, successful validation guarantees that transactions which committed after the CSN_b did not write to any data items in $T_m@b$'s read or write sets. If the CSN of the last transaction that committed at region b before $T_m@b$ started is CSN'_b, then any transaction that committed at region b between CSN_b and CSN'_b, called T_x, will not produce **wr** or **ww** conflicts with $T_m@b$. Further, a **rw**-anti-dependency[4] of the form $T_m@b \xrightarrow{rw} T_x$ will not be a part of the conflict graph at region b. However, T_x may

[4]Transactions produce a **rw**-anti-dependency or **rw**-conflict of the form $T_i \xrightarrow{rw} T_j$ if T_i reads a version of some data item, and T_j writes a later version of that data item[8].

produce a **rw**-anti-dependency, $T_x \xrightarrow{rw} T_m@b$. Thus, SunStorm's validation process produces a valid MVCC snapshot at CSN'_b.

For a transaction at region b, called T_y, that is concurrent with $T_m@b$, MVCC prevents T_y from producing **wr** or **ww** conflicts with $T_m@b$. However, T_y could produce **rw** conflicts with concurrent transactions. SSI tracks the **rw** conflicts and aborts transactions that produce unsafe structures [24, 41]; ensuring that the conflict graph at region b is acyclic. Further, the MVCC at each each compute node ensures that T_m does not produce **ww** or **wr** conflicts with concurrent transactions. Therefore, similar to single-node MVCC, SunStorm's concurrency control protocol needs to disallow **rw** conflict cycles across regions to be serializable.

Having established that the conjunction of SSI and SunStorm's multi-region validation protocol disallows **rw** cycles within a single region, we now show that the concurrency control protocol described in Section 4.3 disallows **rw** cycles that span multiple regions. We prove by contradiction that a cycle of n transactions: $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} ...T_n \xrightarrow{rw} T_1$ is always disallowed. The cycle cannot contain two consecutive SR transactions as it would create a locally unsafe structure at that region (even if the third transaction in the structure is MR) and would be aborted by the regular SSI checks. Therefore, every pair of adjacent transactions in the cycle must contain at least one MR transaction, which causes the reader and writer rules from Section 4.3 to be enforced. We show that these rules cause either one of transaction of this pair to be aborted or otherwise guarantees a commit order of $Commit(T_i) < Commit(T_{i+1})$.

For any pair of adjacent transactions in the cycle, (T_i, T_{i+1}), call the first one to reach the *PREPARED* state, T_f, and its state when the other transaction attempts to prepare, $STATE(T_f)$. The table below summarizes the outcomes of the reader and writer rules.

T_f	$STATE(T_f)$	Outcome
T_i	*PREPARED*	Writer rule aborts T_{i+1}.
T_i	*COMMITTED*	T_{i+1} is not aborted.
T_i	*ABORTED*	T_{i+1} is not aborted.
T_{i+1}	Any $STATE$	Reader rule aborts T_i.

Among all the outcomes listed above, the only case where both transactions commit is if $Commit(T_i) < Prepare(T_{i+1})$, implying $Commit(T_i) < Commit(T_{i+1})$. In every other case, at least one of the two transactions are aborted, ensuring that the cycle $T_1 \xrightarrow{rw} T_2...T_n \xrightarrow{rw} T_1$ is broken. Therefore for every edge in the graph, if both (T_i, T_{i+1}) commit, SunStorm enforces a commit order of $Commit(T_i) < Commit(T_{i+1})$. Generalizing the order to n transactions yields the commit order $Commit(T_1) < Commit(T_2) < ... < Commit(T_n)$. Since T_1 is prepared before any other transaction in the graph, and the graph contains $T_n \xrightarrow{rw} T_1$, T_n is always aborted by the writer rule; thereby breaking the cycle. Thus, SunStorm's conflict graph is acyclic. Therefore, SunStorm is **serializable**.

If a transaction T_i is committed at a subset of regions, X, it is guaranteed to be prepared at every other region, Y, where it accesses data. If T_i and T_j are two conflicting transactions, and T_j arrives after T_i commits at region $x \in X$, then $T_i@x$'s updates will be visible to T_j since they execute at the same region. Further, if T_i and T_j overlap at a region $y \in Y$, where T_i is still prepared, the reader or writer rule will abort T_j. Hence, T_j is either aborted or views all of T_i's effects, ensuring $T_i < T_j$ in the equivalent serial order. Thus, SunStorm is **strictly serializable**.

6 IMPLEMENTATION

We leverage Neon [5] to build a prototype of SunStorm. Neon is an open-source system based on the original single-primary version of Amazon Aurora, using a modified version of PostgreSQL. Its page store and WAL store are named *pageserver* and *safekeeper*, respectively. Our implementation builds a multi-primary, geographically replicated database system over Neon by taking advantage of Neon's data branching functionality. Each geo-partition is a separate Neon branch. Although most of our code is a layer above Neon (including the transaction server discussed in Section 4), we made some changes to Neon's pageserver and core PostgreSQL code.

6.1 Geo-partitioning

SunStorm supports data partitioning at both table and row levels. SunStorm leverages PostgreSQL's partitioned tables that act as logical containers to which child tables are attached. Queries can access these child tables through the parent table as if they form a single cohesive table. We added a column to the relation catalog to indicate the region that owns each relation and used Neon's data branching feature to physically partition data across regions.

In Neon, a new branch inherits all data from its parent branch up to the branching point. A compute node can be created and assigned to the new branch. It can write data to its assigned branch, allowing the WAL to progress independently from its parent.

SunStorm puts every partition in a different branch, which is owned by the corresponding region. It uses an extra branch to store the catalogs which can be owned by any region. A region only writes to its branch but may read from all branches.

6.2 Pageserver

Each safekeeper stores the WALs of all branches, and a pageserver can subscribe to branches. In SunStorm, each region's pageserver connects to the safekeeper in every other region and subscribes to its branch so that it can serve data belonging to any region.

6.3 PostgreSQL

The current implementation of SunStorm uses PostgreSQL version 14.9. We first overview relevant internal details of PostgreSQL, and then describe the changes we made to the PostgreSQL codebase.

