arXiv:2304.09692v1 [cs.DB] 19 Apr 2023

GeoGauss：面向地理复制 SQL 数据库的强一致轻协调联机事务处理

WEIXING ZHOU 和 QI PENG，中国东北大学ZIJIE ZHANG，华为技术有限公司，中国YANFENG ZHANG*，中国东北大学YANG REN 和 SIHAO LI，华为技术有限公司，中国GUO FU 和 YULONG CUI，中国东北大学QIANG LI，华为技术有限公司，中国CAIYI WU、SHANGJUN HAN 和 SHENGYI WANG，中国东北大学GUOLIANG LI，中国清华大学GE YU，中国东北大学

跨国企业开展全球业务，对地理分布式事务数据库有着旺盛需求。现有最先进的数据库采用分片主从复制架构。然而，单主服务模式会导致客户端产生大量跨区域写入，而分片架构则需要多次往返确认（例如，两阶段提交）来确保跨分片事务的原子性。这些局限促使我们寻求另一种设计方案。在本文中，我们提出了采用全副本多主架构的强一致联机事务处理数据库 GeoGauss。为了高效合并来自不同主节点的更新，我们提出了一种多主乐观并发控制机制，将数据复制与并发事务处理统一起来。通过利用基于纪元的增量状态合并规则和乐观异步执行，GeoGauss 借助轻协调协议确保强一致性，并在弱隔离下实现更高的并发度，从而满足我们的需求。地理分布式实验结果表明，在 TPC-C 基准测试中，与最先进的地理分布式数据库 CockroachDB 相比，GeoGauss 的吞吐量提高了 7.06 倍，延迟降低了 17.41 倍。

CCS 概念：• **信息系统 → 关系型并行与分布式数据库管理系统**。

附加关键词与短语：地理分布；多主复制；副本一致性；事务处理；确定性数据库

ACM 参考格式：
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 和 Ge Yu。2023。GeoGauss：面向地理复制 SQL 数据库的强一致、轻协调联机事务处理。*ACM* 数据管理会议录 1，1，第 62 号文章（2023年5月），27 页。https://doi.org/10.1145/3588916

*Yanfeng Zhang 为通讯作者。

作者地址：Weixing Zhou、Qi Peng、Yanfeng Zhang、Guo Fu、Yulong Cui、Caiyi Wu、Shangjun Han、Shengyi Wang、Ge Yu，东北大学，中国辽宁省沈阳市浑南区创新路 195 号，邮编 110169，{zhouwx@stumail, ffpengqi@stumail, Zhangyf@mail, yuge@mail}.neu.edu.cn；Zijie Zhang、Yang Ren、Sihao Li、Qiang Li，华为技术有限公司，中国陕西省西安市，{zhangzijie9, renyang1, sean.lisihao, liqiang199}@huawei.com；Guoliang Li，清华大学，中国北京市海淀区双清路 30 号，liguoliang@tsinghua.edu.cn；

允许免费制作本作品的全部或部分数字或纸质副本，供个人或课堂使用，但不得为营利或商业利益制作或分发副本，且副本须附有本声明及首页的完整引文。必须尊重本作品中由作者以外的其他人拥有版权的组成部分。注明出处后可以摘录。若要以其他方式复制、再版、发布至服务器或向列表重新分发，必须事先获得特定许可和/或支付费用。请通过 permissions@acm.org 申请许可。© 2023 版权所有归所有者/作者所有。出版权已授权给 ACM。2836-6573/2023/5-ART62 $15.00 https://doi.org/10.1145/3588916

1 引言

全球性公司已在世界各地的许多国家建立了数据中心。为了支持其全球业务，我们希望开发一种地理分布式事务型 SQL 数据库，使其分布在多个地理位置各异的地点；例如，许多电信服务提供商已在地理分布式环境下部署了其 ICT 数据库。设计目标是实现高可用性、强一致性和高性能。

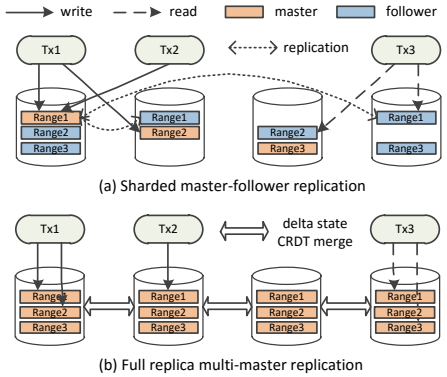


图 1. 分片主从复制与全副本多主复制对比。

高可用性通常通过冗余数据复制实现，即将相同的数据副本存储在多个地理区域中。数据复制不仅有助于实现高可用性，还能提供地理局部性和读取可扩展性，使数据副本靠近不同区域的用户，从而降低读取延迟，并进一步提升整体数据访问吞吐量。现有最先进的地理分布式事务数据库，例如，Google Spanner [41], F1 [59],CockroachDB [67], YugabyteDB [22], TiDB [55], SLOG [61] 和 ConfluxDB [38] 均采用如图 1a 所示的分片主从复制 架构。数据根据键范围划分为多个分片。每个分片分配给单个主节点，负责处理所有读写请求；同时，这些分片会被复制并放置到多个地理分布式从节点 上，仅负责处理读取请求。由于采用单主节点架构，写写冲突会集中到同一个工作节点中，从而易于处理。此外，分片可以分散写入请求，以提高写入吞吐量。

