GeoGauss: Strongly Consistent and Light-Coordinated OLTP for Geo-Replicated SQL Database

WEIXING ZHOU and QI PENG, Northeastern University, China
ZIJIE ZHANG, Huawei Technology Co., Ltd, China
YANFENG ZHANG*, Northeastern University, China
YANG REN and SIHAO LI, Huawei Technology Co., Ltd, China
GUO FU and YULONG CUI, Northeastern University, China
QIANG LI, Huawei Technology Co., Ltd, China
CAIYI WU, SHANGJUN HAN, and SHENGYI WANG, Northeastern University, China
GUOLIANG LI, Tsinghua University, China
GE YU, Northeastern University, China

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 acknowledgments (*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 strongly consistent OLTP 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. By leveraging an epoch-based delta state merge rule and the optimistic asynchronous execution, `GeoGauss` ensures strong consistency with light-coordinated protocol and allows more concurrency with weak isolation, which are sufficient to meet our needs. Our geo-distributed experimental results show that `GeoGauss` achieves 7.06X higher throughput and 17.41X lower latency than the state-of-the-art geo-distributed database CockroachDB on the TPC-C benchmark.

CCS Concepts: • **Information systems → Relational parallel and distributed DBMSs**.

Additional Key Words and Phrases: Geo-distributed; multi-master replication; replica consistency; transaction processing; deterministic databases

ACM Reference Format:
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, and Ge Yu. 2023. GeoGauss: Strongly Consistent and Light-Coordinated OLTP for Geo-Replicated SQL Database. *Proc. ACM Manag. Data* 1, 1, Article 62 (May 2023), 27 pages. https://doi.org/10.1145/3588916

*Yanfeng Zhang is the corresponding author.

Authors' addresses: Weixing Zhou, Qi Peng, Yanfeng Zhang, Guo Fu, Yulong Cui, Caiyi Wu, Shangjun Han, Shengyi Wang, Ge Yu, Northeastern University, No. 195, Chuangxin Road, Hunnan District, Shenyang, Liaoning, China, 110169, {zhouwx@stumail, ffpengqi@stumail, Zhangyf@mail, yuge@mail}.neu.edu.cn; Zijie Zhang, Yang Ren, Sihao Li, Qiang Li, Huawei Technology Co., Ltd, Xian, Shanxi, China, {zhangzijie9, renyang1, sean.lisihao, liqiang199}@huawei.com; Guoliang Li, Tsinghua University, 30 Shuangqing Road, Haidian District, Beijing, China, liguoliang@tsinghua.edu.cn;

2836-6573/2023/5-ART62 $15.00
https://doi.org/10.1145/3588916

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 telecom service providers have deployed their ICT databases under a geo-distributed setting. The design goals are towards high availability, strong consistency, and high performance.

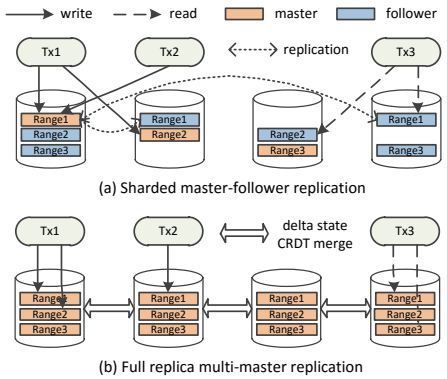


Fig. 1. Sharded master-follower replication vs. full replica multi-master replication.

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 [41], F1 [59], CockroachDB [67], YugabyteDB [22], TiDB [55], SLOG [61] and ConfluxDB [38] adopt a *sharded master-follower replication* architecture as shown in Figure 1a. Data are partitioned into multiple shards according to the 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 increase write throughput.

The sharded master-follower replication architecture is widely adopted [22, 41, 55, 61, 67], 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 [67], it still hurts performance, especially for applications without locality property. 2) The sharded architecture relies on the two-phase commit (2PC) protocol to ensure atomicity. This requires multiple round-trip acknowledgments between the coordinator and the globally distributed workers, which further hurts performance.

Yet another choice for data replication is *full replica multi-master* architecture as shown in Figure 1b, 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, it can serve users with local writes/reads. With a full replica, 2PC is unnecessary to ensure atomicity. A number of multi-master systems emerge in recent years, *e.g.,* Aurora [71], Riak [19], Calvin [70], FaunaDB [9], Anna [73], Aria [51] and Q-store [57]. However, to employ multi-master 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 *Conflict-Free Replicated Datatypes* (CRDT) [65, 73]. Specifically, we adopt an optimized state-based CRDT, *delta-state CRDT* [27, 28], where only recently applied updates to a database state are disseminated instead of the entire database state. The state updates along with its local timestamp information are exchanged among all replicas. At the receiver side, the write-write conflicts are automatically merged by a function that joins any pair of replica updates and must be *associative*, *commutative*, and *idempotent* (*i.e., ACI property*) [65]. The database state at any node is *monotonically* increased by merging the state updates according to the same merge function, so that these replicas are updated independently without a global coordinator. The ACI property of the merge function guarantees that the consistency of replicas can be eventually established after merging of all updates. However, eventual consistency is not acceptable in most transactional applications.

Challenge 2: Strong Consistency. It is desirable to respond a committed/aborted status to the user as fast as possible, but this is not allowed before all replicas reach consistency. Linearizability, as one of the strongest single-object consistency (*i.e.,* all replicas reach the same state after every operation with real-time constraints), requires expensive coordination, while eventual consistency (*i.e.,*, all replicas reach the same state with no time constraint) which can be achieved by CRDT prompts performance without coordination but may incur various anomalies. We address this challenge by introducing *epoch-based merge*, a compromise proposal (between real-time synchronous merge and asynchronous merge) that guarantees consistency at the granularity of *epochs*. Unlike 2PC, where each transaction has a coordinator, we only coordinate at the granularity of epochs, so the coordination is amortized for a set of transactions. After collecting the updates of an epoch from all nodes, the identical set of updates are merged on each node according to the same function with ACI property, and then applied to the previously consistent local replica state. Thus, a globally consistent snapshot can be reached among all nodes on a per-epoch basis.

Challenge 3: Performance. However, by employing the epoch-based merge, the synchronization barriers that require to receive updates from all peers are introduced (the only coordination), which hurts performance a lot. To address this limitation, we employ optimistic execution under a multi-master setting. The master nodes *optimistically* execute their local SQL requests of epoch i based on the current database state, which is not necessarily the most recent epoch snapshot ($i - 1$). The resulting write sets (*a.k.a.* updates) are exchanged among nodes for epoch-based conflict merge. The merge results are used for generating a new snapshot i. Only those successfully validated transactions are allowed for commitment. In other words, the transaction execution phase is performed *asynchronously* among nodes regardless of epoch concept, while the validation phase has to be performed *synchronously* among nodes based on the most recent snapshot. On the other hand, considering that the mainstream sharded master-follower systems [22, 41, 67] support *serializability* (which executes a transaction's logic as a single unit), the optimistic execution allows for overlapping the read and write sets across transactions with weak isolation and as a result brings more concurrency and performance. This greatly meets the requirements of ICT databases for telecom providers on strong replica consistency and high throughput where weak isolation is sufficient in most scenarios.

By integrating the above techniques, we propose *multi-master OCC*, an epoch-based optimistic transaction processing scheme under a multi-master setting. It unifies data replication with optimistic concurrency control, supporting multiple isolation levels with high concurrency while

Table 1. Summary of replicated systems

System	Model	Replication	Rep. Unit	Ordering	Consistency	CC	Isolation
openGauss [15]	SQL	master-follower	redo log	TSO	semi-sync. (eventual)	MV2PL	SI
openGauss MOT [34]	SQL	master-follower	redo log	TSO	semi-sync. (eventual)	OCC	RR
TiDB [55]	SQL	master-follower	binary log	TSO	quorum (linearizability)	MV2PL	SER
Spanner [41]	SQL	master-follower	redo log	TrueTime	quorum (linearizability)	MV2PL	SER
CockroachDB [67]	SQL	master-follower	binary log	HLC	quorum (linearizability)	MVTO	SER
HBase [2]	KV	master-follower	HFile	local time	semi-sync. (eventual)	MV2PL	RC
DynamoDB [43]	KV	masterless	KV	vector clock	quorum (sequential)	last write wins	None
Cassandra [50]	KV	masterless	KV	local time	quorum (sequential)	last write wins	None
Bitcoin [54]	ledger	multi-master	block of txs	PoW	quorum (sequential)	serial	SER
Fabric [33]	ledger	multi-master	block of txs	ordering service	quorum (sequential)	MVOCC	SER
Anna [72]	KV	multi-master	KV	commutative	CALM/CRDT (eventual)	conflict free	RC
Redis CRDT [18]	KV	multi-master	KV	commutative	CRDT (eventual)	conflict free	None
Calvin [70]	SQL*	multi-master	batch of SQLs	local time	deterministic (sequential)	ordered locks	SER
Aria [51]	SQL*	multi-master	batch of SQLs	local time	deterministic (sequential)	dep. graph	SER
GeoGauss (ours)	SQL	multi-master	batch of write sets	local time	epoch CRDT (sequential)	multi-master OCC	SI

guaranteeing strong replica consistency at the granularity of epochs. Furthermore, we build a multi-master geo-replicated OLTP database GeoGauss by modifying openGauss MOT [34], an in-memory storage engine that is highly optimized for multi-core processors. GeoGauss supports a full-featured SQL engine with delta-state CRDT merge. To the best of our knowledge, GeoGauss is the first to integrate CRDT technique into commercial SQL databases with full SQL support.