PostgreSQL employs multi-version concurrency control (MVCC). Each tuple version includes metadata: *xmin*, the transaction identifier (*xid*) of the transaction that created it, and *xmax*, the *xid* of the transaction that replaced it with a newer version (if any). This metadata is used along with the *commit log* (called `pg_xact` which contains commit/abort status of all transactions) to determine the visibility of a tuple version for any given snapshot.

Most of our changes reside within a PostgreSQL extension. However, modifications to the PostgreSQL kernel are still necessary for SunStorm to function. In making decisions to altering it, we aimed to minimize our changes by leveraging existing facilities in the codebase. As a result, we modified approximately 3800 lines of code (with comments), among over a million lines of code, in the kernel.

6.3.1 Commit sequence number. As outlined in Section 3, SunStorm assigns a CSN to every data item modified by a transaction. At first glance, it might appear that PostgreSQL already fulfills this requirement with each tuple's *xmin* and *xmax*. However, they are based on *xid*s assigned by PostgreSQL on the first write operation, so these *xid*s do not necessarily follow the commit order.

Therefore, we introduced a *CSN log* (`pg_csn`), which utilizes the same data structure underlying the `pg_xact`, and maps an *xid* to its CSN. A transaction's CSN is undefined if it is either in progress or aborted. Otherwise, its CSN is assigned to be the LSN generated by appending the transaction's commit record to the WAL. A side benefit of using the LSN is that no additional WAL record is necessary to persist `pg_csn`. While replaying another region's WAL records, the *pageserver* continuously updates its `pg_csn` mapping based on commit records so that it can instantly provide this information to its local compute node when needed.

6.3.2 Remote data. Each region's PostgreSQL instance at the compute node assigns *xid*s independently (which are used as *xmin* and *xmax* in tuple snapshot metadata). Therefore, snapshot metadata from different regions cannot be directly compared. When a region brings in a remote page, it will initially contain snapshot metadata meaningful only at that remote region. Nonetheless, even though the *xmin* and *xmax* values are not comparable to local *xid*s, it is still possible to determine which version of that remote record is currently visible (if any) via the `pg_csn` structure to associate remote *xmin* and *xmax* values with committed and aborted transactions, along with determining the commit order amongst them. If a region creates any new versions on a remote page, the snapshot metadata for these new versions will now be meaningful only locally. The remote region will eventually rewrite these values during validation, but while the page is temporarily written locally, two bits in the tuple header are used to indicate that the current snapshot metadata is now referring to local *xid*s rather than remote *xid*s.

6.3.3 Read/Write set collection, and performing remote writes. PostgreSQL detects **rw**-conflicts via special non-blocking SIREAD locks on data items read, which may be relations, pages, or tuples. We take advantage of these SIREAD locks acquiring functions by incorporating calls to our read collection functions that add the identifiers of these data items to the read set. Writes in PostgreSQL are performed by a few low-level functions responsible for inserting, updating, or deleting tuples in a relation. Our implementation invokes write collection functions within these functions to generate the write set. Remote writes in the write set are represented **logically** using PostgreSQL's logical replication message format. During the commit protocol described in Section 4.4, these writes are sent to the remote region and converted to physical writes upon validation.

6.3.4 Deadlock prevention. PostgreSQL uses locks to prevent concurrent writes and detects and resolves deadlocks. However, multi-region transactions may cause distributed deadlock, which PostgreSQL cannot detect. We prevent this with a wait-die scheme. Since a distributed deadlock must involve at least two multi-region transactions, the scheme can be limited to that scenario: a multi-region transaction (both initiator and validator) is aborted if it is blocked by another multi-region transaction during a tuple update.

6.3.5 Serialization Checks. SunStorm implements the additional serialization checks as a part of PostgreSQL's SSI modules. It makes two optimizations of the rules from Section 4.3. First, when the writer transaction in a multi-region **rw**-anti-dependency is the first to commit, it immediately aborts the reader, simplifying the implementation of the reader rule. Secondly, PostgreSQL instantaneously transitions SR transactions from the *PREPARED* to *COMMITTED* state. Therefore, SR transactions directly transition from the *ACTIVE* to *COMMITTED* state and SunStorm only needs to enforce the writer rule when the reader is a MR transaction.

7 EVALUATION

We evaluated the performance of SunStorm and compared it to state-of-the-art systems from the two schools of thought described in Section 1: the single-primary "Aurora-style" architecture and the shared-nothing architecture. For the single-primary architecture, we use Amazon Aurora PostgreSQL (referred to as Aurora for brevity). For the shared-nothing architecture we utilized CockroachDB and YugabyteDB. Both systmes support geo-partitioning similar to SunStorm, while Aurora does not. Furthermore, SunStorm, CockroachDB and YugabyteDB support multiple writer nodes, while Aurora has a single primary node that handles writes. Aurora's Global Write Forwarding feature can be used to enable writes at secondary regions by forwarding write statements to the primary instance. However, only having a single writer node can be a scalability bottleneck. Although Amazon has several projects that attempt to fix the scalability bottleneck—Aurora Limitless [54] and Aurora DSQL [43]. Aurora Limitless fixes the scalability bottleneck using multiple shards within a single region, thus not addressing the latency issues in the multi-region settingBesides, it requires a minimum of 16 shards, making it impractical to produce equivalent setups for other systems. While Aurora DSQL [6] supports multiple writers across regions, it is currently under preview and only available in two regions. However, unlike other systems in our evaluation, it only guarantees `REPEATABLE READ` isolation level.

7.1 Experimental Setup

Each experiment runs in multiple AWS regions (specified per experiment). In each region, CockroachDB and YugabyteDB are deployed in a three-node cluster of c5d.4xlarge machines[5], which is among the recommended instance types for these systems [3, 7]. SunStorm, with separate compute and storage layers, is architected differently. To enable a fair comparison across different architectures, we used CockroachDB's hardware as a cost baseline and chose SunStorm's instance types accordingly: a c5.4xlarge instance for the compute layer, a c5d.4xlarge for the page store, and 3 c5.2xlarge instances for the WAL store. Similarly, Aurora also has separate storage and compute layers. Information regarding the hardware used for Aurora's storage layer is not publicly accessible. For the compute layer, Aurora provides a limited list of instance-type options. We opted for the db.r5.4xlarge instance, which most closely matches the compute node specifications of the other systems. Although each system uses a different architecture and has a distinct commit protocol, all four systems provide equal availability by replicating data to

[5]These instances run on Intel Xeon Platinum 8000 series processors with clock speeds of 3.6GHz, and have 16 vCPUs with 32GB of memory, and a 400GB NVMe-based SSD.

multiple nodes before committing a transaction. By default, Aurora relies on write quorums of size 4/6 while SunStorm, CockroachDB, and YugabyteDB in our deployment use a replication factor of 3.

