GeoGauss: A Multi-Master Geo-Replicated NewSQL Database

Weixing Zhou†, Qi Peng†, Zijie Zhang§, Yanfeng Zhang†, Yang Ren§, Sihao Li§, Guo Fu†, Yulong Cui†, Qiang Li§, Caiyi Wu†, Shangjun Han†, Shengyi Wang†, Guoliang Li‡
† Northeastern University, China, § Huawei Technology Co., Ltd, ‡ Tsinghua University, China

ABSTRACT

Multinational enterprises conduct global business that has a demand for geo-distributed transactional databases. Existing state-of-the-art databases adopt a sharded master-follower replication architecture. However, the single-master serving mode incurs massive cross-region writes from clients, and the sharded architecture requires multiple round-trip acknowledgements (e.g., 2PC) to ensure atomicity for cross-shard transactions. These limitations drive us to seek yet another design choice. In this paper, we propose a geo-distributed NewSQL database `GeoGauss` with full replica multi-master architecture. To efficiently merge the updates from different master nodes, we propose a multi-master OCC that unifies data replication and concurrent transaction processing on a per-epoch basis. By following a deterministic and optimistic merge rule, `GeoGauss` ensures strong consistency and allows more concurrency with weak isolation, which are sufficient to meet our needs. Our geo-distributed experimental results show that GeoGauss achieves 1.47X-9.33X higher throughput and 2.18X-17.74X lower latency than the state-of-the-art geo-distributed databases on TPC-C benchmark.

KEYWORDS

Geo-distributed; multi-master replication; optimistic concurrency control; coordinator-free; transaction processing

1 INTRODUCTION

Global companies have built their data centers located in many countries worldwide. To support their global business, it is desired to develop a geo-distributed transactional SQL database spread across multiple geographically distinct locations, e.g., many ICT databases used by telecom service providers are deployed under geo-replicated setting. The design goals are towards high availability, strong consistency, and high performance.

High availability is usually achieved by redundant data replication, which is the process of storing the same data copies in multiple geographic zones. Data replication facilitates not only high availability but also geographic locality and read scalability, making data copies close to users at different regions to reduce read latency and to further improve overall data access throughput. Existing state-of-the-art geo-distributed transactional databases, e.g., Google Spanner [34], F1 [49], CockroachDB [55], YugabyteDB [22], and TiDB [47], adopt a *sharded master-follower replication* architecture as shown in Figure 1(a). Data are partitioned into multiple shards according to key range. Each shard is assigned to a single *master* node serving all write/read requests, and it is replicated and placed to multiple geo-distributed *follower* nodes serving only read requests. Due to its single-master architecture, write-write conflicts are gathered in the same worker to be easily coped with. In addition, sharding can disperse write requests to improve write scalability.

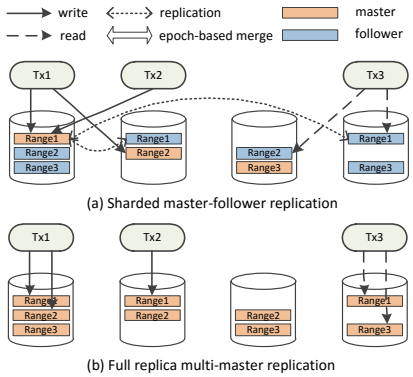


Figure 1: Sharded master-follower replication vs. full replica multi-master replication.

The sharded master-follower replication architecture is widely adopted in production [22, 34, 47, 55], but it suffers from two major drawbacks. 1) The single-master serving mode requires to route the write requests from all clients to the single master node, which leads to cross-region writes and as a result increases transaction latency. Though this drawback can be alleviated by geo-aware partitioning and regional shard placement [55], it still hurts especially for applications without locality property. 2) The sharded architecture relies on two-phase commit (2PC) protocol to ensure atomicity. This requires multiple round-trip acknowledgements between the coordinator and the globally distributed workers, which further hurts performance. 3) The Paxos-based data replication is launched separately from the (SS2PL or time-order based) distributed transaction processing, which brings additional latency, though it has been coupled with the 2PC in some works [34, 55].

Yet another choice for data replication is *full replica multi-master* architecture as shown in Figure 1(b), where each server maintains a full copy of data and all server nodes serve both read and write requests. By placing a full replica in each region, we can serve users with local writes/reads. With a full replica, 2PC can be unnecessary to ensure atomicity. A number of multi-master systems emerge in recent years, e.g., Amazon Aurora [59], Calvin [57], FaunaDB [5], Anna [61], and Aria [41]. However, to employ multi-master replication architecture, there are three key challenges to be addressed.

Challenge 1: Cross-Node Write-Write Conflicts. Different from the master-follower architecture where all writes to the same data are routed to the same node, with multi-master replication, concurrent updates to multiple replicas of the same data can result in cross-node write-write conflicts. Though this could be resolved with distributed two-phase locking protocol, it is too heavy in a geo-distributed setting. We address this challenge by employing *coordinator-free conflict merge* technique [54, 61]. The state updates along with its local timestamp information are exchanged among all

replicas without a global coordinator, and the write-write conflicts are automatically resolved *deterministically* at the receiver side based on a pre-defined rule, e.g., first-write-wins. This coordinator-free execution is well adapted to the geo-replicated scenario, where consistency between replicas is eventually re-established after merging of all updates. However, eventual consistency is not acceptable in most transactional applications.

Challenge 2: Strong Consistency. It is known to be difficult to ensure strong consistency of multiple replicas. The difficulty lies in the fact that the updates arrive at each replica in different orders. Linearizability, as one of the strongest single-object consistency (i.e., all replicas reach to the same state after every operation with real-time constraints), requires expensive coordination, while weak consistency prompts performance without coordination but may incur various anomalies. This reflects the fundamental contradiction between consistency and performance. We address this challenge by introducing *epoch-based merge*, a compromise proposal (between real-time synchronous merge and asynchronous merge) that guarantees consistent snapshot after each *epoch* (e.g., 10 milliseconds). After collecting and merging the updates of an epoch from all nodes, with a *deterministic merge rule* running on every node, a consistent snapshot can be reached on a per-epoch basis.

Challenge 3: Concurrency and Performance. Existing strong consistent multi-master systems (e.g., deterministic databases) rely on ordered locks [57] or dependency graph [41] to ensure *determinism* (i.e., transactions execute in a pre-defined serial order) with *non-deterministic* multi-thread scheduling, limiting the concurrency degree. The mainstream sharded master-follower systems [22, 34, 55] supports *serializability* that executes a transaction's logic as a single unit limiting the performance as compared to weak isolation levels (e.g., read committed) which allow applications more interleavings between conflicting transactions. To address this limitation, we propose *multi-master OCC* that *unifies data replication and optimistic concurrency control*. Master nodes *optimistically* execute their local SQL requests independently and exchange the resulting write sets between each other. The *identical* batch of updates are merged on each node in parallel. It allows for overlapping the read and write sets across transactions with weak isolation and as a result brings more concurrency and performance.

By integrating the above techniques, we build a multi-master geo-replicated database `GeoGauss` based on Huawei's openGauss MOT [28], an in-memory storage engine that is highly optimized for multi-core processors. `GeoGauss` greatly meets the requirements of Huawei's ICT databases for telecomm providers on strong consistency and high throughput where weak isolation is sufficient in most scenarios. To sum up, we list our contributions as follows.

- **Consistent Multi-Master Architecture.** Different from the sharded master-follower architecture that is adopted by state-of-the-art geo-distributed databases, we propose a full replica multi-master architecture that provides strong consistency.
- **High-Performance Multi-Master OCC.** We propose a multi-master OCC that unifies data replication with optimistic concurrency control. It supports multiple weak isolation levels with high concurrency support.
- **Geo-Replicated NewSQL Database `GeoGauss`.** We develop a geo-replicated transactional database `GeoGauss` with full SQL engine by extending Huawei's openGauss MOT engine adapt to geo-distributed environment.

We perform extensive experiments under geo-distributed environment. Our results show that `GeoGauss` achieves 1.47X-9.33X higher throughput and 2.18X-17.74X lower latency than the state-of-the-art geo-distributed database CockroachDB [55] and deterministic databases Calvin [57] and Aria [41] on TPC-C benchmark.

2 BACKGROUND

Geographic data replication systems exhibit three key advantages, high availability, geographic locality, and read scalability. This section reviews existing replicated systems from different dimensions.

2.1 Replicated Data Systems

Table 1 summarizes several replicated systems with their key features. We next study the key techniques used for data replication.

2.1.1 Replication Architectures. There are three main replication architectures, *master-follower*, *multi-master*, and quorum-based *masterless* architecture. A number of production systems employ master-follower replication mainly for redundant backup support. For example, OceanBase [37], PostgreSQL [20], and openGauss [19] all employ the master-follower mode to achieve high availability through their primary-backup setup. However, it is required to route all users' write requests to the single master server, which is unsuited for geo-distributed database where users are spread across regions. With masterless (or leaderless) replication, a user's write/read operation is sent to all replicas, and a quorum protocol is used to avoid stale read by comparing the monotonic version number. For example, DynamoDB [35] and Cassandra [39] both provide high-available key-value (KV) store service through masterless replication. However, all users' write requests are required to be sent to multiple geo-remote servers, which is costly in geo-distributed scenarios. With multi-master replication, all nodes can serve both read and write requests for their local users. It is naturally adapted to geo-distributed requirements. Besides the systems listed in Table 1, there exist other multi-master systems [1–4, 7, 36, 43].