分片主从复制架构得到了广泛采用 [22, 41, 55, 61, 67], ，但存在两大缺点。1）单主服务模式要求将所有客户端的写入请求路由到单个主节点，这会导致跨区域写入，进而增加事务延迟。尽管可以通过地理感知分区和区域分片放置来缓解这一缺点 [67], ，但性能仍会受到影响，尤其是对于不具备局部性的应用。2）分片架构依赖两阶段提交（2PC）协议来确保原子性。这要求协调者与全球分布式工作节点之间进行多次往返确认，进一步影响性能。

数据复制的另一种选择是全副本多主架构，如图 1b 所示，其中每台服务器都维护一份完整的数据副本，且所有服务器节点均处理读取和写入请求。通过在每个区域放置一个完整副本，系统可以为用户提供本地写入和读取服务。借助完整副本，无需使用 2PC 来确保原子性。近年来，涌现出许多多主系统，例如，Aurora [71], Riak [19], Calvin [70], FaunaDB [9], Anna [73], Aria [51] 和 Q-store [57]。然而，要采用多主架构，还需要解决三个关键挑战。

挑战 1：跨节点写写冲突。与主从架构不同，在主从架构中，对同一数据的所有写入都会路由到同一个节点；而在多主复制中，对同一数据的多个副本进行并发更新可能导致跨节点写写冲突。尽管可以通过分布式两阶段锁定协议解决这一问题，但在地理分布式环境中，这种方案的开销过大。我们通过采用无冲突复制数据类型 (CRDT) [65, 73]来应对这一挑战。具体而言，我们采用了一种经过优化的基于状态的无冲突复制数据类型，即增量状态 *CRDT*[27, 28], ，它只传播最近应用于数据库状态的更新，而不是整个数据库状态。所有副本之间都会交换状态更新及其本地时间戳信息。在接收端，写写冲突会通过一个自动合并任意一对副本更新的函数进行处理，该函数必须满足结合性、交换性和幂等性（即 *ACI* 性质） [65]。任何节点上的数据库状态都会通过同一个合并函数合并状态更新而单调递增，因此，各副本可以在没有全局协调者的情况下独立更新。合并函数的 ACI 性质保证了在所有更新合并之后，副本最终能够实现一致性。然而，对于大多数事务应用而言，最终一致性是不可接受的。

挑战 2：强一致性。我们希望尽可能快速地向用户返回已提交/中止状态，但在所有副本达到一致之前，不允许这样做。线性一致性是最强的单对象一致性保证之一（即在实时约束下，每次操作后所有副本都达到相同状态），它需要昂贵的协调；而最终一致性（即不受时间约束，所有副本最终达到相同状态）可通过无冲突复制数据类型实现，无需协调即可提升性能，但可能导致各种异常。我们通过引入基于纪元的合并来应对这一挑战。这是一种折中方案（介于实时同步合并与异步合并之间），可在纪元粒度上保证一致性。与每个事务都有一个协调者的两阶段提交不同，我们只在纪元粒度上进行协调，因此可以将协调开销分摊到一组事务上。收集完所有节点在一个纪元内的更新后，每个节点都会按照具有 ACI 性质的同一函数合并相同的更新集合，然后将其应用于之前一致的本地副本状态。这样，所有节点便能按纪元达到全局一致快照。

挑战 3：性能。然而，采用基于纪元的合并会引入同步屏障，要求接收所有对等节点的更新（这是唯一的协调），从而严重影响性能。为解决这一限制，我们在多主环境下采用乐观执行。主节点基于当前数据库状态，乐观地执行其纪元i 的本地 SQL 请求，而该状态不一定对应最新的纪元快照（$i-1$）。随后，各节点交换生成的写集（即更新），以进行基于纪元的冲突合并。合并结果用于生成新的快照 i。只有成功通过验证的事务才允许提交。换言之，事务执行阶段不受时期概念限制，在各节点之间以异步方式进行；而验证阶段则必须基于最新快照，在各节点之间以同步方式进行。另一方面，考虑到主流的分片主从系统 [22, 41, 67]支持可串行化（将事务逻辑作为一个整体执行），乐观执行允许在弱隔离下使不同事务的读集和写集重叠，从而带来更高的并发性和性能。这充分满足了电信服务提供商的 ICT 数据库对强副本一致性和高吞吐量的要求，因为在大多数场景中，弱隔离已足够。

通过整合上述技术，我们提出了 多主乐观并发控制（*multi-master OCC*），这是一种适用于多主环境的基于纪元的乐观事务处理方案。该方案将数据复制与乐观并发控制相结合，在支持多个隔离级别的同时实现高并发，而又保持.....