To sum up, we make the following three key contributions.

- **Epoch-based Multi-Master OCC.** By relying on delta-state CRDT and epoch-based replication, we propose a multi-master OCC protocol that supports light-coordinated high throughput transactions with strong consistency and supports multiple isolation levels.
- **GeoGauss Database.** We develop a geo-replicated OLTP database GeoGauss with full SQL support and rich optimizations (*e.g.,* high concurrency, efficient communication, pipelining, and fault tolerance) to adapt to a geo-distributed environment.
- **Extensive Experiments.** We conduct extensive experiments under a geo-distributed environment with YCSB and TPC-C benchmarks. We compare GeoGauss with CockroachDB (CRDB) [67], Calvin [70], Aria [51], CalvinFS [69], Q-Store [57], SLOG [61], and a coordination-free KV database Anna [72]. Our results show that GeoGauss achieves 1.11X-7.06X higher throughput and 2.28X-17.41X lower latency than these competitor systems on the TPC-C benchmark and 0.44X-87.36X higher throughput and 0.27X-30.94X lower latency on a medium-contention YCSB benchmark.

2 BACKGROUND

Geo-replicated 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 multiple 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 (*e.g.,* openGauss [15]) employ master-follower replication mainly for redundant backup support, while another set of systems employ master-follower replication for scaling read throughput (*e.g.,* CockroachDB [67], ConfluxDB [38]) by allowing follower replica to serve read requests. However, it is required to route all users' write requests to the single master server, which is unsuited for

geo-distributed databases 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 (*e.g.,* DynamoDB [43], Cassandra [50]). 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 [1, 3, 7, 8, 10, 44, 53], all nodes can serve both read and write requests for their local users. It is naturally adapted to geo-distributed requirements.

In addition, supporting distributed transaction processing on Byzantine Fault Tolerant (BFT) and Crash Fault Tolerant (CFT) clusters has been studied in blockchain systems. For example, ResilientDB [47] proposes a hierarchical multi-master architecture for geo-scale deployments. And there also exist many sharding blockchains which provide scalability by grouping nodes into clusters and partitioning data among several independently-run clusters, *e.g.,* AHL [42], Caper [31], SharPer [32], RingBFT [60] and ByShard [49]. Most of these works focus on optimizing the performance of cross-shard transactions which is also a bottleneck in distributed DBMSs [48]. As an alternative, dynamically transferring the mastership of data to avoid expensive multi-shard coordination can improve the performance, *e.g.,* DynaMast [25] and MorphoSys [26].

2.1.2 Ordering and Consistency of Replicas. With master-follower architecture, some databases use replication only for backup (*e.g.,* MySQL [14], HBase [2], and openGauss [15]). These systems use different ordering techniques due to different requirements, *e.g.,* openGauss relies on a centralized *timestamp oracle (TSO)* service to achieve snapshot isolation, HBase uses local time for identifying the version of a value. To maximize performance, they are configured with asynchronous replication or semi-synchronous replication [20] (using synchronous replication for a subset of the followers). This means that the consistency of all replicas may be not guaranteed at a certain time point (eventual consistency). Another set of sharded databases with master-follower replication provide serializability and linearizability. For example, Google Spanner [41] relies on *TrueTime* , CockroachDB [67] relies on *hybrid logical clock (HLC)*, and OceanBase [46] and TiDB [55] rely on a centralized *timestamp oracle (TSO)* service. The replica consistency is guaranteed with leader-based protocols, *e.g.,* Paxos and Raft. ISS [66] was recently proposed to improve these leader-based consensus protocols by efficiently multiplexing consensus instances for scalability.

Many NoSQL KV databases adopt masterless replication without stable master and use quorum protocol to ensure consistency, such as DynamoDB [43] detects updated conflicts by vector clock and Cassandra [50] uses the local time to identify the version of a value. However, this *quorum-based* approach without a stable master pays for it 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 [43], Riak [19], and Cassandra [50].

For multi-master architecture, MySQL and PostgreSQL both provide tools for cross-site multi-master replication, *e.g.,* MySQL Tungsten [13] and PostgreSQL BDR [16], but they use asynchronous replication and fail to guarantee strong consistency. The blockchain system is a particular multi-master replication system with Byzantine-fault tolerance. The key to ensuring consistency is to reach a consensus on the order of transactions under a trustless environment. For example, permissionless blockchain Bitcoin [54] employs *Proof-of-Work* (PoW), permissioned blockchain Fabric [33] leverages an ordering service to determine a global order of transactions. In this way, these blockchains can achieve sequential consistency and serializable isolation.

Different from others, coordination-free replication, *e.g.,* Berkeley Anna [72] and Redis CRDT [18], utilizes mathematical properties of operations (*i.e., associative*, *commutative*, and *idempotent*) to reach replica consistency with the out-of-order transactions as input on different replicas. Typical works include *Consistency As Logical Monotonicity (CALM)* [29, 30, 35, 36, 39] and *Conflict-free*

Replicated Data Type (CRDT) [28, 56, 64]. 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 identical received set of transactions are executed in batches in a deterministic order. Before execution, all nodes have made an agreement on the ordering rule, *e.g.,* according to the globally unique ID (local time + server ID). Essentially, it guarantees sequential consistency and supports serializable isolation, *e.g.,* Calvin [70], Aria [51], `CalvinFS` [69] and `Q-Store` [57].

2.1.3 Replication with Transaction Processing. Traditional concurrency control techniques, such as *multi-version two-phase locking* (MV2PL) and *multi-version timestamp ordering* (MVTO), 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 follow the last-write-win rule to resolve write-write conflicts.

For multi-master replication, there exist various conflict resolution approaches for concurrent updates. The most special one is Anna [72] 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 [54], the single node that first solves a PoW puzzle executes transactions serially without concurrency. In Fabric [33], all peer nodes use *multi-version optimistic concurrency control (MVOCC)* to execute their local transactions parallelly, then based on the order consensus they validate their executions and abort conflict transactions. Deterministic databases execute transactions according to a predefined serial order, which have limitation on concurrency since the operating system schedules concurrently running threads in a fundamentally non-deterministic way. Existing deterministic databases rely on dependency graphs (Aria [51]) or ordered locks (Calvin [70]) to provide more parallelism while ensuring determinism.

2.2 Requirements

Despite many distributed databases having been proposed, the demand of top telecom customers drives us to propose a new OLTP database that fulfills the following requirements.

- **Multi-Master Architecture.** As motivated in Section 1, multi-master architecture is more suited for a geo-distributed setting, which offers low latency service.
- **Full-Featured SQL Engine.** High SQL coverage and interoperability are essential for a commercial database and can meet the needs of various users, *e.g.,* OLTP, OLAP, and stored procedures.
- **Strong Consistency.** As a geo-replicated data system, 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, *e.g.,* online retail and global trading.
- **High Performance with Weak Isolation.** Many ICT databases used by telecom service providers are deployed under geo-replicated settings, such as Operation Support Systems (OSS), Customer Relationship Management (CRM), and Enterprise Resource Planning (ERP). They demand high throughput and low latency but do not require strong isolation. It is sufficient to support weaker isolation 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. `GeoGauss` is developed based on a relational database openGauss [15]. We rely on its SQL engine

to parse SQL statements and generate physical execution plans and use its row-based memory-optimized storage engine openGauss MOT [34] to store data (other in-memory row-stores with concurrency support are possible alternatives). The local SQL requests are optimistically executed based on the local current database state. Only the write sets (*a.k.a.* updates) obtained after local SQL execution are exchanged between nodes, which will be merged with local updates at the receiver side. The merged updates are then applied to each local replica. Some key features are described as follows.