2.1.2 Replication for Different Data Models. Relational SQL databases and NoSQL databases are considerably different in their storage data format, i.e., structured data in SQL databases and KV in NoSQL databases. Accordingly, the replication unit for data replication is different from each other. In relational database, the replication unit is either *binary log* (logical log that stores the corresponding SQL statements) or *redo log* (physical log that stores the changes to database), while in KV database (e.g., HBase [15], Cassandra [39], DynamoDB[35], FoundationDB [6], and Anna [61]), the replication unit is *KV-pair*. Ledger-based blockchain systems are emerged for Byzantine resistance, e.g., Bitcoin [45] and Hyperledger Fabric [27]. Rather than for high availability, the replication in blockchains is mainly for Byzantine-fault tolerance [52] where a block of transactions is replicated between nodes. In deterministic databases, such as Calvin [57] and Aria [41], *a batch of SQL statements* are replicated among nodes for batched deterministic execution. Both of them provide simplified CRUD interfaces rather than a full SQL engine.

2.1.3 Ordering and Consistency of Replicas. With master-follower architecture, some databases use replication only for backup and

Table 1: Comparison of replicated systems

System	Model	Storage	Replication	Rep. Unit	Ordering	Consistency	CC
openGauss [19]	SQL	disk	master-follower	redo log	TSO	semi-sync. (eventual)	MV2PL
openGauss MOT [28]	SQL	memory	master-follower	redo log	TSO	semi-sync. (eventual)	OCC
TiDB [47]	SQL	disk	master-follower	binary log	TSO	quorum (linearizability)	MV2PL
OceanBase [37]	SQL	disk	master-follower	binary log	TSO	quorum (linearizability)	MV2PL
Spanner [34]	SQL	disk	master-follower	redo log	TrueTime	quorum (linearizability)	MV2PL
CockroachDB [55]	SQL	disk	master-follower	binary log	HLC	quorum (linearizability)	MVTO
HBase [15]	KV	disk	master-follower	HFile	-	semi-sync. (eventual)	MV2PL
DynamoDB [35]	KV	disk	masterless	KV	-	quorum (sequential)	last write wins
Cassandra [39]	KV	disk	masterless	KV	-	quorum (sequential)	last write wins
Anna [61]	KV	memory	multi-master	KV	commutative	CALM/CRDT (causal)	conflict free
Bitcoin [45]	ledger	disk	multi-master	block of txs	POW	quorum (sequential)	serial
Fabric [27]	ledger	disk	multi-master	block of txs	order service	quorum (sequential)	SSI & OCC
MySQL Tungsten [18]	SQL	disk	multi-master	binary log	-	async. (eventual)	prevent conflict
PostgreSQL BDR [10]	SQL	disk	multi-master	redo log	-	async. (eventual)	last write wins
Calvin [57]	SQL*	disk	multi-master	batch of SQLs	deterministic	epoch merge (sequential)	ordered locks
Aria [41]	SQL*	memory	multi-master	batch of SQLs	deterministic	epoch merge (sequential)	dep. graph
GeoGauss (ours)	SQL	memory	multi-master	batch of write sets	deterministic	epoch merge (sequential)	multi-master OCC

recovery (e.g., MySQL [9], HBase [15], and openGauss [19]), where all write and read requests are served by a single master node, so there is no need to tackle with the order inconsistency problem when accepting requests from multiple nodes. To maximize performance, they are configured with asynchronous replication or semi-synchronous replication [13] (using synchronous replication for one of the followers and asynchronous replication for others). This means that the consistency of all replicas may be not guaranteed at a certain time point (eventual consistency).

Another set of mainstream sharded databases with master-follower replication provide serializable isolation and linearizability, which do not allow a stale read after an update in real-time order. For example, Google Spanner [34] relies on *TrueTime* (GPS and Atomic Clock) to achieve global time order, CockroachDB [55] relies on *hybrid logical clock (HLC)*, and OceanBase [37] and TiDB [47] rely on a *centralized timestamp oracle (TSO)* service. The consistency of replicas is guaranteed with quorum-based protocols such as Paxos and Raft.

Many NoSQL KV databases adopt masterless replication, allowing any replica to directly accept write requests from clients, without stable master. User's write requests are sent to w replicas, and the read requests are sent to r nodes in parallel receiving different responses (i.e., the up-to-date value from one node and a stale value from another), requiring $w + r > n$ where n is the total number of nodes. Then newer version numbers are used to determine the up-to-date value. However, this *quorum-based* approach without stable master pays for this in performance. Hence, masterless databases are developed either for use cases that can tolerate eventual consistency or pays for strong consistency in performance, e.g., DynamoDB [35], Riak [12], and Cassandra [39].

For multi-master architecture, MySQL and PostgreSQL both provide tools for cross-site multi-master replication, e.g., MySQL Tungsten [18] and PostgreSQL BDR [10], but they use asynchronous replication and cannot guarantee strong consistency. Blockchain system is a particular multi-master replication system with Byzantine-fault tolerance. The key to reaching consistency is to reach a consensus on the order of transactions under a trustless environment. Permissionless blockchains, such as Bitcoin [45] and Etherum [60] employ *Proof-of-Work* (POW) to elect a single node order a block of transactions and all other nodes follow/validate this block. Permissioned blockchains, such as Hyperledger Fabric [27], leverage an ordering service to determine a global order of transactions. In this way, these blockchains can achieve sequential consistency.

Different from others, coordination-free replication, e.g., Berkeley Anna [61], utilizes mathematical properties of operations (such as commutative property) to reach consistency, since these applications are insensitive to the order of operations. Typical works include *Consistency As Logical Monotonicity (CALM)* [25, 26, 29, 30, 32] and *Conflict-free Replicated Data Type (CRDT)* [24, 48, 53].

Another consistency guarantee approach is adopted by *deterministic databases*. Multiple master nodes accept transaction requests independently and exchange them between each other with batches to achieve replication. On each replica, the same set of transactions are executed in batches in a deterministic order. Before execution, all nodes have made agreement on the ordering rule, e.g., according to the global unique ID. Essentially, it guarantees sequential consistency and supports serializable isolation due to the determinism of ordering, e.g., Calvin [57] and Aria [41].

2.1.4 Replication with Concurrent Transaction Processing. Traditional concurrency control techniques can be directly applied in single master architecture since all concurrent writes are processed locally. In sharded databases, 2PC is required for cross-shard transactions to ensure atomicity and consistency. For KV stores that base on masterless replication, they use last-write-win conflict resolution to handle multiple concurrent writes to the same key.

For multi-master replication, there exist various conflict resolution approaches for concurrent updates. The most special one is Anna [61] which is naturally conflict free since the update operations are required to be insensitive to the execution order, e.g., set union and counter. In Bitcoin [45], the single node that first solves a POW puzzle executes transactions serially without concurrency. In Fabric [27], all peer nodes execute their local transactions based on the previous block snapshot in parallel, then based on the order consensus they validate their executions and abort conflict transactions, which combinedly uses serializable snapshot isolation (SSI) [31] and OCC. MySQL Tungsten [18] can only actively prevents conflicts, and PostgreSQL BDR [10] employs last-write-win to merge conflicts. Deterministic databases execute transactions according to a

predefined serial order, which have limitation on concurrency since operating system schedules concurrently running threads in a fundamentally nondeterministic way. Existing deterministic databases rely on dependency graphs (Aria [41]) or ordered locks (Calvin [57]) to provide more parallelism while ensuring determinism.

2.2 Requirements

Despite many replicated databases have been proposed, the demand of Huawei's top customers drives us to propose a new OLTP database. Four key requirements are described as follows:

- **Multi-Master Architecture.** As motivated in Section 1, multi-master architecture is more suited for a geo-distributed scenarios.
- **Full-Featured SQL Engine.** High SQL coverage and interoperability are important demands from top customers;
- **Sequential Consistency.** Under geo-replicated setting, strong consistency of replicas is desired. However, *linearizability* is too expensive and unnecessary. A little real-time property can be compensated for performance, because *sequential consistency* is sufficient in most scenarios.
- **High Performance with Weak Isolation.** Many ICT databases used by telecom service providers are deployed under geo-replicated setting, such as Operation Support Systems (OSS), Customer Relationship Management (CRM), and Enterprise Resource Planning (ERP). They demand high performance (i.e., high throughput and low latency) but do not require strong isolation levels. It is sufficient to support weaker isolation (e.g., read committed) rather than serializability.

Existing geo-distributed transactional databases pay too much for linearizability and serializability, which cannot satisfy our requirements. Next, we will present GeoGauss.

3 SYSTEM OVERVIEW

This section provides system overview of GeoGauss. The overall architecture is shown in Figure 2.