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

在纪元粒度上保证强副本一致性。此外，我们通过修改 openGauss MOT [34], 这一针对多核处理器进行了高度优化的内存存储引擎，构建了多主地理复制 OLTP 数据库 GeoGauss。GeoGauss 支持具备增量状态 CRDT 合并功能的全功能 SQL 引擎。据我们所知，GeoGauss 是首个将 CRDT 技术集成到支持完整 SQL 功能的商业 SQL 数据库中的系统。

总而言之，我们做出了以下三项关键贡献。

• **基于纪元的多主 OCC**。我们利用增量状态 CRDT 和基于纪元的复制，提出了一种多主 OCC 协议，支持轻协调、高吞吐量且具有强一致性的事务，并支持多种隔离级别。

• **GeoGauss 数据库**。我们开发了地理复制 OLTP 数据库 GeoGauss，提供完整的 SQL 支持和丰富的优化（例如，高并发、高效通信、流水线化和容错），以适应地理分布式环境。

• **大量实验**。我们在地理分布式环境中，使用 YCSB 和 TPC-C 基准测试进行了大量实验。我们将 GeoGauss 与 CockroachDB (CRDB)[67], Calvin [70], Aria [51],CalvinFS [69],Q-Store [57], SLOG [61],以及无协调 KV 数据库 Anna [72]进行了比较。结果表明，在 TPC-C 基准测试中，GeoGauss 的吞吐量比这些竞品系统高出 1.11X–7.06X，延迟低 2.28X–17.41X；在中等竞争度 YCSB 基准测试中，吞吐量高出 0.44X–87.36X，延迟低 0.27X–30.94X。

2 背景

地理复制系统具有三大优势：高可用性、地理局部性和读取可扩展性。本节将从不同维度回顾现有的复制系统。

2.1 复制数据系统

表 1 总结了多个复制系统及其主要特性。接下来，我们将研究数据复制的关键技术。

2.1.1 复制架构。主要有三种复制架构：主从、多主和基于法定人数的无主架构。一些生产系统（例如，openGauss [15]）采用主从复制，主要用于冗余备份；另一些系统则通过允许从副本处理读取请求来采用主从复制，以扩展读取吞吐量（例如，CockroachDB [67], ConfluxDB [38]）。然而，所有用户的写入请求都必须路由到单个主服务器，因此这种架构不适用于

用户分布在不同区域的地理分布式数据库。采用无主（或无领导者）复制时，用户的写入/读取操作会发送到所有副本，并使用法定人数协议通过比较单调版本号来避免过时读取（例如，DynamoDB [43], Cassandra[50]）。然而，所有用户的写入请求都必须发送到多个地理远程服务器，这在地理分布式场景中代价高昂。采用多主复制 [1, 3, 7, 8, 10, 44, 53], 时，所有节点都可以为本地用户处理读取和写入请求，因此天然适用于地理分布式需求。

此外，区块链系统中也有研究在拜占庭容错（BFT）和崩溃容错（CFT）集群上支持分布式事务处理。例如，ResilientDB [47] 提出了一种适用于地理规模部署的分层多主架构。此外，还存在许多分片区块链，通过将节点分组成集群，并在多个独立运行的集群之间划分数据来实现可扩展性，例如，AHL [42], Caper[31], SharPer [32], RingBFT [60] 和 ByShard [49]。这些工作大多专注于优化跨分片事务的性能，而这也是分布式数据库管理系统中的一个瓶颈 [48]。另一种方法是动态转移数据的主控权，以避免代价高昂的多分片协调，从而提升性能，例如，DynaMast [25] 和 MorphoSys [26]。

2.1.2 副本的排序与一致性。采用主从架构时，部分数据库仅将复制用于备份（例如，MySQL [14], 、HBase [2], 和 openGauss [15]）。由于需求不同，这些系统采用了不同的排序技术，例如，openGauss 依赖集中式时间戳预言机（*TSO*）服务来实现快照隔离，而 HBase 则使用本地时间来标识值的版本。为了最大限度地提高性能，它们配置为采用异步复制或半同步复制 [20]（对部分从节点采用同步复制）。这意味着在某个时间点，可能无法保证所有副本的一致性（最终一致性）。另一类采用主从复制的分片数据库则能够提供可串行化和线性一致性。例如，谷歌 Spanner [41] 依赖*TrueTime*，CockroachDB [67] 依赖混合逻辑时钟（*HLC*），而 OceanBase [46] 和 TiDB [55] 则依赖集中式时间戳预言机（*TSO*）服务。通过基于领导者的协议（例如，Paxos 和 Raft）来保证副本一致性。近期提出的 ISS [66] 旨在通过高效复用共识实例，提高这些基于领导者的共识协议的可扩展性。

许多 NoSQL 键值数据库采用无主复制，没有稳定的主节点，并使用法定人数协议来确保一致性。例如，DynamoDB [43] 通过向量时钟检测更新冲突，而 Cassandra [50] 则使用本地时间来识别值的版本。然而，这种基于法定人数的方法由于没有稳定的主节点，会牺牲性能。因此，无主数据库要么面向能够容忍最终一致性的使用场景，要么以性能为代价实现强一致性，例如例如，DynamoDB [43], 、Riak [19], 和 Cassandra [50]。