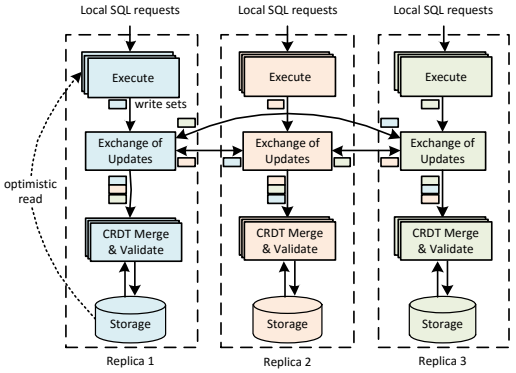


Fig. 2. Overall structure.

Multi-Master Architecture with Full SQL Engine. GeoGauss provides a full SQL engine, supporting standard SQL, application programming interfaces, interoperability, *etc.* GeoGauss is deployed on 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 execution engine. Each regional server processes its local SQL requests independently based on its local replica.

Optimistic Read on Local Replica. Read-only transactions are directly served with a snapshot read result for low latency. For transactions that contain both read and write operations, we perform optimistic read on 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 performance under weak isolation levels.

Epoch-based Replication of Updates. All master nodes exchange their local updates periodically (every epoch). The *epoch* is a short period of time (*e.g.,* 10 ms), and the epoch number is monotonically increased according to local physical time. Different from deterministic databases that perform replication of the SQL statements, GeoGauss exchanges the *write sets* between masters, which can be considered as the *delta state* in state-based CRDT merge. We provide an illustrative example in Figure 3 showing how it works under two replicas setting. 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 (*a.k.a.* updates) are exchanged between replicas at the end of every epoch. After a node receives all remote updates committed at the first epoch, these updates are merged with local write sets to resolve write-write conflicts based on a *CRDT merge rule*, which guarantees the same merge result given the identical set of updates *regardless of arriving order*. On each node, the identical merge outcome is 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 merge operation. Only when the local snapshot for epoch i is generated (after merge and validation), can the local transactions with

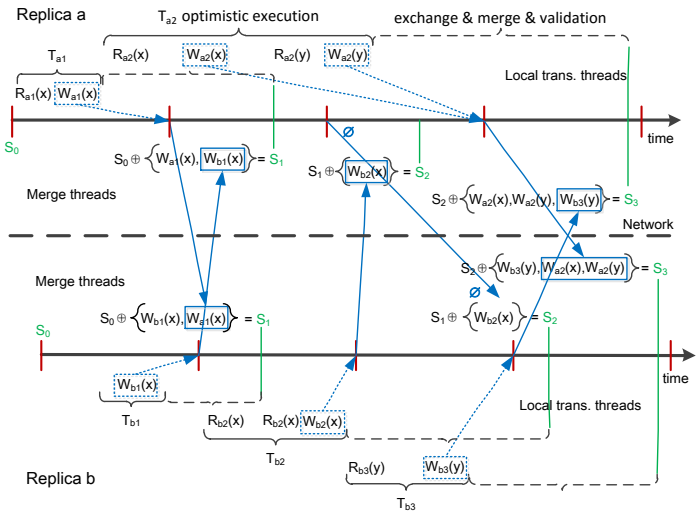


Fig. 3. An illustrative example of update merge with two replicas. Each replica a independently accepts local transactions, *e.g.,* T_{a1} and T_{a2}. $R_{a1}(x)$ and $W_{a1}(x)$ represent the read and write operations on data item x of replica a, respectively. A local transaction with commit epoch number cen cannot be confirmed until snapshot cen is generated, so there exists a period for receiving remote updates, merge, and validation. Red bars are epoch boundaries determined by the local clocks, which are not necessarily synchronized across replicas. Green S_i is the globally consistent snapshot. The blue dotted/solid box is the sent/received write set.

commit epoch number i be returned by the host master with committed/aborted notification, *e.g.,* T_{a2} which is committed at epoch 3 will not return until S_3 is generated on replica a, at which time replica a has received and merged the remote updates that are committed at epoch 3, $W_{b3}(y)$.

CRDT Merge with Transaction Processing. 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. 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 (collected from local and all other remote nodes), at which time the local transactions of epoch i can be returned with committed/aborted response. We rely on delta-CRDT to efficiently resolve cross-region conflicts and achieve a globally consistent state. With a merge operation that has ACI properties, these writes from different replicas can be partially merged to improve efficiency (Associative), can arrive in different orders (Commutative), and even can be retransmitted (Idempotent). The replica state after merging local/remote updates is guaranteed to be *consistent* among all nodes.

Asynchronous Execution and Synchronous Validation. A 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 has to be synchronized. The validation of epoch i's transactions is performed based on the globally consistent snapshot $(i-1)$, where the previously executed transactions that have write-write conflicts with others should be aborted (by checking whether a transaction's pre-write is overwritten by others). As a conventional way, 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 transaction processing, which will be discussed in Section 4.

Algorithm 1: Local Transaction Process (a Thread per Transaction)

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

4 EPOCH-BASED MULTI-MASTER OCC

4.1 Lifecycle of a Local Transaction

A transaction is submitted from the client to the local server, where a *thread* is assigned for the transaction. **Algorithm 1** describes 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 the current time, which is used for read set validation in snapshot isolation. The transaction is then executed and generates read set *RS* and write set *WS* (Line 2-3). The SQL constraints are checked during the transaction execution. If a constraint violation occurs, the transaction will abort before generating the write

set. In the case of a read-only transaction, it reads data on the most recent snapshot and returns after read validation(Line 19-20).

For the transactions that contain write operations, our multi-master OCC requires each transaction to record a few meta information for CRDT merge. Specifically, a transaction is assigned with a *commit epoch number* (cen) and a *commit sequence number* (csn) (Line 4). 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 several ANSI isolation levels, *e.g., Read Committed* (RC), *Repeatable Read* (RR), and *Snapshot Isolation* (SI). They are different in processing logic (Line 5-18).

For a transaction with T.cen, a synchronization point is placed to wait for snapshot (T.cen − 1) to be generated (Line 22). 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 impact correctness and consistency (Section 4.4).

The `DeltaCRDTMerge` function is then launched (Line 23), which defines the rule for merging a new update (*a.k.a.* delta state) into the current database state (see Section 4.2 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 is 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 26-29). This implies that there exists a write-write conflict on the same row. According to the merge rule, only one write wins and is committed, and the others 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 24). This is essential to the correctness by ensuring none of the updates are missing. If the transaction does not meet a write-write conflict or wins in conflict merge, it is allowed to commit. This transaction's write data are used to update the involved rows (Line 31-33). After all transactions with the same commit epoch number cen are validated and applied on the table, a new snapshot for epoch cen is generated.

4.2 Merge of Updates

4.2.1 Epoch-Aware Delta CRDT Merge. CRDTs [63, 65] are distributed datatypes that allow different replicas of a distributed CRDT instance to diverge and ensure that, eventually, all replicas converge to the same state. State-based CRDTs achieve this by propagating updates of the local state by disseminating the entire state across replicas. The received states are then merged to remote states, leading to convergence (*i.e.,* consistent states on all replicas). In delta CRDT [28], only the updates are disseminated instead of the entire state. A received delta-state (*i.e.,* update) is incorporated with the local full state (*i.e.,* database) via a merge function that deterministically reconciles both states. The delta-states can be shipped using an unreliable dissemination layer that may drop, reorder, or duplicate messages, *i.e.,* delta-states can always be out of order, re-transmitted, and re-joined. Furthermore, it is desired to handle concurrent updates for high performance. Therefore, the key is the design of the merge function which must be associative, commutative, idempotent, and parallel-friendly, and most importantly be aware of epoch information.