Extension of openGauss. GeoGauss is developed based on openGauss [19], Huawei's relational database management system. We rely on its SQL engine to generate physical execution plan and design a *coupled transaction and replication layer* that unifies transaction processing with data replication. Only the write sets (a.k.a. updates) obtained after local SQL execution are exchanged between master nodes. The updates collected from multiple remote servers and the local updates are merged to resolve conflicts with a *deterministic* rule. The local transaction execution and the merge of updates are both launched with multiple threads to exploit concurrency. The merged updates are then applied on each local replica independently. The coupled transaction and replication layer are implemented by modifying openGauss's MOT engine [28], which is a high-performance row-based memory-optimized transactional storage engine (rowstore) [38, 40, 44, 46, 58, 62–65]. We add a communication module to MOT for exchanging data between nodes.

Multi-Master Architecture with Full SQL Engine. Our system contains multiple regional servers, each accepting local users' SQL requests and converting high-level SQL statements to low-level read and write requests through the parser, optimizer, and SQL execution engine. Each SQL transaction is processed by a single thread and will not release its private resources until it is committed/aborted. The send/receive buffer management is optimized to pipeline the replication and the merging of updates. Each regional server processes its local SQL requests independently based on its local replica (i.e., latest consistent snapshot).

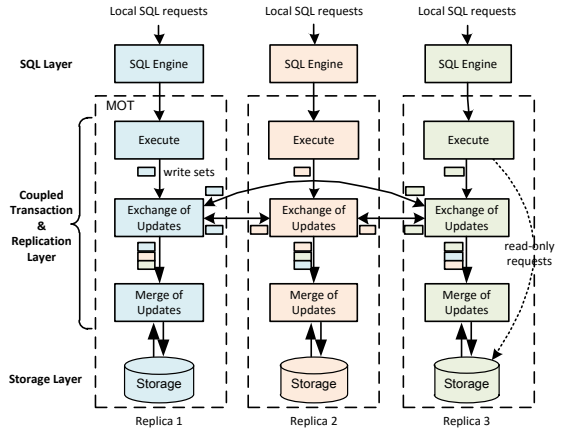


Figure 2: Overall structure.

Optimistic Read on Local Replica. Read-only transactions are directly served with a snapshot read result for low latency. For transactions that contain write operations, we perform optimistic read on the latest snapshot (i.e., the most recent consistent local replica). If the read set of a transaction becomes stale which is discovered before data replication, the transaction is aborted according to different isolation levels. The optimistic read on local replica helps improve read performance under weak isolation levels.

Epoch-based Replication of Updates. All master nodes exchange their local updates periodically (every epoch) for efficiency and consistency. Different from deterministic database that performs *logical replication* (of SQL statements), GeoGauss employs a kind of *physical replication* that exchanges the write sets between masters. Each node a collects the write sets W_a^i of all local transactions T_a^i in epoch i and sends them to ALL other nodes at the end of epoch i. As a receiver, node a has to receive the remote updates W^i from ALL other masters before generating epoch i's snapshot S^i. But it is not necessary to wait the generation of snapshot S_i before producing updates of epoch $(i+1)$. In other words, the generation of snapshots is not synchronized among replicas as will be discussed in Section 4. Note that the replicated write set contains each transaction's commit sequence number (csn) *assigned by local master*, which is globally unique and will be used in deterministic conflict merge process. We rely on a synchronized local clock to assign csn. This will neither impact correctness nor consistency but only loses a little real-time property (see Section 4.3).

Merge of Updates with Transaction Processing. After each master node collects ALL updates from all remote servers, it merges these updates to resolve the conflicts to its own local transactions (i.e., determine a local transaction to be committed or aborted). Based on a *deterministic merge rule* that considers a transaction's local timestamp (indicating execution order), the replica state after merge is guaranteed to be *consistent* among all nodes. The successfully committed inserts, deletes, and updates are then applied to the local replica and are used to update MOT index [28]. As a conventional wisdom, data replication and transaction processing

are designed separately, losing the opportunity for achieving high concurrency. We support high-throughput transaction processing by coupling the merge of updates and the optimistic concurrency control (see Section 4.1).

4 COUPLED TRANSACTION PROCESSING AND DATA REPLICATION

This section presents how the transaction processing and data replication are performed in GeoGauss with consistency guarantee.

4.1 Multi-Master OCC

By only exchanging write sets (a.k.a. updates), the merge of updates and concurrent transaction processing can be unified by our multi-master OCC algorithm, which is performed on a per-epoch basis. The *epoch* is a short period of time (10 milliseconds by default), and the epoch number is monotonically increased according to local physical time. From each replica's perspective, a globally consistent snapshot i will be achieved once it has merged the updates of epoch i and all previous epochs that are collected from local and all other remote nodes, at which time the local transactions of epoch i can be returned with committed/aborted response. The transaction processing is performed optimistically. A master node is allowed to execute transactions of epoch i even though snapshot $(i-1)$ has not been generated. In other words, the execution is not necessarily synchronized across epochs. But the validation of epoch i's transactions has to be performed based on the globally consistent snapshot $(i-1)$, where the previously executed transactions that conflict with the snapshot should be aborted.

4.1.1 Lifecyle of a Local Transaction. A transaction is submitted from client to local server, where a *thread* is assigned for each transaction. **Algorithm 1** depicts the per-thread transaction processing logic. A transaction is first assigned with a *start epoch number* (sen) and a *latest snapshot number* (lsn) (Line 1). The lsn is the latest globally consistent snapshot number maintained by the server at current time, which is used for read set validation in snapshot isolation. The transaction is then executed with generated read set RS and write set WS as output (Line 2-3). If this is a read-only transaction, the read results on the most recent snapshot are directly returned (Line 4-5), since it will not hurt sequential consistency.

For transactions that contain write operations, our multi-master OCC requires each transaction to record a few meta information for conflict merge. Specifically, a transaction is assigned with a *commit epoch number* (cen) and a *commit sequence number* (csn) (Line 6). The cen is the current physical epoch number used to determine the batch of transactions that attempt to commit together, including both the local and remote transactions with the same cen. The timestamp along with its local server id is used to generate a globally unique csn, which is used to determine the execution order within the same epoch. In addition, the transaction's write set WS along with its {sen, csn, cen} is sent to remote peers. GeoGauss supports multiple isolation levels, e.g., *Read Committed* (RC), *Repeatable Read* (RR), and *Snapshot Isolation* (SI). They are different in processing logic (Line 7-21), which will be discussed in Section 4.1.5.

For a transaction with T.cen, a synchronization point is placed to wait for snapshot (T.cen − 1) to be generated (Line 23). The snapshot (T.cen − 1) is generated by merging ALL local and remote transactions of epoch (T.cen − 1). This warrants that operations are applied on the most recent consistent snapshot, otherwise it could bring nondeterminism and lead to inconsistent replica snapshots.

The UpdateRowHeader function is then launched (Line 24), which defines the merging rule for concurrent updates by manipulating the shared table (see Section 4.1.2 and Algorithm 2). A *row header* stores each row's meta information {sen, lsn, csn, cen} indicating the row's update history by an arbitrary transaction thread, which is used for OCC's *pre-write*. For example, suppose a row header firstly updated by a transaction T, i.e., update row header by row.csn = T.csn, is then overwritten by other threads leading to row.csn ≠ T.csn, T will be aborted during the validation phase (Line 27-30). This implies that there exists a write-write conflict on the same row. According to the *deterministic* merge rule, only one write will win and will be committed, while the other are aborted.

The validation phase of transaction T cannot start until all local/remote transactions of epoch T.cen are collected and applied on the row headers (Line 25). This is essential to correctness by

Algorithm 1: Local Trans. Process (a Thread per Trans.)

```
   Input: a transaction T
   Output: return commit or abort
1  {T.sen, T.lsn} ← get current epoch no. and latest snapshot no.;
2  Execute transaction T based on latest snapshot;
3  T.RS ← T's read set; T.WS ← T's write set;
4  if T.WS == ∅ then
5      return COMMIT; //read-only transaction
6  {T.csn, T.cen} ← get current timestamp and epoch no.;
7  if Isolation == RC then
8      Add T.{sen, csn, cen, WS} to send buffer;
9  else if Isolation == RR or SI then
10     //Read set validation
11     for each record in T.RS do
12         row = FindRow(record.key);
13         if row == null then
14             Abort T; //read row is deleted
15         else if RR and record.csn ≠ row.csn then
16             Abort T; //read row is updated
17         else if SI and row.cen − 1 > T.lsn then
18             Abort T; //snapshot is updated
19     Add T.{sen, csn, cen, WS} to send buffer;
20 else
21     //Not support
22 //Operate on the most recent consistent snapshot
23 wait till snapshot T.cen − 1 is generated;
24 UpdateRowHeader(T.{sen,csn,cen,WS}); //Algorithm2
25 wait till all remote/local TXs with T.cen are applied on rowheader;
26 //Validation after all updates of an epoch having been applied
27 for each record in T.WS do
28     row = FindRow(record.key);
29     if row.csn ≠ T.csn then
30         Abort T; //abort if updated by other threads
31 //Write-back:
32 for each record in T.WS do
33     row = FindRow(record.key);
34     row.write_data(record.data);
35 return COMMIT;
```