We used BenchBase (formerly known as OLTPBench) [20] to generate the workloads. We deployed one client machine in each region and provisioned enough capacity for them to avoid client-side bottlenecks. Every client thread issues transactions in a closed-loop. A transaction can either be single-region (SR) or multi-region (MR). Both SR and MR transactions access at least one data item in the local region, while MR transactions access at least two regions. Aurora does not partition data across regions and simply forwards writes to the primary region, which is us-east-1 in all experiments. We executed the transactions at the highest possible isolation level for all systems; however, Aurora only supports the REPEATABLE READ isolation level in the secondary regions with write forwarding. We executed experiments with both consistency options provided by Aurora: strong (GLOBAL) and eventual (EVENTUAL) consistency.

7.2 Microbenchmark

In this experiment, we deployed the systems across three regions: us-east-1, eu-west-1, and ap-northeast-1. We evaluated the systems using a microbenchmark based on the Yahoo! Cloud Serving Benchmark (YCSB) [17], adapted for transactions. The benchmark involves one table with 11 columns. The first column, containing 64-bit integers, serves as the primary key, while each of the other columns consists of 10 random bytes. The table is partitioned over the regions, with each partition containing one million rows.

Each transaction accesses 8 rows from the table without retries for aborts due to errors. We used three YCSB workloads: *Read-Heavy* (95% read, 5% write), *RMW* (50% read-modify-write, 50% read), and *Read-Only* (100% read). For an MR transaction, the number of regions accessed follows a Zipfian distribution and the rows accessed are divided evenly among them. The data within each region is separated into a *hot set* and a *cold set*. Every transaction accesses one row from the hot set in each region involved. We define HOT to be the number of tuples in each region's hot set. Consequently, contention increases as the HOT value decreases.

For the first set of experiments, we varied the % MR parameter at 0%, 5%, 10%, 15%, and 50%, each of which under two HOT settings: 100,000 and 1,000. Fig. 4 summarizes the results of the experiments.

7.2.1 Throughput. The throughput results are presented in first row of Fig. 4. Aurora's throughput stays flat since it does not support geo-partitioning and is thus unaffected by the %MR parameter. Yet, the single-writer limits its throughput and is thus bottlenecked by the speed of the node's writes. Hence, its throughput is much higher for the read-heavy workload than the *RMW* workload. The throughput for the two consistency settings are similar since the primary cost of improved consistency is latency (not throughput). However, the GLOBAL workload performs additional work to verify that the secondary is up-to-date with all the changes committed at the primary at the moment when a transaction began. Therefore, the secondary nodes notice a slight drop in throughput with the GLOBAL setting. This drop is noticeable in the *Read-Heavy* workload, which can better utilize these secondary nodes.

YugabyteDB and CockroachDB support geo-partitioning, and therefore, their throughput increases with better partitioning and fewer MR transactions. However, their shared-nothing architecture limits their throughput. Since they partition data across all nodes even **within** a region, they pay distributed processing costs even for SR transactions that access data on multiple nodes within a region. Although both systems perform similarly, CockroachDB slightly outperforms YugabyteDB for the Read-Heavy workload when there are few MR transactions. This appears to be because YugbyteDB routes SQL queries through its PostgreSQL backend before accessing data via the underlying storage engine, while CockroachDB uses its servers to emulate both the execution and storage engines. This increased efficiency is most noticeable when the overhead of distributed transaction processing is not present.

In contrast, SunStorm only pays distributed processing, coordination, and commit costs for MR transactions. Therefore, it achieves a large jump in throughput where there are fewer MR transactions since it benefits by having multiple writer nodes (one per region) without paying the cost of coordination across regions.

Thus, for workloads that partition well across regions, SunStorm yields all the benefits of geographic partitioning while also yielding the benefits of the Aurora-style approach of avoiding distributed transactions. However, for workloads that do not partition well, standard Aurora is preferable (especially for read-heavy workloads in which the write-scalability bottleneck is less problematic),while SunStorm requires cross-region communication, validation, and two-phase commit. Furthermore, as contention increases, abort rate increases (see Section 7.2.2), decreasing the throughput further – causing it to perform worse than the shared-nothing systems at ($MR = 50\%$). These results are consistent with the expectation that optimistic techniques work best when contention is low [30].

Irrespective of the workload, at MR = 0%, SunStorm yields more than three times the throughput of Aurora, despite only having three times the processing resources for write transactions because all writes in Aurora are processed at a single node. Although the SR transactions within a region do not conflict with SR transactions from another region, they still contend with each other for shared database resources, such as the global data structure that stores the set of active transactions and the writer thread that flushes transaction log records. Therefore Aurora's design of processing all writes at a single primary creates resource contention while serving a larger traffic, leading to a throughput drop.

7.2.2 Abort rate. The second row in Fig. 4 illustrates the abort rate as a percentage of all submitted transactions. DMBS-initiated aborts in SunStorm may occur due to the DBMS's native concurrency control or failed validation (in case of MR transactions). In contrast, Aurora, CockroachDB, and Yugabyte only abort transactions due to the DBMS's concurrency control mechanism.

SunStorm and Aurora with EVENTUAL consistency abort SR transactions (MR = 0%) at similar rates. However, with GLOBAL consistency, Aurora faces a higher abort rate on the *RMW* workload due to increased latency, resulting in an extended conflict window. YugabyteDB experiences a higher abort rate for all cases of SR transactions. This is due to YugabyteDB's choice of optimistic concurrency control techniques as well as the practice of applying back pressure on writes, this avoids bloating of the in-memory layer. In contrast, CockroachDB uses a pessimistic approach by locking keys for writes and, therefore, experiences fewer aborts.