Our merge function, `DeltaCRDTMerge`, is depicted in **Algorithm 2**. It defines the conflict merge rule to determine the successful updates and may be concurrently invoked by multiple local transactions and remote transactions. It relies on a data structure, *row header*, to support epoch-based delta-state merge, which stores each updated row's {sen, lsn, csn, cen}. For ease of illustration,

Algorithm 2: DeltaCRDTMerge

```
   Input: a transaction's T.{sen, csn, cen, WS}, current row headers
   Output: updated row headers
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.{sen, csn, cen} = T.{sen, csn, cen};    //row is not pre-written in current epoch
7      else if row.cen == T.cen then
8          if row.sen == T.sen then
9              if row.csn > T.csn then
10                 row.{sen, csn, cen} = T.{sen, csn, cen};    //first write wins
11             else
12                 Abort T;
13         else if row.sen < T.sen then
14             row.{sen, csn, cen} = T.{sen, csn, cen};    //shorter transaction wins
15         else
16             Abort T;
17     else
18         //row.cen > T.cen will never happen
```

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 the 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 the current epoch by other threads, *i.e.,* row.cen == T.cen. Note that, since the `DeltaCRDTMerge` will never be invoked to merge updates of epoch $(i + 1)$ before completing the merge of all updates of epoch i (the synchronous point at Line 22 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 13-14). 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 insertion conflicts within the same epoch. Instead of invoking `FindRow` which relies on the table index, we use a temporary table for the inserted rows.

The `DeltaCRDTMerge` operation defined in **Algorithm 2** is similar to the first-write-win conflict resolution in OCC, but they have different design goals. Our merge function that has ACI property is designed for merging the local/remote updates on separate nodes to achieve a globally consistent state (*i.e.,* replicating states), even though these updates arrive at each node in different orders (Commutative property), are merged partially (Associative property), are retransmitted (Idempotent property). But the first-write-win rule in OCC is for deciding the committed and aborted transactions in concurrent transaction processing on a single replica.

4.2.2 Handling Remote Updates. The merge of remote updates and the local transaction processing are running concurrently and interacting with each other. On each master node, multiple receive threads and merge threads are continuously running as shown in **Algorithm 3**. The receive thread

Algorithm 3: Epoch-based Multi-Master OCC

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

```
1  Receive thread:
2  while true do
3      TS ← receive a batch of updates with epoch TS.cen;
4      BUF[TS.cen].add(TS);  //add TS to buffer BUF indexed by cen
5  Merge thread:
6  while true do
7      lsn ← get latest consistent snapshot no.;
8      TS = BUF[lsn + 1].get();  //blocking get
9      foreach T in TS do
10         DeltaCRDTMerge(T.{sen, csn, cen, WS});
11         if T is not aborted then
12             Q[lsn + 1].push(T);  //add T to commit queue Q
13     N[lsn + 1] ← N[lsn + 1] + 1;  //counts the executed remote TS
14     if N[lsn + 1] == n − 1 then
15         //all remote updates have been processed
16         NotifyAll({Thread[T] | T.cen = lsn + 1});  //notify all threads blocked at Line 24 (Alg. 1)
17         foreach T in Q[lsn + 1] do
18             Execute Line 26-33 of Alg. 1 for T;  //validate and write
19         wait till all local threads(Thread.T.cen == lsn + 1) are finished;
20         NotifyAll({Thread[T] | T.cen = lsn + 2});  //notify all threads blocked at Line 22 (Alg. 1)
21         lsn ← lsn + 1;  //a new snapshot is generated
22 Local transaction processing primary thread:
23 while true do
24     Fork a new thread Thread[T] for a local transaction T;
25     Thread[T].start();  //Algorithm 1
```

keeps receiving updates TS from remote peers and temporarily stores them in the receive buffer with their commit epoch number T.cen information (Line 3-4). Suppose lsn is the number of the most recent globally consistent snapshot (Line 7). The merge thread merges updates of epoch (lsn + 1) based on the most up-to-date table state (*i.e.,* snapshot lsn) and will produce consistent snapshot *one by one*. Specifically, if there exist remote updates of epoch (lsn + 1) in the buffer, it processes them by invoking the `DeltaCRDTMerge` function (Line 8-10). The transactions that are not aborted are pushed into a commit queue $Q[\text{lsn} + 1]$ (Line 11-12). If the updates of epoch (lsn + 1) from all remote peers have been applied on row headers, it notifies all the local transaction threads for epoch (lsn + 1) and triggers the validation and write-back for each update in the commit queue (Line 16-18). Once all the local threads of epoch (lsn + 1) are finished, it means that a new consistent snapshot is achieved. The merge thread immediately notifies all the local transaction threads of the next epoch that are waiting for this up-to-date snapshot (Line 19-21). We also have a primary thread for assigning a new thread for each new local transaction (Line 23-25).

4.2.3 Case Study. **1) Long running Transactions.** Only when a transaction commits at epoch i can we send its write sets at the end of epoch i. For example in Figure 3, T_{a2} is a long running transaction that crosses three epochs and ends at epoch 3. If the transaction is not aborted in the

read validation phase, its write sets $\{W_{a2}(x), W_{a2}(y)\}$ are sent out together at the end of epoch 3. Note that, the transactions that start at the same epoch are not necessarily completed synchronously before the next epoch can start. They may finish execution and commit at different epochs, so the short transactions do not wait for the long transactions but can commit at earlier epochs. A long transaction (spanning multiple epochs) is more likely to be aborted due to inconsistent read in RR or stale read in SI, where the read data might be modified by early committed short transactions. Nevertheless, if there are 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. **2) Network Delay.** For example in Figure 3, due to network latency, the two replicas probably may not reach a consistent snapshot exactly at the same time point, but they will at a certain time. This will NOT block the system running and will NOT impact sequential consistency (see Section 4.4). 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 22 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.

4.3 Isolation

Weak isolation models cannot guarantee serializability, but their benefits to concurrency are frequently considered by application developers to outweigh the costs of possible consistency anomalies that might arise from their use. GeoGauss supports multiple weak ANSI 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 a 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 21). 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 an *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 11-12). 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 13-14). 3) If a transaction's read row is updated in a new snapshot (snapshot row.cen $-$ 1 due to the synchronous point at Line 22) 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 15-16). Only the transactions that pass read set validation will be sent out (Line 21). For example in Figure 3, 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.

Serializable Snapshot Isolation (SSI). We discuss the possibility to support SSI [37]. We need to exchange each read record's key for read-write dependency detection. Then we can prevent snapshot isolation anomalies by aborting a transaction when a pair of consecutive conflict edges are found. Currently, GeoGauss do not support SSI due to its high cost for transferring the read keys.

4.4 Consistency of Replicas

We show how GeoGauss guarantees the consistency of replicas.

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

Proof. In our system, each master node performs a 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 the consistency of replicas. □

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

Proof. The merge rule (Algorithm 2) is defined by comparing tuples {sen, csn, cen}. We define a strict total order '$\prec$' on the set of updates TS as follows. For any two updates $\mathcal{U}(T_i)$ and $\mathcal{U}(T_j)$ with the same cen (*i.e.,* $T_i.\text{cen} = T_j.\text{cen}$), we have $\mathcal{U}(T_i) \prec \mathcal{U}(T_j)$ if one of the following conditions is met.

- $T_i.\text{sen} > T_j.\text{sen}$;
- $T_i.\text{sen} = T_j.\text{sen}$ and $T_i.\text{csn} < T_j.\text{csn}$.

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 '$\prec$') to commit. That is, the collection of successful updates always equals to $\left\{\{x \in \mathcal{WS} : T_i \in C(x)\} \mid \mathcal{U}(T_i) \prec \mathcal{U}(T_j), \forall T_j \in C(x)\right\}$, 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 (even for duplicated updates), 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. *The epoch-based 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 the consistency of replicas, so we only focus on studying the effects of write operations. Lemma 2 states that given a batch of updates and an initial state, the output state is *deterministic* regardless of the non-deterministic 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 24 in Algorithm 1 and at Line 19 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 22 in Algorithm 1 and the latest snapshot number verification at Line 7 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 is guaranteed at the granularity of epochs. □

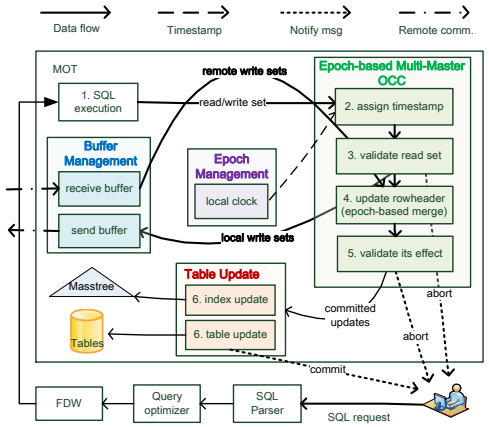


Fig. 4. Data flow of GeoGauss.