Algorithm 2: UpdateRowHeader

Input: a transaction's $T.\{\text{sen, csn, cen}, WS\}$
Output: updated row header

```
1  for each record in T.WS do
2      row = FindRow(record.key);
3      if row == null then
4          Abort T; //row is deleted by other threads
5      if row.cen < T.cen then
6          //row is not pre-written in current epoch
             row.{sen, csn, cen} = T.{sen, csn, cen};
7      else if row.cen == T.cen then
8          if row.sen == T.sen then
9              if row.csn > T.csn then
10                 //first write wins
                     row.{sen, csn, cen} = T.{sen, csn, cen};
11             else
12                 Abort T;
13         else if row.sen < T.sen then
14             //shorter transaction wins
                 row.{sen, csn, cen} = T.{sen, csn, cen};
15         else
16             Abort T;
17     else
18         //row.cen > T.cen will never happen
```

Algorithm 3: Merge of Remote Transactions

Input: continuously received batch of remote updates $TS = \{T.\{\text{sen, csn, cen}, WS\}\}$, the sets of local transaction process threads $LTT[i]$ for epoch i, and number of replicas n
Output: continuously updated table

```
1  Receive thread:
2  while true do
3      Receive a batch of updates TS with epoch T.cen from a remote
         peer and add TS to receive buffer Buf[T.cen];
4  Merge thread:
5  sys.lsn ← get latest consistent snapshot no.;
6  while true do
7      Get a set TS from Buf[sys.lsn + 1]; //blocking get
8      foreach T.{sen, csn, cen, WS} in TS do
9          UpdateRowHeader(T.{sen, csn, cen, WS});
10         if T is not aborted then
11             Add T.{csn, WS} to commit queue Q[sys.lsn + 1];
12     N[sys.lsn + 1] ← N[sys.lsn + 1] + 1;//a counter
13     if N[sys.lsn + 1] == n − 1 then
14         //if updates from all remote peers have been processed
15         Notify all LTT[sys.lsn + 1] (wait point at Line 25 in Alg. 1);
16         foreach T.{csn, WS} in commit queue Q[sys.lsn + 1] do
17             Execute Line 27-34 of Alg. 1 for T;//validate and write
18         wait till all LTT[sys.lsn + 1] finished;
19         Notify all LTT[sys.lsn + 2] (wait point at Line 23 in Alg. 1);
20         sys.lsn ← sys.lsn + 1;//a new snapshot is generated
```

ensuring none of updates is missing. If the transaction does not meet a write-write conflict or wins during conflict merge, it is allowed to commit. This transaction's write data are used to update the involved rows (Line 32-34). After all transactions with the same commit epoch number cen are processed and applied on the table, it indicates that a new snapshot for epoch cen is generated.

4.1.2 Update Rule of Row Headers. The `UpdateRowHeader` is launched concurrently by multiple threads, where all transaction threads attempt to update the row header concurrently. In case concurrent writes occur, we follow a number of rules to determine the successful writes as illustrated in **Algorithm 2**. For ease of illustration, we focus on the update transactions. 1) If a write row is null, which means that it was deleted by other threads in past few epochs, we abort this transaction (Line 3-4). 2) If the write row is not pre-written in current epoch (T.cen) yet, i.e., row.cen $<$ T.cen, we update its row header for a candidate commit (Line 5-6). Otherwise, this row has been updated in current epoch by other threads, i.e., row.cen == T.cen. Note that, since the `UpdateRowHeader` will never be invoked to merge updates of epoch $(i+1)$ before completing merge of all updates of epoch i (the synchronous point at Line 23 in Algorithm 1), row.cen $>$ T.cen will never happen. 3) We let shorter transactions win by comparing their T.sen, i.e., row.sen $<$ T.sen. A transaction with larger T.sen means that it is closer to the current epoch T.cen (because T.cen $-$ T.sen is smaller) and is likely to commit (Line 13-16). 4) Suppose the same sen, we use csn to determine the order and follow the first-write-win rule (Line 8-12). Note that, for an insert request, the `FindRow` function cannot locate the row. In addition, multiple concurrent insert transactions may insert into the same row. We use a *temporary table* to store the inserted rows to deal with the insert conflicts within the same epoch. Instead of invoking `FindRow` that relies on the table index, we use the temporary table for newly inserted data.

4.1.3 Merge of Remote Updates. The merge of remote updates and the local transaction processing are running concurrently and interact with each other. On each master node, multiple receive threads and merge threads are continuously running as shown in **Algorithm 3**. The receive thread keeps receiving updates TS from remote peers and temporarily stores them in the receive buffer with their commit epoch number T.cen information (Line 2-3). Suppose sys.lsn is the number of the most recent globally consistent snapshot (Line 5). The merge thread merges updates of epoch (sys.lsn + 1) based on the most up-to-date table state (i.e., snapshot sys.lsn) and will produce consistent snapshot *one by one*. Specifically, if there exist remote updates of epoch (sys.lsn + 1) in the buffer, it processes them by invoking the `UpdateRowHeader` function (Line 7-9). The transactions that are not aborted are pushed into a commit queue $Q[sys.\text{lsn}+1]$ (Line 10-11). If the updates of epoch (sys.lsn + 1) from all remote peers have been applied on row headers, it notifies all the local transaction threads for epoch (sys.lsn + 1) that are waiting for completing merge and triggers the validation and write-back for each update in the commit queue (Line 15-17). Once all the local threads of epoch (sys.lsn + 1) are finished, it means that a new consistent snapshot is achieved. The merge thread immediately notifies all the local transaction threads of next epoch that are waiting for this up-to-date snapshot (Line 18-20).

4.1.4 Optimistic Concurrency Control. The merge of updates is launched by allowing all local transaction threads and merge threads pre-write row headers concurrently. Then during the validation

phase, only the transactions whose pre-writes on row header are not overwritten by other threads are allowed to commit for ensuring *atomicity*. This is quite similar to OCC. The distinctions from traditional OCC lie on two facts. 1) The pre-writes can be originated from remote workers with their {sen, csn, cen} being assigned by remote source workers. The *uniqueness* of sequence number is the key to achieve consistency as will be illustrated in Section 4.3. 2) The pre-write and validation are performed on a *per-epoch* basis. The pre-write of transactions in epoch $(i+1)$ cannot start until snapshot i is generated. The validation of transactions in epoch $(i+1)$ cannot start until all updates of epoch $(i+1)$ have been completely applied on row headers.

4.1.5 Isolation. Weak isolation models cannot guarantee serializability, but their benefits to concurrency are frequently considered by application developers to outweigh costs of possible consistency anomalies that might arise from their use. GeoGauss supports multiple weak isolation levels as shown in **Algorithm 1**, allowing users to choose the optimal one for their specific application.

Read Committed (RC). For a transaction T of epoch T.cen, only the committed data as snapshot can be accessed in our system, so the support of RC isolation is straightforward. None of the local transactions is aborted and all of them are used to generate write sets. Its write set $T.WS$ along with its metadata are immediately added to the send buffer (Line 8). Note that it cannot guarantee to read the most up-to-date committed data since the most recent snapshot T.cen -1 might not be generated yet. But it can guarantee that the latter read item is a more up-to-date one.

Repeatable Read (RR) and Snapshot Isolation (SI). As a single-version in-memory database, we realize the RR and SI mainly through *optimistic* approach, i.e., read set validation. 1) If the previously read record is deleted, the transaction is aborted under these two isolation levels (Line 13-14). 2) During a transaction's execution, each time it reads a record, the row's csn is updated as the record's csn. If a row is updated by other threads after its first read (i.e., row.csn $\neq$ $record$.csn), the transaction is aborted under RR isolation (Line 15-16). 3) If a transaction's read row is updated in a new snapshot (snapshot row.cen -1 due to the synchronous point at Line 23) since it starts execution (snapshot T.lsn), the initial read snapshot is updated (i.e., row.cen$-1 > T$.lsn) which violates SI. The transaction is aborted under SI (Line 17-18). Only the transactions that pass read set validation will be sent out (Line 19).

4.2 An Illustrative Example

We provide an illustrative example in Figure 3 showing how our algorithm works with two replicas. Each replica runs multiple local transaction processing threads and merging threads concurrently. Suppose a consistent snapshot S_0 among two replicas in the beginning, replica a and replica b independently execute their local transactions and independently generate their local write sets. For the transactions that finish execution, their write sets (updates) are exchanged between replicas at the end of every epoch. After a node receives all remote updates for the first epoch, these updates are merged with local write sets to resolve write-write conflicts based on a deterministic merge rule (see Algorithm 2), and are applied to the original snapshot S_0 to generate a new snapshot, i.e., $S_1 = S_0 \oplus \{W_{a1}(x), W_{b1}(x)\}$, where $\oplus$ represents the deterministic

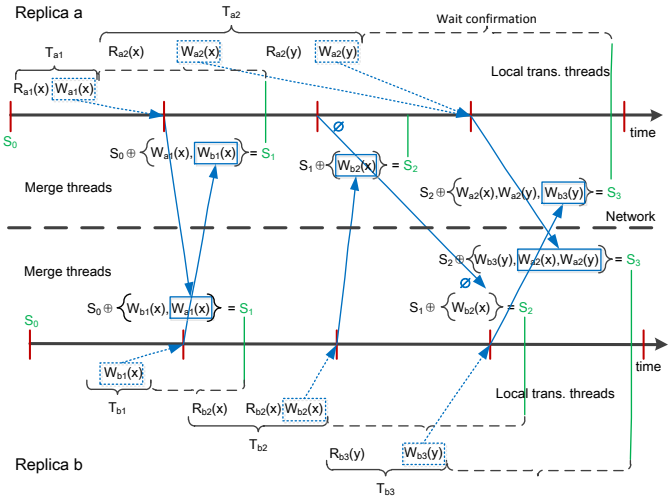


Figure 3: An illustrative example of multi-master OCC with two replicas. Each replica a independently accepts local transactions, e.g., T_{a1} and T_{a2}. $R_{a1}(x)$ and $W_{a1}(x)$ represent the read and write operations on data item x of replica a, respectively. A local transaction with commit epoch number cen **cannot be confirmed until snapshot** cen **is generated, so there exists a wait confirmation period. Red lines are epoch boundaries. The local clock time of two replicas is not necessarily synchronized. Green S_i is the globally consistent snapshot. Blue box is the sent or received write set.**

merge operation. Only when the local snapshot for epoch *cen* is generated, can the local transactions with commit epoch number *cen* be returned by host master with committed/aborted notification, e.g., T_{a2} will not return until S_3 is generated on replica a.