对于多主架构，MySQL 和 PostgreSQL 都提供了跨站点多主复制工具，例如例如，MySQL Tungsten [13] 和 PostgreSQL BDR [16], ，但它们采用异步复制，无法保证强一致性。区块链系统是一种具有拜占庭容错能力的特殊多主复制系统。在不可信环境下，就事务顺序达成共识是确保一致性的关键。例如，无许可区块链比特币 [54] 采用工作量证明（PoW），许可型区块链 Fabric [33] 则借助排序服务确定事务的全局顺序。通过这种方式，这些区块链能够实现顺序一致性和可串行化隔离。

与其他方案不同，无需协调的复制（如例如，Berkeley Anna [72]和 Redis CRDT[18],）利用操作的数学性质（即结合性、交换性和幂等性），使不同副本将乱序事务作为输入时仍能实现副本一致性。典型工作包括逻辑单调性即一致性（*CALM*） [29, 30, 35, 36, 39] 和无冲突

复制数据类型（无冲突复制数据类型） [28, 56, 64]。确定性数据库采用了另一种一致性保证方案。多个主节点独立接受事务请求，并通过批次相互交换这些请求以实现复制。在每个副本上，收到的相同事务集合都会按确定的顺序分批执行。执行之前，所有节点都已就排序规则达成一致，例如，依据全局唯一 ID（本地时间 +服务器 ID）。本质上，该方案保证顺序一致性并支持可串行化隔离，例如，Calvin [70], Aria [51],CalvinFS [69] 和 Q-Store [57]。

2.1.3 事务处理中的复制。 传统的并发控制技术，例如多版本两阶段锁 （MV2PL）和多版本时间戳排序 （MVTO），可以直接应用于单主架构，因为所有并发写入都在本地处理。在分片数据库中，跨分片事务需要两阶段提交，以确保原子性和一致性。对于基于无主复制的键值存储，它们遵循最后写入者获胜规则来解决写写冲突。

对于多主复制，并发更新存在多种冲突解决方法。其中最特别的是 Anna [72] ，它天然不存在冲突，因为更新操作必须不受执行顺序影响，例如，集合并集和计数器。在比特币 [54], 中，最先解决工作量证明谜题的单个节点会串行执行事务，不存在并发。在 Fabric [33], 中，所有对等节点都使用多版本乐观并发控制（*MVOCC*）并行执行本地事务，然后根据共识顺序验证执行结果，并中止发生冲突的事务。确定性数据库按照预定义的串行顺序执行事务。由于操作系统以根本上非确定性的方式调度并发运行的线程，这类数据库的并发能力受到限制。现有的确定性数据库依靠依赖图（Aria [51]）或有序锁（ Calvin [70]），在保证确定性的同时提供更高的并行性。

2.2 需求

尽管已有许多分布式数据库被提出，但头部电信客户的需求促使我们提出一种新的联机事务处理数据库，以满足以下要求。

• **多主架构**。 如第 1 节所述，多主架构更适合地理分布式环境，能够提供低延迟服务。

• **功能完备的 SQL 引擎**。 广泛的 SQL 覆盖度和互操作性对于商用数据库至关重要，也能满足各类用户的需求，例如，联机事务处理、联机分析处理和存储过程。

• **强一致性**。 对于地理复制数据系统，我们期望副本具有强一致性。然而，线性一致性代价过高且没有必要。牺牲少许实时性可以换取性能，因为顺序一致性已足以满足大多数场景的需求，例如，在线零售和全球交易。

• **弱隔离下的高性能**。 电信服务提供商使用的许多 ICT 数据库都部署在地理复制环境中，例如运营支撑系统（OSS）、客户关系管理（CRM）和企业资源规划（ERP）。这些系统要求高吞吐量和低延迟，但不要求强隔离。支持较弱的隔离级别即可，无须达到可串行化。

现有的地理分布式事务数据库为实现线性一致性和可串行化付出了过高代价。
y, which cannot satisfy our requirements. Next, we will present GeoGauss.

3 SYSTEM OVERVIEW

本节介绍 GeoGauss 的系统概况。整体架构如图 2 所示。GeoGauss 基于关系型数据库 openGauss [15]开发。我们依靠其 SQL 引擎

解析 SQL 语句并生成物理执行计划，并使用其基于行、针对内存优化的存储引擎 openGauss MOT [34] 存储数据（其他支持并发的内存行存储也可以作为替代方案）。本地 SQL 请求基于本地当前数据库状态以乐观方式执行。节点之间只交换本地 SQL 执行后获得的写集（也称为更新），接收端再将其与本地更新合并。随后，这些合并后的更新会应用到每个本地副本。以下介绍一些关键特性。

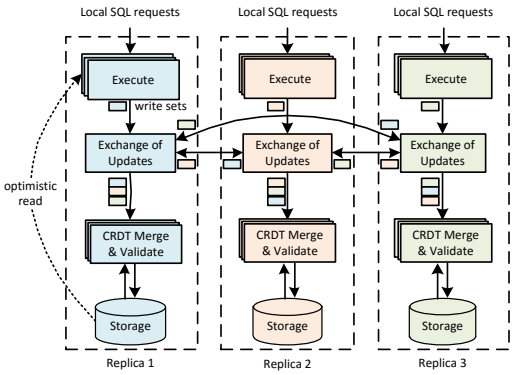


Fig. 2. Overall structure.