The read operations on the local replica are not exchanged among replicas. The order of write operations in an epoch is deterministic and consistent on each individual replica (Lemma 2). The epoch snapshots on all replicas are generated one by one. Thus, the consistency enforced by our algorithm is a kind of *sequential consistency*.

5 IMPLEMENTATION

We implement `GeoGauss` based on openGauss 2.0 MOT storage engine [15]. 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 22 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 [17] and Gzip [11] to compress the buffered write sets. Instead of the heavyweight gRPC [12] used by openGauss, we prefer the lightweight ZeroMQ [23] (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 a pipelining technique that sends write sets in mini-batches in a streamlined manner. Note that, an EOF message is sent indicating the end of the epoch (according to physical time) in streamlined communication, so the receiver can ensure the completeness of an epoch of transactions. Furthermore, we improve the pipelining by zero-copy send and receive. 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 the operating system. We copy the write sets data through zero-copy and directly send them out through Protocol Buffers. This can alleviate the serialization and deserialization costs.

5.2 Fault Tolerance

Under a multi-master architecture, once a master node fails to provide service, its local transaction requests can be routed to another master node. We employ Raft protocol [5] to make a consensus on the status of live nodes, which can prevent permanent blocking (the whole system may be blocked waiting for the write sets from a failed node). This is light weight since it is invoked only when the status of alive nodes is changed. Under such an epoch-consistent replicated system, multiple master nodes work as strongly consistent replicas, so there is little risk that all master nodes fail simultaneously. We can also place more replicas for backup to increase robustness as existing commercial databases do. However, there is a unique problem that should be tackled under such architecture. That is, we should prevent the loss of updates problem, *i.e.,* some updates successfully commit on the local node but fails on remote nodes. To avoid such faults, `GeoGauss` provides three fault tolerance options, from light to heavy-weighted.

Local WriteSet Backup Server. We place one (or more) writeset backup servers associated with each replica in each region, caching the generated local write sets for backup. Each time a server node sends local write sets to remote nodes, it also sends a copy to the local writeset backup server, which will send back an ack message as a response. Only when all remote write sets with commit epoch number cen have been collected and the local write sets with cen have been backed up, 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 nodes will ask the corresponding writeset backup server to check whether there exists a loss of updates by comparing the monotonic cen. If so, the remote nodes 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 performance.

Remote WriteSet Backup Server. In case a region fails, the local backup server will fail together, and the above scheme may not work. Similarly, we can place one (or more) remote backup servers in other regions for caching the write sets. In this case, it requires one cross-region round-trip-time (RTT) for sending backup updates and receiving the ack message. This is more expensive than the local backup method, which requires only 0.5 cross-region RTT for receiving the remote updates. This solution can ensure fault tolerance when one or more regions fail.

Raft Replication of WriteSets. To maximize the fault tolerance, the write sets can be replicated through Raft consensus protocol on a per-epoch basis. This is similar to the deterministic databases which rely on Raft to replicate SQL inputs. Each node (as a Raft leader) sends the generated write sets to other receiver master nodes (as Raft followers), who will send back ack messages. If the leader collects more than half of the ack messages, it sends a commit request to the receivers indicating that the write sets can be applied to the existing database state. Therefore, the receiver requires ~1.5 RTTs to receive the write sets from the sender node.

6 DISCUSSION

Transaction-based vs. Batch-based vs. Epoch-based. Traditional wisdom prefers to handle transactions on a per-transaction basis [41, 67]. This is suboptimal under a 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 the atomicity and consistency of replicas. An alternative is batch-based processing (*e.g.,* deterministic databases [51, 70]) that limits the number of updates for each batch instead of a 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. In addition, the epoch-based and batch-based approaches are less expensive for fault tolerance since the consensus and durability are established on a set of transactions instead of an individual transaction.

Epoch Length. Intuitively, the epoch length should be set longer than the round trip time (RTT). But it is not necessary due to the deterministic execution with the coordination-free property. Unlike the coordination-based approach (*e.g.,* 2PC) that relies on the 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 the 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. 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 7.5.1).

Advantages over Deterministic Databases. Existing *deterministic databases* [9, 24, 51, 52, 57, 61, 62, 68–70] 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 [24]. This might require a locking mechanism to achieve concurrency while ensuring deterministic serial order, but this results in high scheduling overhead. A number of approaches are proposed to reduce the scheduling overhead, *e.g.,* Aria [51], QueCC [58], PWV [45], and LADS [74]. However, there exist several disadvantages of deterministic databases, including 1) lack of support for interactive SQL because it is required to disseminate the SQL statements before executing them, which limits their application; 2) the additional scheduling overhead for determinism; 3) undesirable performance due to imbalanced workloads among transactions, especially for long running transactions (because a previous batch of transactions must finish executing before a new batch can begin). Compared with deterministic databases which exchange SQL statements, `GeoGauss` which disseminates replica-generated write sets has distinct advantages. 1) We support a full SQL engine (*e.g.,* interoperability) by disseminating the write sets when a transaction commits. 2) With our CRDT merge rule for write sets (Algorithm 2), we do not need additional scheduling overhead for ensuring the determinism for concurrent processing. 3) In `GeoGauss`, transactions are optimistically executed and arranged into batches *according to their commit time* for synchronous validation. It is allowed to process new transactions during the execution of long transactions, so the impact of long transactions is greatly alleviated.

7 EVALUATION

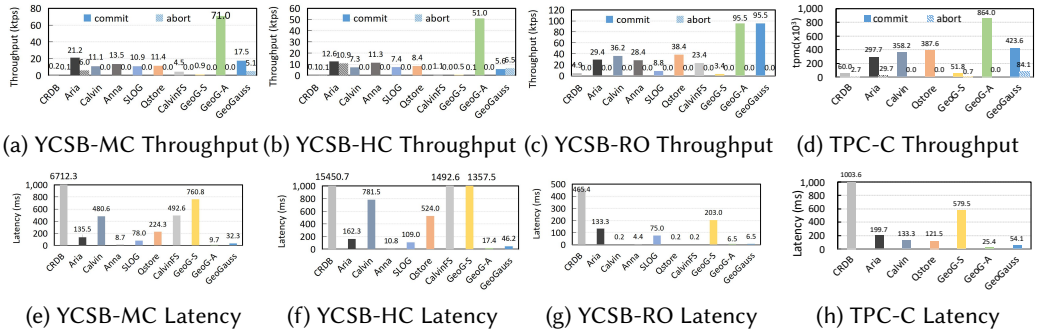
This section evaluates `GeoGauss` through cross-region experiments.

Cluster Setup. Our cross-region cluster contains 3 geo-distributed nodes (corresponding to 3 replicas), including a node in Zhangjiakou city (North China), a node in Chengdu city (Southwest China), and a node in Shenzhen city (South China). Each node (Aliyun ecs.r6e.8clarge instance) is equipped with 32 vCPUs, 256GB DRAM, running Centos 7.6 OS. The network bandwidth between cross-region nodes is about 100 Mbps/s. A separate node (ecs.c6.8xlarge) is set in each region to simulate the local client for sending SQLs.

Competitors and Configurations. We choose CockroachDB (`CRDB`) [67], `Calvin` [70], `Aria` [72], `CalvinFS` [69], `Q-Store` [57], `SLOG` [61], and a coordination-free KV database `Anna` [72] for comparison. To investigate the performance improvement by *optimistic asynchronous execution and synchronous validation*, we also implement two variants `GeoG-S` and `GeoG-A` based on `GeoGauss`.