Long Running Transactions. Only when a transaction finishes execution at epoch i can we send its write sets at the end of epoch i. For example, T_{a2} is a long running transaction that crosses three epochs and ends at epoch 3, its write sets $\{W_{a2}(x), W_{a2}(y)\}$ are sent out together at the end of epoch 3. The transaction could be aborted due to violation of isolation requirements, e.g., the read set is updated under RR isolation. Nevertheless, if there is no updates for an epoch, e.g., epoch 2 on replica a, an empty message is sent out to prevent endless wait at remote peers.

Network Delay. Due to network latency, the two replicas probably may not reach a consistent snapshot exactly at the same time point, but they will at certain time. This will NOT block system running and will NOT impact sequential consistency (see Section 4.3). For example, snapshot S_2 is generated at epoch 3 on replica a and at epoch 4 on replica b. The write set $W_{b3}(y)$ of T_{b3} is routinely sent out to replica a at the end of epoch 3. However, the merge of updates of epoch i cannot start until snapshot $(i-1)$ is generated (Line 23 in Algorithm 1). For example, local update $W_{b3}(y)$ of epoch 3 cannot be merged with snapshot S_1, but can only be merged with snapshot S_2 and the updates of epoch 3 ($W_{a2}(x)$ and $W_{a2}(y)$) to generate S_3.

Isolation. The transactions that violate isolation are aborted during read set validation before their write sets being sent out. With RC, since a transaction can only read committed data from snapshot, it will never be aborted, even though it may read a very old snapshot. With RR, a transaction is aborted if its read data is stale.

For example, T_{b2} is aborted under RR because its first read $R_{b2}(x)$ is the x in S_0 while its second read $R_{b2}(x)$ is the updated x in S_1. With SI, a transaction is aborted if its read snapshot is stale. For example, T_{a2} will be aborted under SI because its first read $R_{a2}(x)$ is based on S_0 while its second read $R_{a2}(y)$ is based on S_1.

4.3 Consistency of Replicas

We show how GeoGauss guarantees consistency of replicas.

Lemma 1. *Following multi-master OCC, the read operations will not affect consistency of replicas.*

Proof. In our system, each master node performs local read on snapshots and then generates the write set. That is, the write set is generated by a single source worker and then sent out for write-write conflict merge. No matter which snapshot (old or new) a transaction reads, the write data by the transaction are *deterministic* before they are sent out. The consistency of read operations will only affect isolation property. The transactions that read different versions of data violate isolation constraints (e.g., RR and SI) and will be aborted without producing updates. Only the updates can change the state of replicas. Therefore, the read operations will not affect consistency of replicas. □

Lemma 2. *Suppose an initial database state S_i and a set of updates $TS = \{\mathcal{U}(T_1), \mathcal{U}(T_2), \ldots, \mathcal{U}(T_n)\}$ where $\mathcal{U}(T_i) = T_i.\{\text{sen, csn, cen}, WS\}$. No matter what order these updates input in, Algorithm 2 will produce the deterministic state S_{i+1}.*

Proof. Algorithm 2 is used to merge write-write conflicts. Basically, the merge rule is defined by comparing tuples {sen, csn, cen} (Line 5-16). We define a strict total order '$\preceq$' on the set of updates TS as follows. For any two updates $\mathcal{U}(T_i)$ and $\mathcal{U}(T_j)$ of TS, $\mathcal{U}(T_i) \preceq \mathcal{U}(T_j)$ if one of the following conditions is met.

- T_i.cen > T_j.cen;
- T_i.cen = T_j.cen and T_i.sen > T_j.sen;
- T_i.cen = T_j.cen and T_i.sen = T_j.sen and T_i.csn < T_j.csn.

Because T.csn is generated based on T's source worker id and its commit timestamp assigned by the source worker, which is *globally unique*, there must be a strict order between any two updates T_i and T_j. For conflict updates on a row x, Algorithm 2 allows for the updates of transaction T with the "smallest" (according to '$\preceq$') to commit. That is, the collection of successful updates always equals to $\{\{x \in \mathcal{WS} : T_i \in C(x)\} \mid \mathcal{U}(T_i) \preceq \mathcal{U}(T_j), \forall T_j \in C(x)\}$, where x is a table row, $\mathcal{WS} = \{T_1.WS \cup T_2.WS \cup \ldots \cup T_n.WS\}$ is the union of the write sets of all transactions in an epoch, and $C(x)$ indicates a set of transactions that write row x. Therefore, with an initial state S_i, no matter what order these updates input in, the *deterministic* order implicitly defined on the set of updates with unique tuples {sen, csn, cen} will guarantee the deterministic output S_{i+1}. □

Theorem 3. GeoGauss *following multi-master OCC (Algorithm 1,2,3) enforces consistency of replicas at the granularity of epochs.*

Proof. Lemma 1 states that read operations will not affect consistency of replicas, so we only focus on studying the effects of write operations. Lemma 2 states that given an batch of updates and an initial state, the output state is *deterministic* regardless of nondeterministic input order of these updates. To achieve consistency of replicas, it is crucial to guarantee 1) the identical initial state and 2) the identical batch of updates on all replicas. First, the synchronization points at Line 25 in Algorithm 1 and at Line 18 in Algorithm 3 ensure that each replica runs an *identical* batch of updates, i.e., having merged all local/remote updates of an epoch. Second, the synchronization point at Line 23 in Algorithm 1 and the latest snapshot number verification at Line 5 in Algorithm 3 ensure that the updates of epoch $(i+1)$ are applied on snapshot i, which guarantees the *identical* initial snapshot. Therefore, the consistency of replicas are guaranteed at the granularity of epochs. □

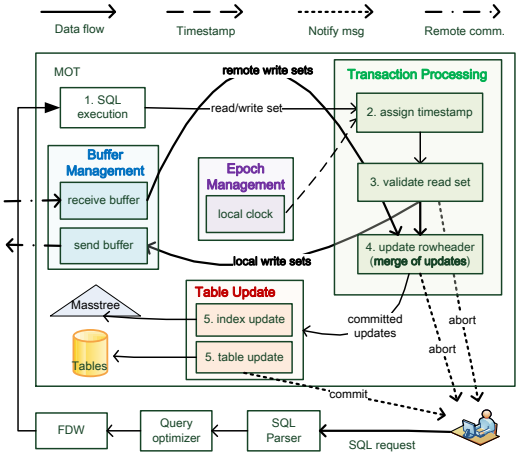


Figure 4: Data flow of GeoGauss.

The read operations on local replica are not exchanged among replicas. The order of write operations in an epoch is deterministic and consistent on each individual replica. The epoch snapshots on all replicas are generated one by one sequentially. Thus, the consistency enforced by GeoGauss is a kind of *sequential consistency*.

5 IMPLEMENTATION

We implement GeoGauss based on openGauss 2.0.0 MOT storage engine [19]. The data flow of GeoGauss is depicted in Figure 4.

5.1 Parallelism and Communication

Thread Blocking and Notification. A large number of transaction threads might be blocked waiting for a notification signal (Line 23 in Algorithm 1). It is possible that a thread entering into the blocking phase and a signal notifying the thread concurrently occur. If the thread misses the notification signal, it will be suspended permanently. To avoid the anomalies, we let each thread sleep and actively check the condition periodically, which works well with a small number of threads. On the other hand, we introduce a mutex lock to prevent concurrency anomalies. But these threads that share the mutex lock have to be invoked one by one serially. To maximize parallelism, we create multiple mutex locks and make each one only shared by a disjoint subset of threads. This can effectively alleviate the burst of thread notifications.

Message Queue and Pipelining. We rely on Protocol Buffers [11] to compress the buffered write sets transmitted between nodes. Instead of the heavyweight gRPC [8] used by openGauss, we prefer the lightweight ZeroMQ [14] (with publish-subscribe mode) to

realize efficient data transmission between nodes. Furthermore, as depicted in Figure 3, the write sets in a send buffer are packaged and sent out together at the end of each epoch. This could seize computation resources and incur network bursts. To overlap the communication and computation, we introduce pipelining technique that sends write sets in mini-batches in a streamlined manner. Note that, an EOF message is sent indicating the end of epoch (according to physical time) in streamlined communication, so the receiver can ensure the completeness of an epoch of transactions.