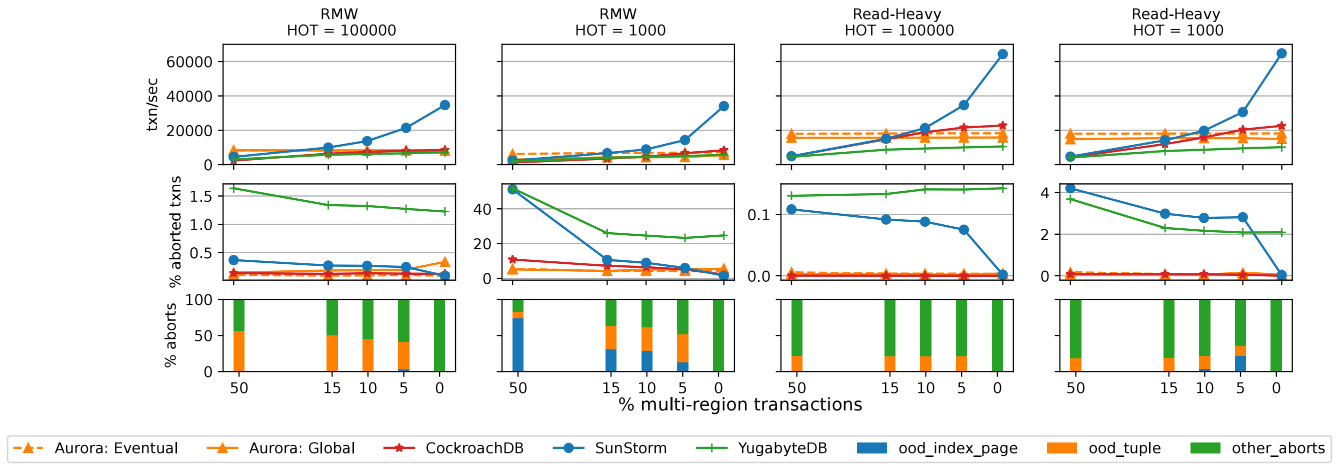


Figure 4: Microbenchmark performance

Similarly, MR transactions in SunStorm and YugabyteDB are more susceptible to aborts because they take longer to complete. While their abort rates are lower than 2% at low contention, they increase with contention as the probability of a conflicting transaction existing during the contention window increases. SunStorm's abort rate is generally lower than YugabyteDB's because it requires fewer rounds of communication for MR transactions. As discussed in Section 4.4, SunStorm only communicates with other regions at commit time, and uses a decentralized 2PC protocol that requires a single round-trip between the initiator and the validators. A shorter execution and commit duration leads to a smaller window where an uncommitted remote transaction can cause other transactions to abort. There is no notion of a MR transaction in Aurora.

For the Read-Heavy workload, all systems have a low abort rate ($\leq$ 4%), irrespective of contention level. This is due to the small percentage of write transactions, ideal for OCC protocols [30].

7.2.3 Abort reasons. To better understand the effects of SunStorm's validation mechanism, we measured the different causes of aborts for SunStorm transactions. We track three main causes: (1) outdated index pages, (2) outdated tuples, and (3) other aborts by PostgreSQL. Only the first two are caused by SunStorm validation. No aborts were caused by distributed deadlocks in our experiments. The results are shown in the last row of Fig. 4. As expected, validation aborts take a greater share of abort percentage as the %MR increases.

Under low contention, all SunStorm validation aborts are caused by outdated tuples instead of indexes due to the *Heap-Only-Tuple* optimization employed by PostgreSQL that skips index updates when tuple updates do not involve indexed columns and the old tuple's page contains sufficient space for the new tuple (which is usually true under low contention and updates are spread out across all pages). However, under high contention, the pages with HOT tuples get updated repeatedly, causing the need to spill over to new pages, ultimately causing outdated index aborts instead of outdated tuple aborts, since outdated index pages are checked first.

7.2.4 Latency. In Fig. 5, we plot the latency data of the systems for two regions (us-east-1 and eu-west-1) for `HOT` = 100000 and `HOT` = 1000, representing low and high contention cases, respectively, for the MR = 5% and MR = 50% workloads. Latency for ap-northeast-1 is similar to eu-west-1. The x-axis categorizes transactions as n-regions and the y-axis plot the latency in milliseconds. "1-region" refers to SR transactions, while "2-regions" and "3-regions" refer to MR transactions that access data in 2 and 3 regions, respectively.

To summarize the latency, we use a modified box plot, where the upper whisker represents the 99th percentile latency instead of the usual 1.5 times the interquartile range (the lower whisker is the normal 1.5 times interquartile range). Aurora exhibits low latency in its primary region (us-east-1), and high latency in the secondary regions due to write forwarding. Additionally, under `GLOBAL` consistency, secondary nodes incur a cross-region round-trip to obtain the primary's latest state at the start of each transaction, increasing latency. In contrast, SunStorm, YugabyteDB and CockroachDB leverage data access locality through geo-partitioning, to lower latency for SR transactions ('1 region' columns) across all regions. In Aurora's primary region, SunStorm's SR transactions have slightly lower 99th percentile latency, since they do not have to compete for processing resources with transactions from other regions, while in Aurora all transactions are processed by the same primary node.

In general, both YugabyteDB and CockroachDB exhibit similar latency trends across various configurations. While both systems employ geo-partitioning, but have a much higher latency baseline when compared to SunStorm, due to the shared-nothing architecture. Since even SR transactions are processed as distributed transactions across the nodes within the region and must pay the latency associated with distributed execution and commit.

The latency of MR transactions in SunStorm is close to a single round-trip to the furthest participating region. This is because SunStorm accesses local data during transaction execution and communicates across regions only for commit validation, which involves only one round of communication. Conversely, YugabyteDB and CockroachDB need to communicate with other regions for reads of remote data, resulting in possibly multiple round-trips.