- `CRDB`: For fairness, `CRDB` is configured with *in-memory store* and configured with *stale reads* from outside the read row's home region. As suggested by `CRDB` documentation, we place 2 additional nodes in each region for maximizing its performance. `CRDB` supports strong consistency and serializable isolation.
- `Calvin`, `Aria`, `CalvinFS`, and `Q-Store`: `Calvin` and `Aria` are two typical deterministic databases with multi-master replication. They replicate SQLs instead of write sets, and they do not provide a full SQL engine, so they do not support interactive queries. They provide strong consistency and serializable isolation. `Calvin` leverages ordered locks to achieve concurrency, and `Aria` relies on dependency analysis and transaction reordering. For these two systems, we use the implementations from [4] and follow their default configurations. `CalvinFS` is a distributed file system that extends `Calvin` to achieve metadata management. We use the implementation from [6] and follow its default configurations. `Q-Store`, which is implemented based on `Calvin`, uses a queue-oriented transaction processing method instead of ordered locks to reduce the scheduling overhead.
- `SLOG`: `SLOG` adopts the sharded master-follower architecture. The input SQLs are contained in logs and are replicated to followers. For cross-shard transactions, they are sent to a single node for deciding a global order, which is used for deterministic execution on each shard replica. The shard replica replays the received logs through deterministic execution to achieve serializability.
- `Anna`: `Anna` achieves wait-free execution via the merge of lattice-based composite data structures (similar to CRDT). It only supports causal consistency and eventual consistency (by default), so a submitted SQL request is not returned with a committed or aborted response. It also supports a variety of weak isolation levels, where RC isolation is configured by default.
- `GeoGauss`: Our system is configured with 10ms epoch length by default. It supports strong consistency and multiple weak isolation levels (RC by default). `CRDB` and `GeoGauss` use a standard benchmark interface, in which each connection only sends a single transaction once at a time. In these two systems, each node is configured with 256 connections.
- `GeoG-S`: It adopts *synchronous execution and synchronous validation*, *i.e.,* it does not start the execution of epoch i's transactions until the snapshot $(i-1)$ is generated.
- `GeoG-A`: It removes epoch concept and adopts *asynchronous execution and asynchronous validation*. Similar to `Anna`, it provides eventual consistency and does not guarantee strong consistency at the granularity of epochs.

Workloads. We use two popular benchmarks, YCSB [40] and TPC-C [21]. For YCSB, we use one table with 10 columns and 1,000,000 rows, and we configure it with 256 connections. To make it a transactional benchmark, we wrap operations within transactions and let each transaction contain 10 operations. We evaluate three different variations of YCSB workload: 1) YCSB-MC (medium contention): 80% reads and 20% writes with a hotspot of 10% tuples that are accessed by ~60% of all queries ($\theta = 0.8$ in Zipfian distribution). 2) YCSB-HC (high contention): 50% reads and 50% writes



(a) YCSB-MC Throughput (b) YCSB-HC Throughput (c) YCSB-RO Throughput (d) TPC-C Throughput

(e) YCSB-MC Latency (f) YCSB-HC Latency (g) YCSB-RO Latency (h) TPC-C Latency

Fig. 5. Comparison results on throughput and latency.

with a hotspot of 10% tuples that are accessed by ~75% of all queries ($\theta = 0.9$). 3) YCSB-RO (read only): all read queries and a uniform access distribution. For the TPC-C benchmark, we configure it with 800 warehouses and 120 client connections for sending query requests. Since `Calvin` and `Aria` do not provide SQL engine (do not support interactive queries and complex queries, *e.g.*, join and range scan), they can only support New-Order transactions and Payment transactions. We follow [51] to make a TPC-C benchmark by mixing 50% New-Order and 50% Payment transactions. `CalvinFS`, `SLOG`, and Anna do not support this TPC-C benchmark, so we do not run the TPC-C benchmark on them.

7.1 Cross-Region Comparison Results

We run comparison experiments on our cross-region cluster. The throughput for successfully committed and aborted transactions and the average latency are reported in Figure 5. `GeoGauss` shows higher throughput over most of the other systems in the YCSB-MC workload, and achieves the highest throughput in read-intensive workloads. `GeoG-A` is faster than `GeoGauss`, but it cannot guarantee strong consistency. Both `Anna` and `GeoG-A` cannot respond to users with committed or aborted notifications since the state is not known with a time constraint (they have no aborted transactions). Strictly speaking, they are not transactional systems. `GeoG-S` is a highly synchronized variant of `GeoGauss`, which guarantees strong consistency but at the expense of performance. `CRDB` is slow due to its expensive coordination cost under a geo-distributed environment. `Calvin` and `Aria` show much higher throughput than `CRDB`. `CalvinFS`, an extension of `Calvin`, uses Quorum protocol to achieve replica consistency, resulting in reduced performance compared with `Calvin`. `Q-Store` reduces the scheduling overhead of `Calvin` with limited performance improvement because the main overhead in a geo-distributed environment is coordination rather than scheduling. `SLOG` shows poorer performance than `Calvin` and `Aria`, because it requires to send cross-shard transactions to a single node for determining the global order, which hurts performance a lot under a cross-region scenario without a locality-aware sharding scheme. The writes and linearizable reads of a data item in `SLOG` must be directed to its master replica, so the throughput on YCSB-RO is similar to that on YCSB-MC. `GeoGauss` and `Aria` exhibit higher abort rates due to their optimistic execution logic. `GeoGauss` leverages optimistic execution based on a stale snapshot and validates its effects before committing, while `Aria` relies on dependency graph analysis to resolve conflicts. These tend to have higher abort rates than pessimistic locking methods, *e.g.*, `Calvin`. Note that, deterministic databases and their extensions do not support the standard TPC-C benchmark. They are limited in interoperability and need to know what data will be accessed in advance.

Regarding the latency, `GeoGauss` exhibits superiority over the other strongly consistent systems on YCSB-MC, YCSB-HC, and TPC-C. `Anna` is a coordination-free KVS with eventual consistency. `GeoG-A` only supports CRDT merge without epoch-based consistency, which only supports eventual

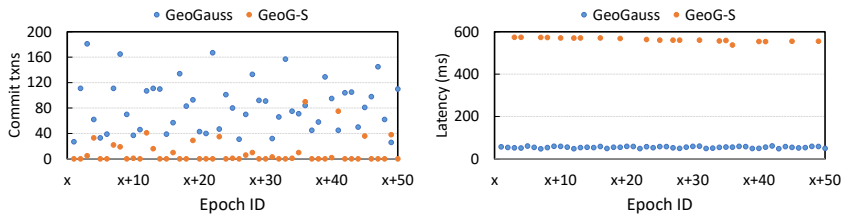
Table 2. Runtime breakdown of a transaction (TPC-C).

	GeoG-S	GeoG-A	GeoGauss
SQL Parse	4.6 ms	4.6 ms	4.6 ms
Execute	5.8 ms	6.5 ms	4.8 ms
Wait	**564.2 ms**	**0 ms**	**34.1 ms**
Merge	4.0 ms	10.9 ms	9.4 ms
Log	0.8 ms	6.5 ms	4.7 ms

consistency. Both of them do not need coordination, so their latency results are lower than the others. CRDB and GeoG-S are with high latency due to the long waiting time under a geo-distributed environment. Calvin and Aria need additional time for scheduling these batched transactions to achieve deterministic execution, so they result in higher latency than GeoGauss. On the YCSB-RO workload, Calvin and its extensions (CalvinFS and Q-Store) show very low read latency because they directly return local read data and do not have the input parsing overhead. Calvin does not offer full SQL support, while the input query in GeoGauss needs to go through SQL parser and optimizer, so the latency in GeoGauss is higher than Calvin and its extensions. Aria requires a preprocessing step to perform dependency analysis for input queries, so Aria exhibits much higher latency than Calvin and its extensions.

7.2 Performance Gain Analysis

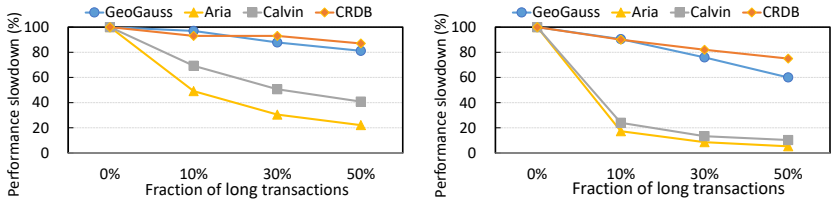
In our system, the transaction processing mainly contains five phases, including SQL parsing, transaction execution, waiting for the most recent snapshot being generated (Line 22 in Algorithm 1) or all remote/local transactions of the same cen being applied (Line 24 in Algorithm 1), merging (Algorithm 2), and logging for duration. We study the cost of different phases to investigate the bottleneck. Table 2 shows the average time spent in each phase of a successfully committed TPC-C transaction. We can see that the waiting phase dominates the runtime in GeoG-S. By leveraging optimistic asynchronous execution (still needing synchronous validation), GeoGauss reduces the wait time dramatically. This is the key to improving the performance while at the same time guaranteeing strong consistency on a per-epoch basis (by synchronous validation). On the contrary, GeoG-A does not wait for synchronization, so it does not provide strong consistency. The merge time and the logging time of GeoG-S are shorter because the processed transactions in GeoG-S are much less than the other two.