具备完整 SQL 引擎的多主架构。 GeoGauss 提供完整的 SQL 引擎，支持标准 SQL、应用程序编程接口、互操作性，等。 GeoGauss 部署在多个区域服务器上，每个服务器都接受本地用户的 SQL 请求，并通过解析器、优化器和执行引擎将高级 SQL 语句转换为低级读取和写入请求。每个区域服务器根据其本地副本独立处理本地 SQL 请求。

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.

基于纪元的更新复制。 所有主节点定期（每个纪元）交换各自的本地更新。纪元 是一个较短的时间段（例如，10 ms），纪元编号根据本地物理时间单调递增。与执行 SQL 语句复制的确定性数据库不同，GeoGauss 在主节点之间交换写集 ，这可以被视为基于状态的无冲突复制数据类型合并中的增量状态 。图 3 给出了一个示例，说明该机制在两个副本的设置下如何工作。假设两个副本在初始时刻具有一致的快照 S_0，副本 a 和副本 b 分别独立执行各自的本地事务，并独立生成各自的本地写集。对于已完成执行的事务，其写集（也称为 更新）会在每个纪元结束时在副本之间进行交换。当一个节点收到第一个纪元提交的所有远程更新后，这些更新将与本地写集合并，并根据无冲突复制数据类型合并规则解决写写冲突。该规则保证，对于相同的更新集合，无论到达顺序如何，都能得到相同的合并结果。在每个节点上，相同的合并结果都会应用于原始快照 S_0 ，以生成新的快照，即，$S_1 = S_0 \oplus \{W_{a1}(x)\ W_{b1}(x)\}$，其中 $\oplus$ 表示合并操作。只有当纪元 i 对应的本地快照生成后（经过合并和验证），本地事务才能与

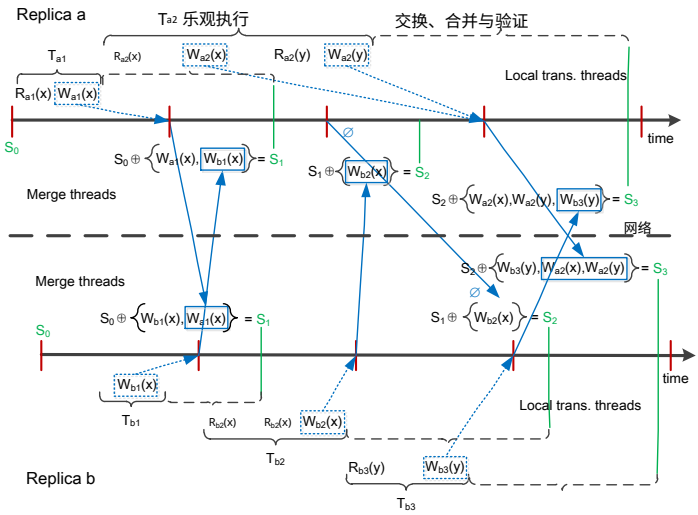


图3. 两个副本的更新合并示例。每个副本 a 都独立接受本地事务，例如，T_{a1} 和 T_{a2}。$R_{a1}(x)$ 和 $W_{a1}(x)$ 分别表示副本 a上数据项 x的读写操作。提交纪元编号为 cen 的本地事务，只有在生成快照 cen 后才能确认，因此存在一个用于接收远程更新、合并和验证的时间段。红色条表示由本地时钟确定的纪元边界，这些本地时钟在各副本之间不一定同步。绿色 S_i 是全局一致快照。蓝色虚线/实线框分别是发送/接收的写集。

提交纪元编号 i 由主机主节点随已提交/已中止通知返回，例如，T_{a2} 在纪元 3 提交的 S_3，只有在副本上生成 a后才会返回；此时副本 a 已接收并合并在纪元 3 提交的远程更新 W_{b3}（y）。

带事务处理的CRDT合并。只需交换写集（即更新），我们的多主乐观并发控制算法便可统一处理更新合并与并发事务处理，并按纪元执行。从每个副本的角度来看，一旦合并了纪元 i 及之前所有纪元的更新（这些更新收集自本地节点和所有其他远程节点），就能获得全局一致快照 i；此时，纪元 i 的本地事务即可返回已提交/已中止响应。我们依靠增量CRDT高效解决跨区域冲突，并实现全局一致状态。借助具有ACI性质的合并操作，来自不同副本的这些写入可以部分合并以提高效率（结合性），可以以不同顺序到达（交换性），甚至可以重新传输（幂等性）。合并本地/远程更新后的副本状态保证在所有节点之间一致。