Zero-Copy Send and Receive. We further improve the pipelining approach by zero-copy. Usually, when a user space process has to execute system operations like reading or writing data from/to a network device, it has to perform one or more system calls that are then executed in kernel space by operating system. Instead of packaging the transmitted data with Protocol Buffers, we copy the write sets data through zero-copy and directly send them out. This can eliminate the serialization and deserialization costs.

5.2 Fault Tolerance

It seems intuitive to support fault tolerance under a multi-master architecture with full replicas. Once a regional node fails to provide service, its local transaction requests can be served by another master node. However, this scenario is more complicated under a coordinator-free architecture. After collecting and merging the write sets with commit epoch number cen from all remote peers, a server can respond to local users with committed or aborted notifications for the transactions with cen. This approach avoids expensive coordination but suffers from two risks. 1) Permanent Blocking: The whole system may be blocked waiting for the write sets from a failed server. 2) Loss of Updates: A node has collected and merged all remote write sets with local ones and returns to local users, but its local updates are not received by other peers before server breakdown, which leads to an inconsistent state. Even though a timeout mechanism can be used by the receiver to detect node failure, it is possible that the source node was already failed, leading to the loss of local updates.

Avoiding Permanent Blocking. We employ Raft protocol (using braft implementation [17]) to make consensus on the status of live nodes. Raft elects a leader to propose commands. The Raft leader heartbeats followers periodically. If a node failure is detected through heartbeat, the leader proposes to remove this failed node. All the alive nodes make consensus and proceed without considering this failed node. Similarly, in case the failed node recovers or a new node joins, a Raft process is launched to make consensus on the new cluster status. Different from traditional coordination-based data replication systems where consensus is invoked for each data update, the consensus in our system is invoked only when the number of alive nodes is changed, which will not affect the coordination-free data flow.

Avoiding Loss of Updates. We place one (or more) *writeset cache server* associated with each replica node in each region, which caches the generated local write sets for backup. Each time a server node sends local write sets to remote peers, it also simultaneously sends a copy to the local writeset cache server, which will send back an ack message after receiving. Only when all remote write sets with commit epoch number cen have been collected and merged and the local write sets with cen have been backup, can we return to users the committed/aborted states of the transactions with cen. Once a node failure is detected and removed, the other remote peers will ask the corresponding writeset cache server to check whether there exists loss of updates by comparing the monotonic cen. If so, the remote peers need to pull the lost write sets and merge them to advance to the consistent snapshot cen before proceeding. Since the local backup occurs simultaneously with the sending of local updates and the local network is much faster than the cross-region network, the cost of local backup is hidden which will not impact the overall performance.

6 DISCUSSION

Transaction-based vs. Batch-based vs. Epoch-based. Traditional wisdom prefers to handle transactions on a per-transaction basis [34, 55]. This is suboptimal under high-contention workload and geo-distributed environment due to the expensive coordination cost for each individual transaction. Our multi-master OCC algorithm can also be slightly changed to support transaction-based conflict merge. However, this brings more complexity for ensuring atomicity and consistency of replicas. An alternative is batch-based processing (e.g., deterministic databases [41, 57]) that limits the number of updates for each batch instead of fixed time period. The batch-based approach could have undesirable performance due to imbalanced workloads among transactions, especially for long running transactions, due to the barriers across batches. Our epoch-based multi-master OCC divides transactions into multiple individual subsets according to the local time and their commit time (i.e., cen). The updates of long running transactions will be merged in later epochs without blocking the merging process. The dependencies between transactions in two subsets are ensured by the monotonically increased snapshot number, and the dependencies between transactions within the same subsets are ensured by the deterministic order defined by Algorithm 2. In addition, the epoch-based and batch-based approach is less expensive for fault tolerance since the consensus and durability are established on a set of transactions instead of individual transaction.

Epoch Length. Intuitively, the epoch length should be set longer than the round trip time (RTT). But it is not necessary to do this due to the deterministic execution with coordination-free property. Unlike coordination-based approach (e.g., 2PC) that relies on remote server's confirmation on the validity of remote data (e.g., through locking), our multi-master OCC only verifies the completeness of an epoch of updates (i.e., have collected updates from all peers from the receiver's point of view) and does not need to coordinate with remote peers (with one or more RTTs) before proceeding. In our cross-region experiments, it is common that the latest snapshot that is used for generating read/write sets lags behind the current physical time by **3-5 epochs** (corresponding to single-trip delay 30-50ms). Despite the epoch length can be set regardless of network RTT, it should be limited by our serving model. Our system reuses the serving model of openGauss, in which a thread is allocated to serve a submitted transaction and will not release its resources until the transaction is committed or aborted. Before that, these serving threads might be blocked and stop serving other requests. Setting a short epoch length can increase the frequency of update

exchanges and as a result shortens the confirmation time, which not only reduces the blocking time to improve throughput but also decreases the latency. However, too frequent communications and update merges will increase the scheduling cost that outweighs the benefits (see Section 7.4.1).

Distinctions from Deterministic Databases. Existing *deterministic databases* [5, 23, 41, 42, 50, 51, 56, 57] replicate batches of SQL transaction requests to multiple master nodes. After collecting all transactions of the same batch, each replica is required to execute these transactions according to a predefined serial order. This requirement is stricter than that required for an execution to be serializable (which only requires that transactions execute according to some serial order) since the operating system schedules threads in a fundamentally nondeterministic way [23]. Existing deterministic databases rely on single-thread ordered locks (e.g., Calvin [57]) or dependency graph resolution (e.g., Aria [41]) to support concurrency while ensuring deterministic serial order. Executing a transaction's logic as a single unit fundamentally limits the performance of serializability as compared to weak isolation levels (e.g., read committed) which allow applications more interleavings between conflicting transactions. The higher the overlap of the read and write sets across transactions, the higher the improvement in concurrency. According to Huawei's production usages (see Section 2.2),*read committed* isolation is sufficient in most scenarios.

Compared with deterministic databases that exchange and merge SQL statements, our multi-master OCC that exchanges and merges replica-generated write sets has distinct advantages. 1) Our multi-master OCC supports weaker isolation levels with higher degree of concurrency. 2) To support concurrent execution on each node, we do not need additional scheduling overhead for ensuring the determinism (e.g., single thread lock manager in Calvin [57] or dependency graph resolution in Aria [41]). 3) In deterministic databases, a previous batch of transactions must finish executing before a new batch can begin, which could lead to undesirable performance due to imbalanced workloads among transactions, especially for long running transactions. While in GeoGauss, transactions are arranged into epochs *according to their commit time* rather than being arranged into batches according to their submission time. It is allowed to process new transactions during the execution of long transactions, so the impact of long transactions is greatly alleviated.

7 EVALUATION

This section conducts cross-region experiments to evaluate GeoGauss.

Cluster Setpup. Our cross-region cluster contains three geo-distributed nodes, including a node in Zhangjiakou city (China north), a node in Chengdu city (China southwest), and a node in Shenzhen city (China south). Each node (Aliyun ecs.r6e.8xlarge instance) is equipped with 32 vCPUs, 256GB DRAM, running Centos 7.6 OS. The network bandwidth between cross-region nodes is about 100Mbps/s. We also set a separate node (ecs.c6.8xlarge instance) in each region simulating local client to send SQL requests.

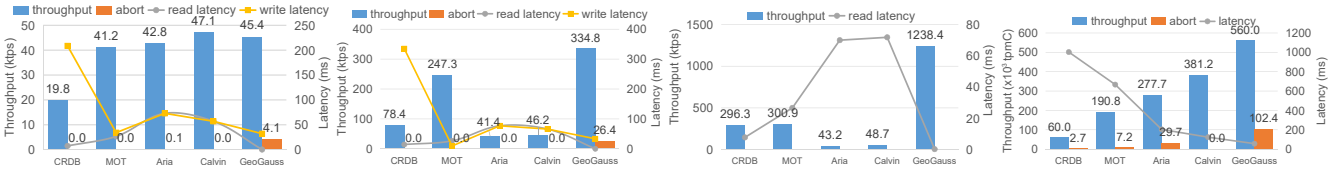
Competitors and Configurations. We choose CockroachDB [55] (CRDB), our base system openGauss MOT [19] (MOT), and two deterministic databases Calvin [57] and Aria [41] for comparison. The configurations of these competitors and our system are described as follows. 1) CRDB: For fairness, CRDB is configured with *in-memory store* and configured with *stale reads* from outside the read row's home region. In addition, as suggested by CRDB documentation, we use more nodes running CRDB for maximizing its performance. We place 2 additional nodes in each region and totally have 9 nodes and 3 replicas running CRDB. 2) MOT: OpenGauss MOT follows a primary-backup architecture where the backup replicas cannot serve read/write requests. We place the primary node and the other two backup nodes in the same region to construct a 3-node local cluster, and let the one local client and the two remote clients submit requests to the single primary node. Thus, MOT works as a *centralized* database with backup nodes. 3) Calvin and Aria: For these two deterministic databases, we use the implementation from [16] and set with their default configurations. Their worker threads are set as 28 leaving 4 remaining cores for other works such as communication, dependency analysis, lock ordering, etc. 4) GeoGauss: Our system is configured with 10ms epoch length by default. Note that, CRDB, Calvin, and Aria only supports serializable isolation, while MOT and GeoGauss support multiple weak isolation levels (RC is used by default). For CRDB, MOT, and GeoGauss, they use standard benchmark interface, in which each connection only sends a single transaction once a time. In these systems, each node is configured with 256 connections. Additionally, all these systems are configured with 3 replicas.