(a) Number of commit transactions (b) Latency

Fig. 6. Throughput and latency of each epoch (TPC-C).

As analyzed above, the synchronous execution in GeoG-S which requires processing the transactions of epoch i based on the snapshot $(i-1)$ can greatly impact the performance. To investigate the impact of synchronous execution, we further measure the number of committed transactions and the average latency of each epoch in GeoGauss and GeoG-S. Figure 6a shows the number of committed transactions in a number of consecutive epochs. In GeoG-S, there is no commit transaction in many epochs and there are commit transactions every other 2-4 epochs. This is because the single-trip time is around 30 ms and GeoG-S stops serving when waiting for the remote write



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

Fig. 7. Effect of long transactions (YCSB-MC).

sets of the previous epoch. While GeoGauss which exploits optimistic asynchronous execution can continuously commit transactions. Figure 6b shows the average latency of the epochs that have committed transactions. The latency is much longer than the single-trip delay (~580 vs. ~30 ms) due to the ripple effect.

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-MC 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 both decrease 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. GeoGauss is more robust to long transactions. This is because that transactions in GeoGauss optimistically read data and write updates in their private cache, and new transactions are processed concurrently with the execution of long transactions. 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 though its latency is much longer than ours, because the (manually set) delay is hidden in its *parallel commit* phase.

Table 3. Average WAN traffic per transaction (KB/txn)

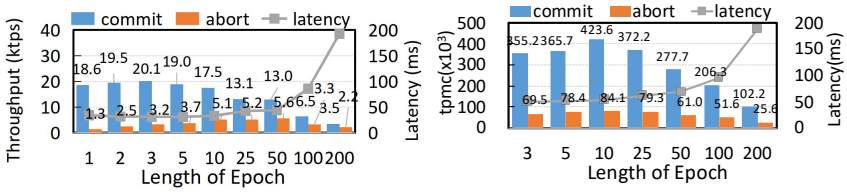
	YCSB-RO	YCSB-MC	YCSB-HC	TPC-C
GeoGauss	0	0.28	0.5	0.6
Calvin	0.13	0.19	0.28	0.24

7.4 WAN Traffic

GeoGauss produces more WAN traffic than deterministic databases, because GeoGauss sends outputs (*i.e.,* write sets) instead of inputs (SQL statements). Table 3 reports the average WAN traffic for each transaction in GeoGauss and in Calvin. The reported numbers are the average size of each transaction after compression by Gzip [11] (see Section 5.1). As the write data size is bigger, transferring the write set results in more network traffic than transferring the input SQLs. However, we find that Calvin cannot fully utilize the WAN bandwidth (only about 25 Mbps on TPC-C, which is less than 100 Mbps WAN bandwidth). The bottleneck of deterministic databases is the scheduling for deterministic execution and the execution cost, especially for long transactions. In GeoGauss, we rely on CRDT for merging write sets to achieve determinism (regardless of arrival order and scheduling order), which is more efficient. Furthermore, we let each replica execute local SQLs first and only exchange outputs, so the execution cost is disseminated.

7.5 Varying Configurations

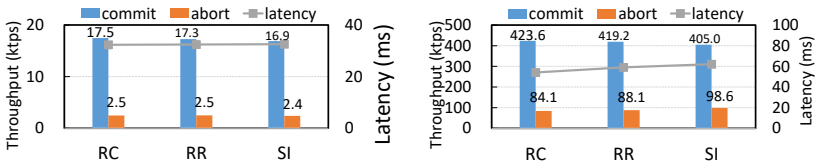
7.5.1 Epoch Length. As discussed in Section 5.1, GeoGauss uses one thread per transaction (*i.e.,* per connection) and will not release the connection until the transaction is committed or aborted. Thus, a longer epoch could lead to a phenomenon that all the connections are occupied by transactions waiting for confirmation and the system may stop serving for a period. On the other hand, it also affects the latency since a submitted write request must wait for the epoch snapshot to be generated before responding to users. A long epoch will result in long latency, while a short epoch may result in too frequent conflicts merging and also degrades performance.



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

Fig. 8. Effect of epoch length.

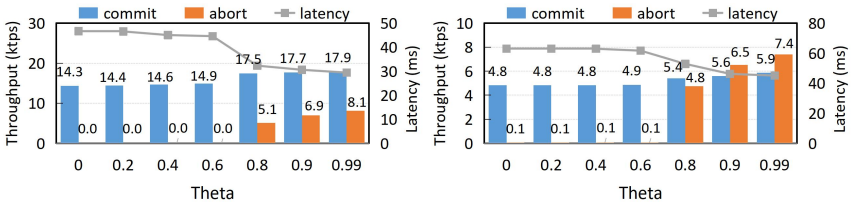
Figure 8 shows the throughput and latency results of YCSB-MC and TPC-C workloads with different epoch lengths ranging from 1 ms to 200 ms. The throughput results are under expectation as analyzed above. The latency is determined on one hand by the single-trip delay when the epoch length is short; on the other hand by the epoch length setting when the epoch length is long.



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

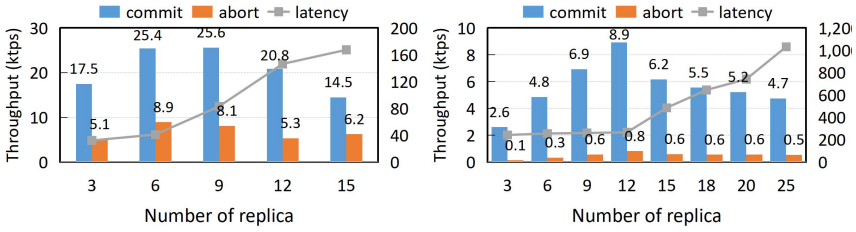
Fig. 9. Performance with different isolation levels.

7.5.2 Isolation. We study the performance when adopting different isolation levels in this experiment. Figure 9 shows the results under YCSB-MC and TPC-C workloads. There is not too much difference in 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.



(a) YCSB-MC (b) YCSB-HC

Fig. 10. Performance with different contention levels.



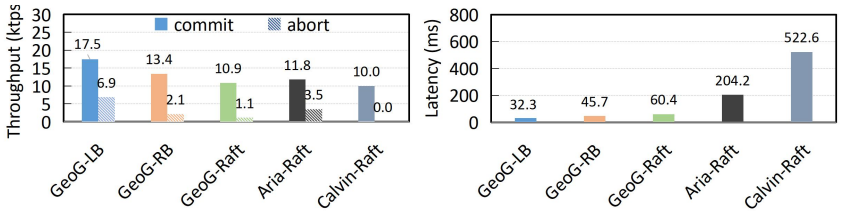
(a) YCSB-MC (China) (b) YCSB-MC (WorldWide)

Fig. 11. Scalability (YCSB-MC).

7.5.3 Contention Levels. To study the performance under different contention levels, we vary the θ parameter (the skew factor) from 0 to 0.99 in YCSB-MC and YCSB-HC workloads. As shown in Figure 10, the abort rate is higher when θ is bigger due to a higher probability of conflict, and the abort rate under the write-intensive workload YCSB-HC is even higher than in YCSB-MC. It is interesting that the throughput is a bit higher and the latency is a bit lower when θ is bigger. This is because that the aborted transactions resulted from conflicts will release the threads early without blocking (Line 24 in Algorithm 1), which allows serving more new transactions (that do not conflict with others) to commit. If the new transaction is read-only, it will be directly returned, which helps reduce the latency and improve the throughput.

7.6 Scalability

In this experiment, we study the scalability of `GeoGauss`. We scale the number of replica nodes from 3 to 15 and run YCSB-MC. As shown in Figure 11a, the throughput is steadily increased when the number of replica nodes is no more than 9, then the throughput decreases. This is because more replicas will result in more replication overhead, which degrades the performance. the latency increases when using more replicas. This is because the epoch-based merge requires to receive updates from all replicas, which may lead to longer synchronization time. To investigate the scaling performance in a cross-continental setting. We set up a worldwide cluster that spreads across continents, including 25 nodes deployed in five data centers (London, Singapore, Tokyo, Silicon Valley, and Virginia). We scale the number of replica nodes from 3 to 25 and run YCSB-MC. The throughput and latency results are shown in Figure 11b. We can see a similar trend with the scalability experiment in China. The peak throughput in the worldwide cluster is lower than in China, and the latency is much longer, which is under expectation.