异步执行与同步验证。允许节点执行纪元 i 的事务，即使快照（$i-1$）尚未生成。换言之，执行不一定需要跨纪元同步，但验证必须同步进行。纪元 i 的事务基于全局一致快照（$i-1$）进行验证，其中与其他事务存在写写冲突的已执行事务应当中止（通过检查某个事务的预写是否被其他事务覆盖）。按照传统方式，数据复制与事务处理采用分离式设计，失去了实现高并发的机会。我们通过结合更新合并与乐观事务处理，支持高吞吐量的事务处理，具体内容将在第 4 节讨论。

算法 1：本地事务处理（每个事务一个线程）

输入： 一个事务 T

输出： 返回提交或中止

```
1  {T.sen T.lsn} ← 获取当前纪元编号和最新快照编号;
2  基于最新快照执行事务 T;
3  T.RS ← T's read set; T.WS ← T's write set;
4  {T.csn T.cen} ← 获取当前时间戳和纪元编号;
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;   //读取行已删除
13         否则，如果可重复读 且 record.csn ≠ row.csn 则
14             中止 T;   //读取行已更新
15         否则，如果快照隔离 且 row.cen − 1 > T.lsn 则
16             中止 T;   //snapshot is updated
17 else
18     //Not support
19 if T.WS == ∅ then
20     return COMMIT;   //只读事务
21 添加 T.{sen，csn，cen_WS} 到发送缓冲区;
22 等待快照 T.cen − 1 生成;
23 DeltaCRDTMerge(T.{sen,csn,cen,WS});   //算法2，基于一致性快照（T.cen − 1）
24 等待，直到远程/本地事务中带有 T.cen 的所有更新都应用到行头上;
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

事务从客户端提交到本地服务器，并为该事务分配一个线程。**算法 1** 描述了每个线程的事务处理逻辑。首先，为事务分配一个起始纪元编号 (sen) 和一个最新快照编号 (lsn)（第 1 行）。lsn 是服务器当前维护的最新全局一致快照编号，用于快照隔离中的读集验证。随后执行事务并生成读集 RS 和写集 WS（第 2-3 行）。事务执行期间会检查 SQL 约束。如果发生约束违反，事务将在生成写入前中止。

集合。对于只读事务，它会读取最新快照中的数据，并在读取验证后返回（第 19–20 行）。

对于包含写操作的事务，我们的多主乐观并发控制要求每个事务记录一些元信息，以便进行无冲突复制数据类型合并。具体而言，系统会为事务分配一个提交纪元编号 （cen）和一个提交序列号 （csn）（第 4 行）。cen 是当前的物理纪元编号，用于确定尝试一起提交的事务批次，其中包括具有相同 cen 的本地事务和远程事务。时间戳与其本地服务器 ID 一同用于生成全局唯一的 csn，该编号用于确定同一纪元内的执行顺序。此外，事务的写集 WS及其 {sen,csn,cen} 会发送给远程对等节点。GeoGauss 支持多个 ANSI 隔离级别，例如读已提交 （RC）、可重复读 （RR）和快照隔离 （SI）。它们在处理逻辑上有所不同（第 5–18 行）。

对于具有 T.cen 的事务，会设置一个同步点，以等待快照（T.cen -1）生成（第22行）。快照（T.cen -1）通过合并纪元（T.cen -1）的所有本地和远程事务生成。这确保操作应用于最新的一致性快照，否则可能会影响正确性和一致性（第4.4节）。

随后启动 DeltaCRDTMerge 函数（第 23 行），该函数定义了将新更新（即增量状态）合并到当前数据库状态中的规则（参见第 4.2 节和算法 2）。行头部存储每一行的元信息 {sen,lsn,csn,cen}，用于指示任意事务线程对该行的更新历史，并供乐观并发控制的预写使用。例如，假设某行头部首先由事务 T 更新，也就是说，由 row.csn $= T$.csn 更新该行头部，随后被其他线程覆盖，导致 row.csn $\neq T$.csn，那么 T 将在验证阶段中止（第 26–29 行）。这意味着同一行上存在写写冲突。根据合并规则，只有一个写入会获胜并提交，其余写入都会中止。

只有在收集并应用纪元T.cen 的所有本地/远程事务后，事务 T 的验证阶段才能开始（第24行）。这对于确保没有遗漏任何更新、保证正确性至关重要。如果该事务未发生写-写冲突，或在冲突合并中胜出，则允许提交。该事务的写入数据用于更新涉及的行（第 31–33行）。在验证并应用具有相同提交纪元编号 cen 的所有事务后，将生成纪元 cen 的新快照。

4.2 更新合并

4.2.1 感知纪元的增量 *CRDT* 合并。 CRDT [63, 65] 是分布式数据类型，允许分布式 CRDT 实例的不同副本出现分歧，同时确保最终所有副本收敛到相同状态。基于状态的 CRDT 通过在副本间传播整个状态，以传播本地状态的更新。随后，接收到的状态会与远程状态合并，从而实现收敛（即所有副本上的状态一致）。在增量 CRDT [28], 中，传播的只有更新，而非整个状态。收到的增量状态（即 更新）通过合并函数与本地完整状态（即 数据库）合并，该函数可确定性地协调两种状态。增量状态可以通过不可靠的传播层发送，该层可能丢弃、重排或重复消息，即 增量状态可能始终乱序、被重新传输并再次合并。此外，为了实现高性能，还需要处理并发更新。因此，关键在于设计合并函数：它必须具备结合性、交换性、幂等性且适于并行处理，最重要的是能够感知纪元信息。

我们的合并函数 DeltaCRDTMerge 如**算法 2**所示。它定义了冲突合并规则，用于确定成功的更新，并且可由多个本地事务和远程事务并发调用。它依赖一种数据结构行头来支持基于纪元的增量状态合并，该结构存储每个已更新行的 {sen,lsn, csn, cen}。为便于说明，

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};    //先写入者胜
11             else
12                 中止 T;
13         否则如果 row.sen < T.sen 则
14             row.{sen, csn, cen} = T.{sen, csn, cen};    //较短事务优先
15         else
16             中止 T;
17     else
18         //row.cen > T.cen 永远不会发生
```