High contention adversely affects SunStorm's latency, especially at the tails. This is because some SR transactions may conflict

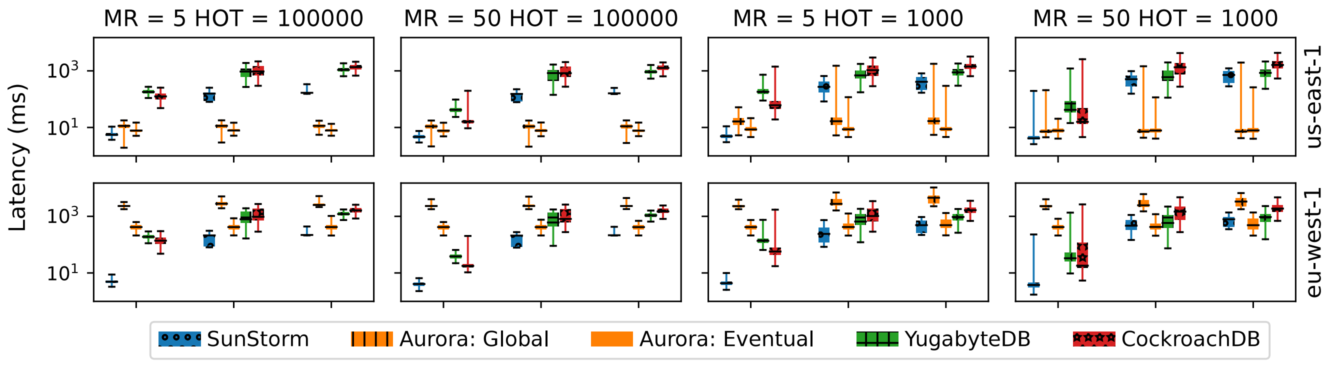


Figure 5: Microbenchmark latency (RMW). Top row is Aurora's primary region. Bottom row is a non-primary region.

with and end up waiting behind slower MR transactions, raising their latency to match that of those MR transactions. While this phenomenon may also occur in YugabyteDB and CockroachDB, the impact is less noticeable since all transactions in these systems generally have much higher latency. Similar to the results from the previous subsection, Aurora's numbers are again only affected by running over multiple regions to the extent that MR transactions may access more hot records. In the primary (`us-east-1`) region, Aurora with the `GLOBAL` setting experiences higher tail latency because the local transactions are more likely to be blocked by the slower transactions originating in the remote regions.

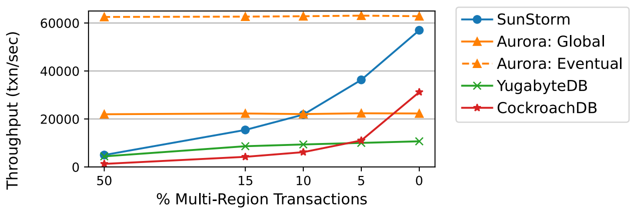


Figure 6: Read-Only workload throughput (HOT = 1000)

7.2.5 Read-Only throughput. Next, we run a Read-Only workload. The throughput results are shown in Fig. 6. Aurora with `EVENTUAL` consistency achieves the highest throughput because the secondary regions operate almost independently in the absence of writes, and its poor consistency guarantees gives it a large advantage relative to the other systems. However, when it uses `GLOBAL` consistency (which is more comparable to the other systems), it runs much slower because the secondary nodes contact the primary at the start of every transaction and wait for it to respond with the last persisted LSN. SunStorm's throughput at MR= 50% is comparable to the Read-Heavy workload, as all MR transactions are validated even if they are read-only (see Section 4.4). However, at MR= 0%, SunStorm's throughput almost matches Aurora under `EVENTUAL` consistency because then it has negligible validation overhead across regions. YugabyteDB and CockroachDB's performance are similar to their performance on the Read-Heavy workloads since Read-Only transactions still follow the same execution protocol and require coordination within a region. Cockroach performs significantly better at MR= 0% because of YugabyteDB's reliance on PostgreSQL backends as opposed to CockroachDB's tablet server approach.

7.3 TPC-C

Next, we compare the systems using the TPC-C benchmark [1]. We partitioned the data by warehouse ID and evenly distributed the warehouse data across the us-east-1, eu-west-1, and ap-northeast-1 regions. We ran the experiment with 30 and 300 total warehouses; each region owning 10 and 100 warehouses, respectively. 1% of `NewOrder` and 15% of `Payment` transactions involve "remote" warehouses, defined as any warehouse different from the main one. Given that `NewOrder` and `Payment` transactions account for 45% and 43% of the traffic, respectively, remote transactions constitute 6.9% of the workload. We change this definition to specifically mean a warehouse in a different physical region, making these transactions multi-region. Transactions have no "thinking time" or "keying time" and can retry up to twice in case of a system-initiated abort.

We ran the experiment by progressively increasing the number of clients until each system reached saturation. In Fig. 7, the x-axis represents the total throughput across all transaction types, while the y-axis shows the latency of `Payment` transactions. The latency trends of other transaction types are similar and thus not shown. Each successive point (per line) in the graph corresponds to an increase in the number of clients. We present both p50 and p99 latency for each system. Since 15% of the Payment transactions in the multi-writer systems are MR, the p99 latency serves as a representative measure for the MR transaction latency. To make the graph easier to read, the graph shows Aurora only for the `GLOBAL` setting, since that configuration matches more closely to the consistency level of SunStorm and YugabyteDB, and the difference between `GLOBAL` and `EVENTUAL` is similar to the previous experiments.

In general, SunStorm consistently achieves higher peak throughput than other systems across all regions. Although YugabyteDB and CockroachDB lag behind Aurora's peak throughput in the us-east-1 region, they maintain a similar throughput level in the eu-west-1 region. The decrease in Aurora's throughput in its secondary regions is due to each write and final commit within a transaction incurring a network round-trip. However, YugabyteDB's throughput drops to the same level as Aurora's in the ap-northeast-1 region