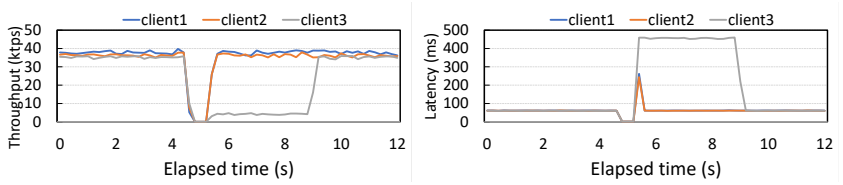
(a) throughput (b) latency

Fig. 12. Performance with fault tolerance (YCSB-MC).

7.7 Fault Tolerance

In the above experiments, the fault tolerance supports of deterministic databases, *e.g.*, `Calvin` and `Aria`, are turned off for optimal performance, and `GeoGauss` only uses local write set backup server which does not affect performance. In this experiment, we turn on Raft replication in

Calvin, Aria, and GeoGauss, and compare the performance of these systems under fault tolerance support (*i.e.,* Calvin-Raft, Aria-Raft, GeoG-Raft). We also measure the performance of GeoGauss with the other two weaker fault tolerance mechanisms, *i.e.,* local and remote write set backup server (GeoG-LB and GeoG-RB). As shown in 12, GeoG-Raft shows comparable throughput with Calvin-Raft and Aria-Raft. When adopting weaker fault tolerance mechanisms, *e.g.,* GeoG-LB and GeoG-RB, the throughput can be greatly increased. Regarding the latency results, GeoG-Raft shows much lower latency than Calvin-Raft and Aria-Raft owing to our optimistic execution scheme. As discussed in Section 5.2, GeoG-LB, GeoG-RB, and GeoG-Raft require ~0.5 RTT, ~1 RTT, and ~1.5 RTT, respectively, so GeoG-LB shows lower latency than GeoG-RB and GeoG-Raft.



(a) client-throughput (b) client-latency

Fig. 13. Performance under failures (YCSB-MC).

To investigate the performance fluctuation under failures, we manually shut down a node and see how GeoGauss acts after failure. Figure 13 shows the changes in throughput and latency from each client's perspective, where we place a client that connects to the server node in each region. The temporary performance degradation during a 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 500 ms timeout setup). The requests of client3 that were previously connected to the crashed node are routed to the nodes that are still providing services in other regions. But the transaction requests are executed in remote regions, resulting in decreased throughput and increased latency on client3. After the crashed node resumes, the Raft-based membership management will notice and let client3 connect to the node in the same region.

8 CONCLUSION

This paper presents GeoGauss, a strongly consistent and light-coordinated geo-distributed transactional database with multi-master replication architecture. We have shown that the performance of geo-distributed transaction processing can be greatly improved by GeoGauss with strong consistency and weak isolation. By employing epoch-based output replication and optimistic asynchronous execution, 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. GeoGauss also overcomes the disadvantage of deterministic databases and supports interactive SQL execution, which has wider applicability.

ACKNOWLEDGMENTS

The work is supported by the National Natural Science Foundation of China (62072082, U2241212, 62202088, 62232009, 61925205, 62137001, 62272093), the National Social Science Foundation of China (21&ZD124), CCF-Huawei Populus euphratica Innovation Research Funding (CCF-HuaweiDB2022003), the Fundamental Research Funds for the Central Universities (N2216012, N2216015), Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist).

REFERENCES

[1] 2022. Apache CouchDB. http://couchdb.apache.org/
[2] 2022. Apache HBase. https://hbase.apache.org/
[3] 2022. ArangoDB. https://www.arangodb.com/
[4] 2022. Aria: A Fast and Practical Deterministic OLTP Database. https://github.com/luyi0619/aria
[5] 2022. Baidu braft. https://github.com/baidu/braft
[6] 2022. CalvinFS. http://https://github.com/kunrenyale/CalvinFS
[7] 2022. Cloudant. https://www.ibm.com/hk-en/cloud/cloudant
[8] 2022. ExtremeDB: Cluster Distributed Database System. https://www.mcobject.com/cluster/
[9] 2022. FaunaDB. https://fauna.com/
[10] 2022. Galera Cluster for MySQL. https://galeracluster.com/
[11] 2022. GNU Gzip. https://www.gnu.org/software/gzip/
[12] 2022. gRPC: A high performance, open source universal RPC framework. https://grpc.io/
[13] 2022. MySQL Tungsten. https://www.continuent.com/products/tungsten-replicator
[14] 2022. MySQL's primary-secondary replication. https://dev.mysql.com/
[15] 2022. openGauss. https://opengauss.org/
[16] 2022. PostgreSQL BDR. https://wiki.postgresql.org/wiki/BDR_Project
[17] 2022. Protocol Buffers. https://developers.google.com/protocol-buffers
[18] 2022. Redis CRDT. https://redis.com/blog/diving-into-crdts/
[19] 2022. Riak: Enterprise NoSQL Database. https://riak.com/
[20] 2022. Semi-synchronous replication at facebook. http://yoshinorimatsunobu.blogspot.com/
[21] 2022. TPC-C Homepage. https://www.tpc.org/tpcc/
[22] 2022. YugabyteDB: Distributed SQL Database. https://www.yugabyte.com/
[23] 2022. ZeroMQ: An open-source universal messaging library. https://zeromq.org/
[24] Daniel J Abadi and Jose M Faleiro. 2018. An overview of deterministic database systems. *Commun. ACM* 61, 9 (2018), 78–88.
[25] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. DynaMast: Adaptive dynamic mastering for replicated systems. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1381–1392.
[26] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: automatic physical design metamorphosis for distributed database systems. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3573–3587.
[27] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2015. Efficient state-based crdts by delta-mutation. In *International Conference on Networked Systems*. Springer, 62–76.
[28] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *J. Parallel and Distrib. Comput.* 111 (2018), 162–173.
[29] 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.
[30] 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.
[31] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. Caper: a cross-application permissioned blockchain. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1385–1398.
[32] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 International Conference on Management of Data*. 76–88.
[33] 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.
[34] 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.
[35] 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.
[36] Peter David Bailis. 2015. *Coordination avoidance in distributed databases*. University of California, Berkeley.
[37] 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.
[38] Prima Chairunnanda, Khuzaima Daudjee, and M Tamer Özsu. 2014. ConfluxDB: Multi-master replication for partitioned snapshot isolation databases. *Proceedings of the VLDB Endowment* 7, 11 (2014), 947–958.
[39] 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.

[40] 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.

[41] 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.

[42] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*. 123–140.

[43] 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.

[44] 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.

[45] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[46] Ant group. 2022. OceanBase. https://open.oceanbase.com/

[47] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. Resilientdb: Global scale resilient blockchain fabric. *arXiv preprint arXiv:2002.00160* (2020).

[48] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment* 10, 5 (2017), 553–564.

[49] Jelle Hellings and Mohammad Sadoghi. 2021. Byshard: Sharding in a byzantine environment. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2230–2243.

[50] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[51] 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.

[52] 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.

[53] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *Proc. VLDB Endow.* 12, 11 (2019), 1316–1329.

[54] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.

[55] Pincap. 2022. TiDB. https://pingcap.com/products/tidb

[56] Nuno Preguiça. 2018. Conflict-free replicated data types: An overview. *arXiv preprint arXiv:1806.10254* (2018).

[57] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication. In *Proceedings of the 23rd International Conference on Extending Database Technology (EDBT)*. 73–84.

[58] Thamir M Qadah and Mohammad Sadoghi. 2018. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*. 13–25.

[59] 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.

[60] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. 2021. RingBFT: Resilient Consensus over Sharded Ring Topology. *arXiv preprint arXiv:2107.13047* (2021).

[61] 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.

[62] 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.

[63] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of convergent and commutative replicated data types.* Ph. D. Dissertation. Inria–Centre Paris-Rocquencourt; INRIA.

[64] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. 386–400.

[65] 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.

[66] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. 2022. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 17–33.

[67] 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.

[68] 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.

[69] Alexander Thomson and Daniel J Abadi. 2015. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)*. 1–14.

[70] 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.

[71] 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.

[72] Chenggang Wu, Jose M Faleiro, Yihan Lin, and Joseph M Hellerstein. 2019. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering* 33, 2 (2019), 344–358.

[73] 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.

[74] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. 2016. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Transactions on Knowledge and Data Engineering* 28, 10 (2016), 2635–2650.

Received July 2022; revised October 2022; accepted November 2022