Workloads. We use two popular benchmarks, YCSB [33] and TPC-C [21]. For YCSB workload, we use one table with 10 columns and 1,000,000 rows. We choose YCSB-A (50% read and 50% write), YCSB-B (95% read and 5% write), YCSB-C (100% read). For TPC-C benchmark, we configure it with 800 warehouses and 200 client connections for sending query results. Since Calvin and Aria do not provide full SQL engine and they do not support complex queries (e.g., join and range scan), they can only support New-Order transactions and Payment transactions. We use mixed TPC-C workloads with 50% New-Order transactions and 50% Payment transactions.

7.1 Geo-Distributed Cross-Region Results

We run geo-distributed experiments on our cross-region cluster. The throughput for successfully committed and aborted transactions and the average read/write latency are reported in Figure 5.

Regarding the write-intensive YCSB-A workload, Calvin shows the highest throughput while CRDB shows the lowest throughput. CRDB, MOT, and GeoGauss are all limited by the maximum number of connections (each connection only serves one transaction once a time). A write operation takes up more time for its connection since write-write conflicts should be resolved, while a read-only operation is fast since stale-read is configured. Thus, we can observe that CRDB, MOT, and GeoGauss all show lower throughput for write-intensive YCSB workload. Calvin and Aria implement their own integrated test tools without connections from outside, so they do not have this limitation. But we can see that they show stable throughput and latency results for different YCSB workloads. This is because that they employ deterministic execution of transactions, which has limited improvement opportunity for concurrency when processing more read operations. CRDB shows much higher write latency than the other systems due to its expensive coordination cost, but shows low read latency since we configured it with stale read. Calvin and Aria process transactions in batches and need more time for scheduling these batched transactions to achieve



(a) YCSB-A (b) YCSB-B (c) YCSB-C (d) TPC-C

Figure 5: Comparison results on throughput and latency.

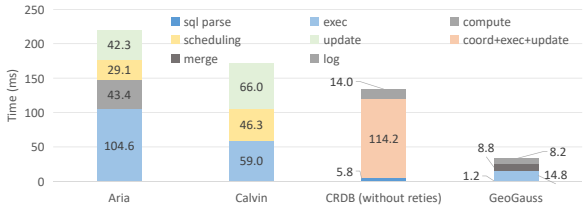
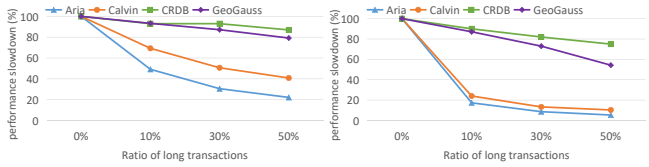


Figure 6: Runtime breakdown (TPC-C).



(a) Delay = 20ms (b) Delay = 100ms

Figure 7: Effect of long transactions.

deterministic execution. GeoGauss exhibits lower write latency and super low read latency because it directly reads local snapshot for read-only transactions. The read latency of MOT is higher than GeoGauss because MOT is configured as a centralized database and it may serve remote read requests. In addition, the abort rate in GeoGauss is the highest since we use weak isolation and have explored more concurrency at the expense of more conflicts.

Regarding the TPC-C workload with more complicated transactions, GeoGauss exhibits superiority over the other systems, say 56 ktps and 56.5 ms average latency. Comparing with CRDB, it achieves 9.33X higher throughput and 17.74X lower latency. This improvement can be mainly attributed to the more concurrency resulted from weaker isolation (for high throughput) and the coordinator-free multi-master architecture (for low latency). Though the centralized MOT shows better performance than CRDB, it is expected to get lower throughput when scaling to more nodes. In addition, GeoGauss achieves 1.47X-2.02X higher throughput and 2.18X-3.61X lower latency over deterministic databases, considering that GeoGauss provides full SQL engine and additional logging mechanism. Aria shows higher abort rate than Calvin since that Aria relies on dependency graph for aborting conflict transactions while Calvin relies on ordered lock to achieve deterministic execution. Notice that, as shown in [41], Aria should perform better than Calvin in a *local* cluster environment, where the higher scheduling cost of Calvin will dominate the runtime. The abort rate of GeoGauss is the highest among all systems due to high probability of conflicts. But we found that most of conflict transactions are aborted during the local execution phase and the local read set validation phase before they are sent out, so the aborted transactions will not waste too much computation/network resources.

7.2 Performance Breakdown

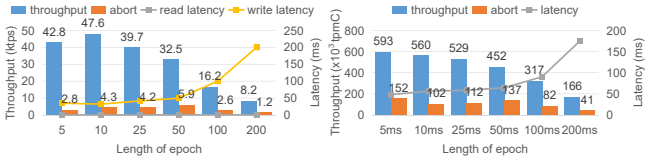
We study the cost of different phases during transaction processing. Figure 6 shows a runtime breakdown of a successfully committed transaction of these systems with TPC-C workload. All systems need to execute transaction and update database. The two deterministic databases Aria and Calvin process transactions in batches and require expensive global synchronization barriers across batches, which prolongs the execute phase. Moreover, they have to spend additional scheduling cost for ensuring the deterministic execution order, i.e., read/write reservation in Aria and ordered locking in Calvin. Aria further needs a compute step to analyze transaction logic. CRDB and GeoGauss both provide full SQL engine and logging service, so they spend some time on SQL parsing and logging. CRDB relies on coordination to achieve atomicity. The coordination cost, execution cost, and update cost are mixed together which cannot be decoupled, so we show the total time for the mixed cost. The coordination that requires several RTTs is expensive in cross-region environment. It is noticeable that CRDB will retry a conflict transaction and will be aborted only when the retry times exceed a threshold. The official TPC-C benchmark provided by CRDB counts the retried transactions for average latency calculation (but we found many retries). In this runtime breakdown experiment, we only count the successfully committed transactions without retries for average cost measurement. GeoGauss shows much faster execution and update due to the coordinator-free execution and more concurrency resulted from weak isolation.

7.3 Long Running Transactions

As discussed in the distinctions from deterministic databases (see Section 6), GeoGauss should exhibit superiority over deterministic databases when processing long running transactions, since a barrier exists across batches in deterministic databases. In this experiment, we manually add a fixed 20 ms/100 ms delay to a randomly selected YCSB-A transaction to simulate long running transactions. Figure 7 plots the system's performance slowdown (with respect to throughput) when varying the portion of long transactions. The throughput of Aria and Calvin are both decreased a lot when processing more long transactions. The performance slowdown of deterministic databases is even more significant when processing longer transactions as shown in Figure 7b, say 80% slowdown for 10% long transactions. While GeoGauss is more robust to long transactions, because GeoGauss allows to process new transactions during the execution of long transactions of previous epochs. It is worth mentioning that the synchronization point in our algorithm is for ensuring the completeness of remote updates but not a global barrier across batches of transactions. CRDB is robust to long transactions even though its latency is much longer than ours, because the (manually set) delay is hidden in CRDB's *parallel commit* phase.

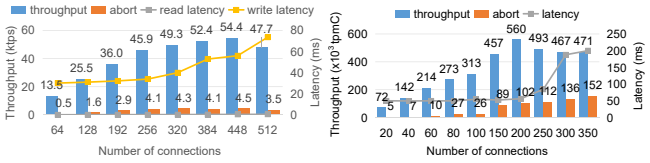
7.4 Varying Key Parameters of GeoGauss

7.4.1 Epoch Length. As discussed in Section 5.1, GeoGauss uses one thread per transaction (i.e., per connection) and will not release



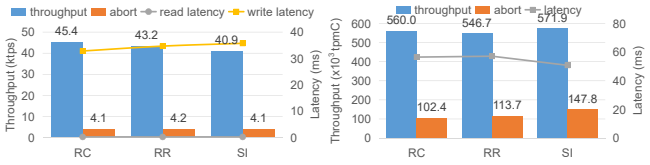
(a) YCSB-A (b) TPC-C

Figure 8: Effect of epoch length.



(a) YCSB-A (b) TPC-C

Figure 9: Varying number of serving threads.



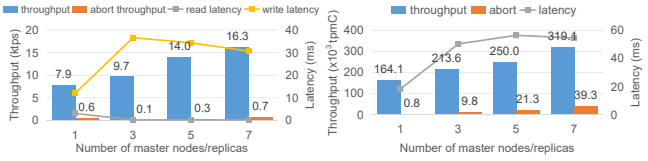
(a) YCSB-A (b) TPC-C

Figure 10: Performance of different isolation levels.

the connection until the transaction is committed or aborted. Thus, a longer epoch could lead to a phenomenon that all the connections are occupied by transactions waiting for confirmation and the system may stop serving for a period. On the other hand, it also affects the write latency since a submitted write request must wait for the epoch snapshot to be generated before responding to users a committed/aborted notification. Long epoch will result in long write latency, while a short epoch may result in too frequent conflict merging and also degrades performance. Figure 8 shows the throughput and write/read latency results of YCSB-A and TPC-C workloads with different epoch lengths ranging from 5 ms to 200 ms. The throughput results are under expectation as analyzed above. The write latency is on one hand determined by the single-trip delay when the epoch length is short; on the other hand determined by the epoch length setting when the epoch length is long. The read operations are all served by local region server, so the read latency can be ignored.