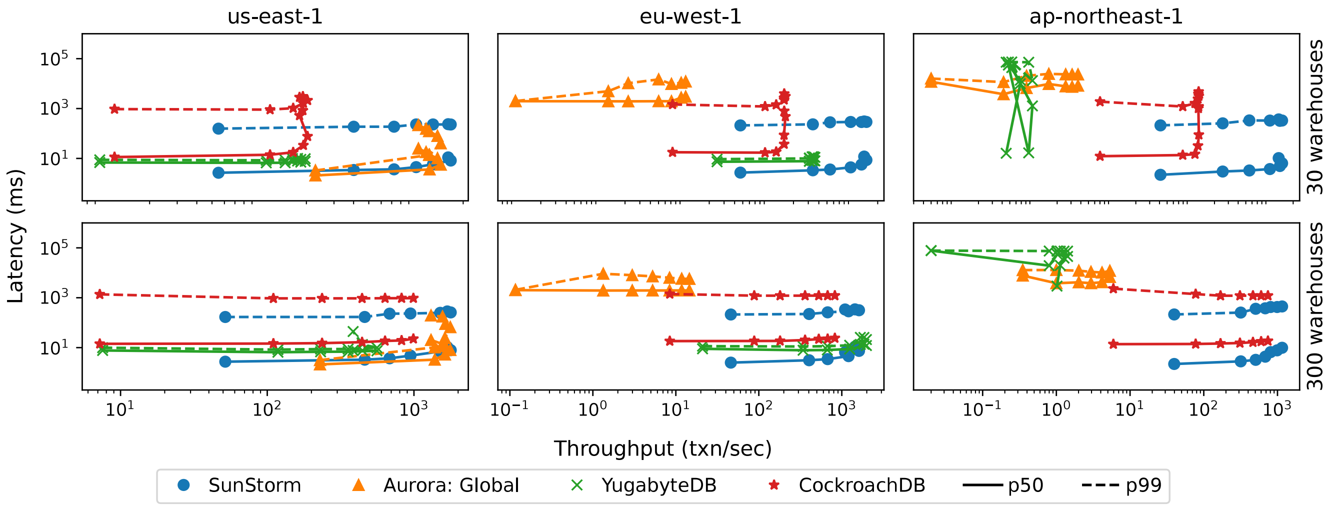


Figure 7: TPC-C Payment Transaction Performance

due to the much higher latency connections to the other two regions, which slow down YugabyteDB's cross-region reads that may occur multiple times throughout its MR transaction. CockroachDB, however, achieves better throughput in ap-northeast-1, producing consistent throughput across regions. We attribute this behaviour to CockroachDB's reliance on write locks as opposed to YugabyteDB's default optimistic concurrency control techniques.

Similarly, SunStorm's validation protocol emulates S-2PL for MR transactions, leading to consistent throughput across regions. However, unlike CockroachDB, MR transactions in SunStorm require exactly one round trip across regions, resulting in lower latency.

Since the majority of transactions in TPC-C are SR, all systems except Aurora have low median latency in every region. In line with the microbenchmark experiment, MR transactions in distributed systems are slower than those of Aurora in us-east-1 (its primary region) but outperform Aurora in other regions.

Increasing the number of warehouses reduces contention in the TPC-C benchmark. Therefore, the overall performance of geo-partitioned systems improves when the number of warehouses is increased to 300. However, Aurora's scalability is limited by relying on a single primary instance, resulting in smaller improvements.

7.4 Scalability

We evaluate the scalability of the systems by running the RMW workload with 0% and 5% MR with `HOT` at 10000. We expanded the deployment to 6 regions: us-east-1, eu-west-1, ap-northeast-1, us-west-1, eu-central-1, and ap-southeast-1. Fig. 8 shows that all systems, except Aurora, scale linearly with additional regions. However, each SunStorm region contributes significantly more to the overall throughput capacity of the system because SR transactions execute entirely at their primary region and are generally unaffected by the number of regions in the deployment. SunStorm's throughput drop between the MR = 0% and MR = 5% cases is consistent with the results from Section 7.2, where cross-region coordination for MR transactions requires a single round trip between the initiator and the validating regions which reduces throughput. However, the drop is slightly larger for this experiment, since transactions may access data across all 6 deployed regions, as opposed to 3 regions used in Section 7.2. In summary, SunStorm scales (almost) linearly when adding more regions, effectively bridging the gap between the shared-nothing and shared-storage schools of thought.

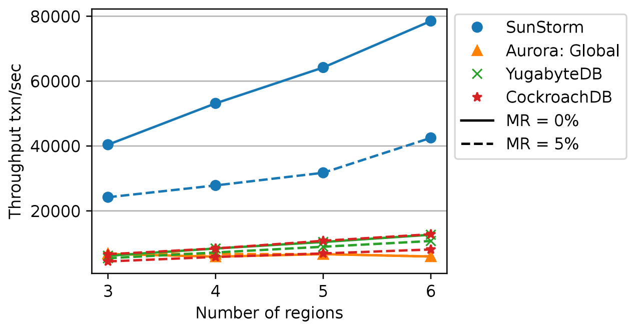


Figure 8: Scalability results (HOT = 10000)

7.5 Comparison with Deterministic Databases

Next, we evaluate SunStorm against Detock [38], which uses a shared-nothing architecture similar to CockroachDB but relies determinism to yield better throughput, especially under high contention. We use the same deployment as the shared-nothing systems in Section 7.1. Detock does not support the JDBC protocol, which means we cannot use the Benchbase tool [20] to send it the experimental workload. Instead, the workload is generated internally, sidestepping much of the client-server communication overhead. This, along with the fact that it is only a key-value store that does not support full DBMS functionality (indexes, constraint checking, etc) makes it hard to make an apples-to-apples comparison. We therefore normalize the results to compare relative trends across

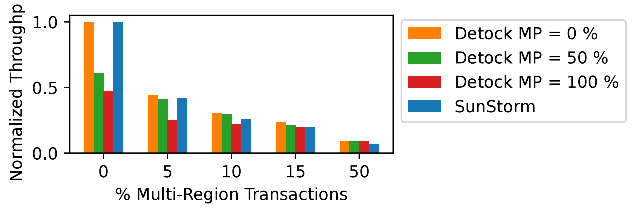


Figure 9: Normalized Read-Modify-Write performance

systems, similar to the comparisons in [38, 42]. We executed the high-contention Read-Modify-Write (RMW) microbenchmark described in Section 7.2. Fig. 9 shows the results of this experiment. Similar to the other shared-nothing systems, Detock suffers from the presence of multi-partition transactions. Thus, SunStorm's primary advantage relative to shared-nothing systems is also present vs. Detock. However, Detock's focus on high throughput under contention is clear in this high contention experiment, allowing it to scale better as the percentage of multi-region transactions increase and SunStorm suffers from increased validation failures.