我们重点关注更新事务。1）如果写入行为空，表示它在过去几个纪元中已被其他线程删除，我们就中止该事务（第3-4行）。2）如果写入行尚未在当前纪元（T.cen）中预写入，即，row.cen $< T$.cen，我们就更新其行头，以便候选提交（第5-6行）。否则，该行已在当前纪元中被其他线程更新，即，row.cen $== T$.cen。请注意，由于在完成纪元 i 的所有更新合并（算法 1 第22行的同步点）之前，绝不会调用 DeltaCRDTMerge 来合并纪元（$i + 1$）的更新，因此 row.cen $> T$.cen永远不会发生。3）我们通过比较事务的 T.sen，让较短的事务优先，即，row.sen $< T$.sen。T.sen较大的事务意味着它更接近当前纪元 T.cen（因为 T.cen $- T$.sen较小），因此更有可能提交（第13-14行）。4）假设sen相同，我们使用csn确定顺序，并遵循先写优先规则（第8-12行）。请注意，对于插入请求，FindRow 函数无法定位该行。此外，多个并发插入事务可能会向同一行插入数据。我们使用临时表 存储插入的行，以处理同一纪元内的插入冲突。我们不调用依赖表索引的 FindRow，而是使用临时表存储插入的行。

算法 2 中定义的 DeltaCRDTMerge 操作类似于乐观并发控制中的先写优先冲突解决机制，但二者的设计目标不同。我们具有 ACI 性质的合并函数旨在合并不同节点上的本地/远程更新，以实现全局一致状态（即复制状态），即使这些更新以不同顺序到达各个节点（交换性）、被部分合并（结合性）或被重新传输（幂等性）。而乐观并发控制中的先写优先规则，则用于在单个副本上进行并发事务处理时，决定事务是已提交还是中止。

4.2.2 处理远程更新。远程更新的合并与本地事务处理并发运行并相互交互。在每个主节点上，多个接收线程和合并线程如**算法 3**所示持续运行。接收线程

Algorithm 3: Epoch-based Multi-Master OCC

输入： 持续接收的远程更新批次 $TS = \{T.\{\text{sen}, \text{csn}, \text{cen}^W \quad S\}\}$, 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 $TS \leftarrow$ 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 $\leftarrow$ get latest consistent snapshot no.;
8 $TS = BUF[\text{lsn}+1]$.get(); //blocking get
9 **foreach** T *in* TS **do**
10 DeltaCRDTMerge(T.{sen, csn, cen WS});
11 **if** T *is not aborted* **then**
12 $Q[\text{lsn}+1]$.push(T); //add T to commit queue Q
13 $N[\text{lsn}+1] \leftarrow N[\text{lsn}+1] + 1$; //统计已执行的远程操作 TS
14 **if** $N[\text{lsn}+1] == n - 1$ **then**
15 //all remote updates have been processed
16 通知所有({$Thread[T] \mid T.\text{cen} = \text{lsn} + 1$}); //notify all threads blocked at Line 24 (Alg. 1)
17 **foreach** T *in* $Q[\text{lsn}+1]$ **do**
18 Execute Line 26-33 of Alg. 1 for T; //validate and write
19 等待所有本地线程（$Thread.T.cen == \text{lsn} + 1$）完成;
20 通知所有({$Thread[T] \mid T.\text{cen} = \text{lsn} + 2$}); //notify all threads blocked at Line 22 (Alg. 1)
21 lsn $\leftarrow$ 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