7.4.2 Number of Connections. As presented in Section 5.1, GeoGauss allocates a thread for each transaction, corresponding to a benchmark connection. The connections may be blocked waiting for the committed/aborted confirmation of this transaction. Using a small number of connections has the risk of all connections being blocked which will degrade the throughput performance, while using too many connections may result in too much thread scheduling cost and hurt performance. Figure 9 shows the throughput and latency results under YCSB-A and TPC-C workloads when varying the number of connections. As expected, the throughput is higher with the increased number of connections but decreases when using too many connections. The write latency is stable with a small number of connections but going higher when the number of connections exceeds a certain number owing to the limited network bandwidth.

7.4.3 Isolation. We study the performance when adopting different isolation levels in this experiment. Figure 10 shows the results under



(a) YCSB-A (b) TPC-C

Figure 11: Scalability.

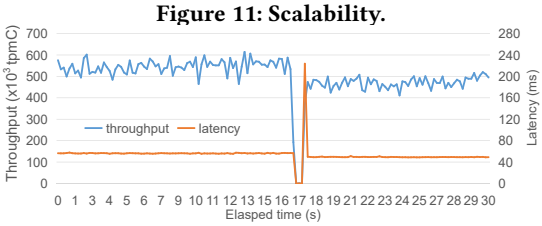


Figure 12: Fault tolerance of GeoGauss.

YCSB-A and TPC-C workloads. There is not too much difference on the throughput and latency results under different isolation levels, except that the abort rate is higher with higher isolation levels. This is under expectation since *RR* and *SI* require a read set validation step that may increase the abort rate.

7.5 Scalability

In this experiment, we study the scalability of GeoGauss. We scale the number of replica nodes from 1 to 7 and run YCSB-A and TPC-C. As shown in Figure 11, the throughput is steadily increased with the increasing number of replica nodes. It achieves 1.68X and 1.49X higher throughput with 7 nodes compared to that with 3 nodes for YCSB-A and TPC-C, respectively. The latency is relatively stable when scaling from 3 nodes to 7 nodes. The write latency under the multi-master setting is higher than that under the centralized single-node setting, since it involves cross-region merge of updates.

7.6 Fault Tolerance

The fault tolerance of GeoGauss relies on Raft-based membership management and writeset cache servers (see Section 5.2). We manually shutdown a node and see how GeoGauss acts and recovers after failure. Figure 12 shows the changes of throughput and latency as we include a node failure. The temporary performance degradation during an node failure is due to the blocking of service, i.e., waiting for the updates from the failed node. GeoGauss can quickly respond to this failure owing to our Raft-based membership management (with 500ms timeout setup). Our system recovers to serve users' request in a short time (around 1s). After removing the failed node, our system contains only two nodes, so the overall throughput decreases accordingly.

8 CONCLUSION

This paper presents GeoGauss, a coordinator-free geo-distributed NewSQL database. We have shown that the performance of geo-distributed transaction processing can be greatly improved by GeoGauss with weak isolation, which allows for overlapping the read and write sets across transactions and as a result brings more concurrency. By coupling coordinator-free data replication and optimistic concurrency control, our multi-master OCC algorithm can efficiently merge the conflicts and at the same time enforces strong consistency of replicas at the granularity of epochs. The key features of GeoGauss, including full SQL engine, multi-master replication, strong consistency, and high performance, greatly meet the requirements of telecom service providers.

REFERENCES

[1] 2021. Apache CouchDB. http://couchdb.apache.org/
[2] 2021. ArangoDB. https://www.arangodb.com/
[3] 2021. Cloudant. https://www.ibm.com/hk-en/cloud/cloudant
[4] 2021. ExtremeDB: Cluster Distributed Database System. https://www.mcobject.com/cluster/
[5] 2021. FaunaDB. https://fauna.com/
[6] 2021. FoundationDB. https://www.foundationdb.org/
[7] 2021. Galera Cluster for MySQL. https://galeracluster.com/
[8] 2021. gRPC: A high performance, open source universal RPC framework. https://grpc.io/
[9] 2021. MySQL's primary-secondary replication. https://dev.mysql.com/
[10] 2021. PostgreSQL BDR. https://wiki.postgresql.org/wiki/BDR_Project
[11] 2021. Protocol Buffers. https://developers.google.com/protocol-buffers
[12] 2021. Riak: Enterprise NoSQL Database. https://riak.com/
[13] 2021. Semi-synchronous replication at facebook. http://yoshinorimatsunobu.blogspot.com/
[14] 2021. ZeroMQ: An open-source universal messaging library. https://zeromq.org/
[15] 2022. Apache HBase. https://hbase.apache.org/
[16] 2022. Aria: A Fast and Practical Deterministic OLTP Database. https://github.com/luyi0619/aria
[17] 2022. Baidu braft. https://github.com/baidu/braft
[18] 2022. MySQL Tungsten. https://www.continuent.com/products/tungsten-replicator
[19] 2022. openGauss. https://opengauss.org/
[20] 2022. PostgreSQL. https://www.postgresql.org/
[21] 2022. TPC-C Homepage. https://www.tpc.org/tpcc/
[22] 2022. YugabyteDB: Distributed SQL Database. https://www.yugabyte.com/
[23] Daniel J Abadi and Jose M Faleiro. 2018. An overview of deterministic database systems. *Commun. ACM* 61, 9 (2018), 78–88.
[24] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *J. Parallel and Distrib. Comput.* 111 (2018), 162–173.
[25] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. 2017. Blazes: Coordination Analysis and Placement for Distributed Programs. *ACM Trans. Database Syst.* 42, 4, Article 23 (oct 2017), 31 pages.
[26] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach.. In *CIDR*. 249–260.
[27] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.
[28] H. Avni, A. Aliev, O. Amor, A. Avitzur, I. Bronshtein, E. Ginot, S. Goikhman, E. Levy, Lu Levy, I., F., and L. Mishali. 2020. Industrial-Strength OLTP Using Main Memory and Many Cores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3099–3111.
[29] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (2014), 185–196.
[30] Peter David Bailis. 2015. *Coordination avoidance in distributed databases.* University of California, Berkeley.
[31] Michael J Cahill, Uwe Röhm, and Alan D Fekete. 2009. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–42.
[32] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and Lattices for Distributed Programming. In *Proceedings of the Symposium on Cloud Computing (SoCC '12)*. 1:1–1:14.
[33] 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 (SoCC '10)*. 143–154.
[34] 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.
[35] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.
[36] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. 2006. Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06) (EuroSys '06)*. 117–130.
[37] Ant group. 2022. OceanBase. https://open.oceanbase.com/
[38] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 1675–1687.
[39] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
[40] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 21–35.
[41] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2047–2060.
[42] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-Based Commit and Replication in Distributed OLTP Databases. *Proc. VLDB Endow.* 14, 5 (2021), 743–756.
[43] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *Proc. VLDB Endow.* 12, 11 (2019), 1316–1329.
[44] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2014. MaaT: Effective and Scalable Coordination of Distributed Transactions in the Cloud. *Proc. VLDB Endow.* 7, 5 (jan 2014), 329–340.
[45] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.
[46] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 677–689.
[47] Pincap. 2022. TiDB. https://pingcap.com/products/tidb
[48] Nuno Preguiça. 2018. Conflict-free replicated data types: An overview. *arXiv preprint arXiv:1806.10254* (2018).
[49] Ian Rae, Eric Rollins, Jeff Shute, Sukhdeep Sodhi, and Radek Vingralek. 2013. Online, Asynchronous Schema Change in F1. *Proc. VLDB Endow.* 6, 11 (aug 2013), 1045–1056.
[50] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: Serializable, Low-Latency, Geo-Replicated Transactions. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1747–1761.
[51] Kun Ren, Alexander Thomson, and Daniel J Abadi. 2014. An evaluation of the advantages and disadvantages of deterministic database systems. *Proceedings of the VLDB Endowment* 7, 10 (2014), 821–832.
[52] Pingcheng Ruan, Tien Tuan Anh Dinh, Dumitrel Loghin, Meihui Zhang, Gang Chen, Qian Lin, and Beng Chin Ooi. 2021. Blockchains vs. Distributed Databases: Dichotomy and Fusion. In *Proceedings of the 2021 International Conference on Management of Data*. 1504–1517.
[53] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. 386–400.
[54] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Proceedings of the Symposium on Self-stabilizing Systems (SSS '11)*. 386–400.
[55] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.
[56] Alexander Thomson and Daniel J Abadi. 2010. The case for determinism in database systems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 70–80.
[57] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 1–12.
[58] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. 18–32.
[59] 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 (SIGMOD '17)*. 1041–1052.
[60] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
[61] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. 2019. Autoscaling tiered cloud storage in Anna. *Proceedings of the VLDB Endowment* 12, 6 (2019), 624–638.
[62] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792.
[63] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 1629–1642.
[64] 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. *Proc. VLDB Endow.* 11, 10 (jun 2018), 1289–1302.

[65] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for in-Memory Databases. *Proc. VLDB Endow.* 9, 6 (2016), 504–515.