8 RELATED WORK

Scaling write throughput in a shared-storage system can be done by supporting a multi-master setup, allowing different compute nodes to modify the same set of data items in the storage layer. Systems using this method exist both commercially (IBM Db2 pureScale, Oracle RAC, PolarDB-MP, Aurora Limitless) and academically (Starry and Taurus MM). IBM Db2 pureScale [4] and Oracle RAC [15, 40] transfer modified pages over the network instead of log records, causing high network utilization.

Fundamentally, there are two approaches to improve the throughput of Aurora-style systems: sharding and ownership transfer. Examples of the first approach are PolarDB-MP [51] and Aurora Limitless [54], which deploy the primary shards in a single geographic region, while Aurora DSQL [6], PolarDB-X [13], and Starry [56], which support deployments across multiple geographic regions. PolarDB-MP [51] relies on a disaggregated shared memory (and storage) to provide all nodes with data access, but its reliance on high-bandwidth networks [4, 51] makes it unsuitable for multi-region deployments. Aurora Limitless [54] uses a group of data access nodes within a region, each owning a portion of sharded tables or full copies of reference (non-sharded) tables. However, writes from secondary regions are still forwarded to the primary, incurring high latency. PolarDB-X's [13] deploys multiple single-primary systems across data centers (DC) and uses hybrid logical clocks to provide snapshot isolation. However, it is optimized for running in DCs within close proximity (1ms round-trip time), and only supports snapshot isolation. Starry [56] and Aurora DSQL [43] enable database nodes to process transactions independently by relying on OCC to order and validate transactions. Starry allows each database node to send transactions to its local storage replica and uses a single centralized sequencer to resolve conflicts, making it unsuitable for multi-region deployments. Aurora DSQL supports multiple writers across regions by validating writes on a per-log entry basis. Similar to SunStorm, it incurs single round-trip latency per transaction, irrespective of the number of remote operations. However, it does not support strict serializability.

Taurus MM [19] and GaussDB [33] are examples of the second approach. Taurus MM uses vector-scalar clocks for the masters to see a consistent state and employs a hybrid page-row locking mechanism, which involves a global lock manager, making it unsuitable for multi-region deployments. GaussDB [33] eliminates the need for a global lock manager by persisting page lock information in directory pages. However, it relies on RDMA for communication.

SunStorm, PolarDB-X, Aurora DSQL all avoid Aurora's write-scalability bottleneck via partitioning, and are thus sensitive to partition skew (although Aurora DSQL supports multiple writers per region, the single adjudicator per region limits a region's throughput). In such cases, integration of ownership-based approaches, such as that found in GaussDB, can improve performance.

The shared-nothing architecture, which has long been promoted as the go-to architecture for scaling OLTP workloads [44], has seen extensive research in geo-distributed transactions for both fully-replicated systems [18, 23, 29, 34, 36, 46, 50, 53, 55] and partially-replicated (geo-partitioned) systems [2, 16, 21, 27, 38, 42, 45]. The transaction execution methods of these systems fall into two camps: non-deterministic and deterministic. The systems in the first camp typically combine 2PC and a consensus protocol [31, 32, 39] to commit transactions at the end of their execution [2, 16, 18, 21, 23, 29, 34, 45, 50, 53, 55], while those in the second camp deterministically execute transactions based on an order pre-established by a consensus protocol [27, 36, 38, 42, 46]. Both approaches are competitive in terms of latency. However, deterministic systems can handle more transactions per second especially under high contention workloads, but further research is needed to enhance their transaction expressivity. SunStorm falls in the non-deterministic camp as its use of traditional DBMSs aligns better with this approach.

To the best of our knowledge, SunStorm is the first DBMS to scale write throughput in an Aurora-style shared-storage architecture that supports strict serializability across multiple regions. It uses the shared-storage architecture for single-region transactions and applies shared-nothing techniques for multi-region transactions.

9 CONCLUSION

SunStorm enables geographic partitioning in a shared storage architecture. Since SunStorm routes single-region transactions to the nearest primary node, it reduces the observed latency and produces the combined throughput of multiple writers. Hence, for well-partitioned workloads, it achieves all the advantages of Aurora-style systems by avoiding distributed transaction processing or coordination within a region, along with lower latency and higher scalability. Furthermore, SunStorm guarantees strict serializability while limiting the coordination overhead to a single round-trip.

ACKNOWLEDGMENTS

This work was supported by Neon Inc. and the National Science Foundation under grants DGE-1840340 and IIS-1910613. We would like to thank Movead Li, the original contributor of CSN based snapshots proposal in PostgreSQL.

REFERENCES

[1] 2010. TPC Benchmark C. https://www.tpc.org/tpcc/. Accessed: 2024-2-22.

[2] 2016. YugabyteDB. https://www.yugabyte.com/. Accessed: 2024-2-26.

[3] 2016. YugabyteDB - Deployment checklist. https://docs.yugabyte.com/preview/deploy/checklist/#amazon-web-services-aws. Accessed: 2024-2-18.

[4] 2019. Transparent Application Scaling with IBM DB2 pureScale. White Paper.

[5] 2021. Neon — Serverless, Fault-Tolerant, Branchable Postgres. https://neon.tech/. Accessed: 2024-2-3.

[6] 2025. *Amazon Aurora DSQL (preview)*. https://aws.amazon.com/rds/aurora/dsql/ Accessed: 2025-1-5.

[7] 2025. *Deploy CockroachDB on AWS EC2*. https://www.cockroachlabs.com/docs/v24.3/deploy-cockroachdb-on-aws.html Accessed: 2025-1-5.

[8] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[9] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1743–1756. https://doi.org/10.1145/3299869.3314047

[10] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

[11] Philip A Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221. https://doi.org/10.1145/356842.356846

[12] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading.

[13] 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, and Junbin Kang. 2022. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2859–2872. https://doi.org/10.1109/ICDE53745.2022.00259

[14] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2477–2489. https://doi.org/10.1145/3448016.3457560

[15] Chandrasekaran and Bamford. 2003. Shared Cache - The Future of Parallel Databases. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, Vol. 0. 840. https://doi.org/10.1109/ICDE.2003.1260883

[16] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 210–227. https://doi.org/10.1145/3447786.3456238

[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing* (Indianapolis Indiana USA). ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[18] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J J Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (Aug. 2013), 1–22. https://doi.org/10.1145/2491245

[19] Alex Depoutovitch, Chong Chen, Per-Ake Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, and Calvin Sun. 2023. Taurus MM: Bringing Multi-Master to the Cloud. *Proceedings VLDB Endowment* 16, 12 (Aug. 2023), 3488–3500. https://doi.org/10.14778/3611540.3611542

[20] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: an extensible testbed for benchmarking relational databases. *Proceedings VLDB Endowment* 7, 4 (Dec. 2013), 277–288. https://doi.org/10.14778/2732240.2732246

[21] Tamer Eldeeb, Philip A Bernstein, Asaf Cidon, and Junfeng Yang. 2024. Chablis: Fast and General Transactions in Geo-Distributed Systems. In *CIDR*.

[22] Tamer Eldeeb, Xincheng Xie, Philip A Bernstein, Asaf Cidon, and Junfeng Yang. 2023. Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 343–360.

[23] Hua Fan and Wojciech Golab. 2019. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *Proceedings VLDB Endowment* 12, 11 (July 2019), 1471–1484. https://doi.org/10.14778/3342263.3342627

[24] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* 30, 2 (June 2005), 492–528. https://doi.org/10.1145/1071610.1071615

[25] R Guerraoui and A Schiper. 1995. The Decentralized Non-Blocking Atomic Commitment Protocol. In *Proceedings.Seventh IEEE Symposium on Parallel and Distributed Processing*. IEEE, 2–9. https://doi.org/10.1109/SPDP.1995.530658

[26] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[27] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. 2024. Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems. *Proceedings VLDB Endowment* 17, 3 (Jan. 2024), 469–482. https://doi.org/10.14778/3632093.3632109

[28] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Matei Zaharia, and Xiangyao Yu. 2023. Epoxy: ACID transactions across diverse data stores. *Proceedings VLDB Endowment* 16, 11 (July 2023), 2742–2754. https://doi.org/10.14778/3611479.3611484

[29] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2465351.2465363

[30] H T Kung and J T Robinson. 1981. On Optimistic Methods For Concurrency Control. *ACM Transactions on Database Systems* (1981), 351–351. https://doi.org/10.1109/vldb.1979.718150

[31] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[32] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (Dec. 2001), 51–58.

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

[34] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-latency multi-datacenter databases using replicated commit. *Proceedings VLDB Endowment* 6, 9 (July 2013), 661–672. https://doi.org/10.14778/2536360.2536366

[35] C Mohan. 1990. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. *VLDB J.* (1990). https://doi.org/10.5555/94362.94465

[36] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 517–532. https://doi.org/10.5555/3026877.3026917

[37] Ravi Murthy and Gurmeet Goindi. 2022. AlloyDB for PostgreSQL intelligent scalable storage. https://cloud.google.com/blog/products/databases/alloydb-for-postgresql-intelligent-scalable-storage. Accessed: 2024-2-3.

[38] Cuong D T Nguyen, Johann K Miller, and Daniel J Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. *Proc. ACM SIGMOD Int. Conf. Manag. Data* 1, 2 (June 2023), 1–27. https://doi.org/10.1145/3589293

[39] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. *USENIX* (2014). https://doi.org/10.5555/2643634.2643666

[40] Oracle. 2001. Oracle RAC. https://www.oracle.com/database/real-application-clusters/. Accessed: 2024-2-29.

[41] Dan R K Ports and Kevin Grittner. 2012. Serializable snapshot isolation in PostgreSQL. *Proceedings VLDB Endowment* 5, 12 (2012), 1850–1861. https://doi.org/10.14778/2367502.2367523

[42] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. SLOG: serializable, low-latency, geo-replicated transactions. *Proceedings VLDB Endowment* 12, 11 (July 2019), 1747–1761. https://doi.org/10.14778/3342263.3342647

[43] Amazon Web Services. 2024. Introducing Amazon Aurora DSQL. *AWS Database Blog* (2024). https://aws.amazon.com/blogs/database/introducing-amazon-aurora-dsql/ Accessed: 2024-12-1.

[44] Michael Stonebraker. 1985. The case for shared nothing. In *HPTS*.

[45] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery,

New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[46] 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* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[47] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, Andy Woods, and Peyton Walters. 2022. Enabling the Next Generation of Multi-Region Applications with CockroachDB. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2312–2325. https://doi.org/10.1145/3514221.3526053

[48] 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* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[49] X/Open Company Ltd. 1991. *Distributed Transaction Processing: The XA Specification.* Technical Report.

[50] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 231–243. https://doi.org/10.1145/3183713.3196912

[51] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/3626246.3653377

[52] Xinjun Yang, Yingqiang Zhang, Hao Chen, Chuan Sun, Feifei Li, and Wenchao Zhou. 2023. PolarDB-SCC: A Cloud-Native Database Ensuring Low Latency for Strongly Consistent Reads. *Proceedings VLDB Endowment* 16, 12 (Aug. 2023), 3754–3767. https://doi.org/10.14778/3611540.3611562

[53] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proceedings VLDB Endowment* 15, 12 (Aug. 2022), 3385–3397. https://doi.org/10.14778/3554821.3554830

[54] Channy Yun. 2023. Amazon Aurora Limitless Database. https://aws.amazon.com/blogs/aws/join-the-preview-amazon-aurora-limitless-database/. Accessed: 2024-2-29.

[55] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R K Ports. 2018. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.* 35, 4 (Dec. 2018), 1–37. https://doi.org/10.1145/3269981

[56] Zihao Zhang, Huiqi Hu, Xuan Zhou, and Jiang Wang. 2022. Starry: multi-master transaction processing on semi-leader architecture. *Proceedings VLDB Endowment* 16, 1 (Sept. 2022), 77–89. https://doi.org/10.14778/3561261.3561268

[57] Tobias Ziegler, P Bernstein, Viktor Leis, and Carsten Binnig. 2023. Is Scalable OLTP in the Cloud a Solved Problem? *Conference on Innovative Data Systems Research* (2023).