持续接收来自远程对等节点的更新 TS，并将其与提交纪元编号 $T.$（cen）信息一起暂存于接收缓冲区中（第 3-4 行）。假设 lsn 是最新全局一致快照的编号（第 7 行）。合并线程基于最新的表状态（即 快照 lsn）合并纪元为（lsn + 1）的更新，并将逐个生成一致快照（逐个）。具体而言，如果缓冲区中存在纪元为（lsn + 1）的远程更新，则调用 DeltaCRDTMerge 函数处理这些更新（第 8-10 行）。未中止的事务会被推入提交队列 Q[lsn +1]（第 11-12 行）。如果所有远程对等节点的纪元为（lsn +1）的更新都已应用到行头上，它就会通知所有纪元为（lsn +1）的本地事务线程，并触发对提交队列中每项更新的验证和写回（第 16-18 行）。当纪元为（lsn + 1）的所有本地线程均已完成时，表示已生成新的全局一致快照。合并线程会立即通知所有正在等待该最新快照的下一纪元本地事务线程（第 19-21 行）。我们还设有一个主线程，用于为每个新的本地事务分配一个新线程（第 23-25 行）。

4.2.3 案例分析。**1）长时间运行的事务**。只有当事务在纪元i 提交时，我们才能在纪元 i结束时发送其写集。例如，在图 3 中， T_{a2} 是一个跨越三个纪元、并在纪元 3 结束的长时间运行事务。如果该事务在

读取验证阶段未被中止，其写集 $\{W_{a2}(x)\ W_{a2}(y)\}$ 会在纪元 3 结束时一并发送出去。请注意，在同一纪元开始的事务不一定要在下一个纪元开始之前同步完成。它们可能在不同纪元完成执行并提交，因此短事务无需等待长事务，可以在较早的纪元提交。长事务（跨越多个纪元）更可能因可重复读中的读取不一致或快照隔离中的过时读取而中止，因为读取的数据可能已被较早提交的短事务修改。不过，如果某个纪元没有更新，例如副本 a上的纪元 2，则会发送一条空消息，以避免远端节点无限等待。**2）网络延迟**。例如，在图 3 中，由于网络延迟，两个副本可能无法恰好在同一时刻获得一致的快照，但最终会达到一致。这不会阻碍系统运行，也不会影响顺序一致性（见第 4.4 节）。例如，快照 S_2 在副本 a的纪元 3 生成，而在副本 b的纪元 4 生成。T_{b3} 的写集 $W_{b3}(y)$会在纪元 3 结束时照常发送到副本 a。然而，纪元 i的更新合并要等到快照 $(i-1)$ 生成后才能开始（算法 1 的第22行）。例如，纪元 3 的本地更新 $W_{b3}(y)$无法与快照 S_1合并，只能与快照 S_2 以及纪元 3 的更新（$W_{a2}(x)$ 和 $W_{a2}(y)$）合并，从而生成 S_3。

4.3 隔离

弱隔离模型无法保证可串行化，但应用开发者通常认为，其对并发性的益处足以抵消使用它们可能引发的一致性异常所带来的代价。如**算法 1**所示，GeoGauss 支持多种 ANSI 弱隔离级别，允许用户针对具体应用选择最优的隔离级别。

读已提交（RC）。对于纪元 T.cen 的事务 T，在我们的系统中，只能以快照形式访问已提交的数据，因此支持 RC 隔离非常直接。本地事务均不会中止，并且都会用于生成写集。其写集 $T.WS$及其元数据会立即添加到发送缓冲区（第 21 行）。需要注意的是，它无法保证读取最新的已提交数据，因为最新的快照 T.cen -1 可能尚未生成。但它可以保证后一次读取的数据项更加及时。

可重复读（RR）和快照隔离（SI）。作为单版本内存数据库，我们主要通过乐观方式实现 RR 和 SI，即读集验证。1）如果先前读取的记录已被删除，在这两种隔离级别下，事务都会中止（第11-12行）。2）在事务执行期间，每次读取记录时，都会将该行的 csn 更新为该记录的 csn。如果其他线程在首次读取该行后更新了它（即 row.csn $\neq record$.csn），则在 RR 隔离级别下，事务会中止（第13-14行）。3）如果自事务开始执行（快照 T.lsn）以来，其读取行在新快照中已更新（快照 row.cen -1，这是由第22行的同步点导致的），则初始读取快照会更新（即 row.cen $-1 > T$.lsn），这违反了 SI。因此，在 SI 下事务会中止（第15-16行）。只有通过读集验证的事务才会被发送出去（第21行）。例如，在图3中，T_{b2} 在 RR 下会中止，因为它首次读取的 $R_{b2}(x)$ 是 x中的，而第二次读取的 $R_{b2}(x)$ 是更新后的 x中的。在 SI 下，如果事务的读取快照已过期，事务就会中止。例如，T_{a2} 在 SI 下会中止，因为它首次读取 $R_{a2}(x)$ 时基于 S_0，而第二次读取 $R_{a2}(y)$ 时基于 S_1。

可串行化快照隔离（SSI）。我们讨论支持 SSI [37] 的可能性。我们需要交换每条读取记录的键，以检测读写依赖关系。然后，当发现一对连续的冲突边时，我们可以通过中止事务来防止快照隔离异常。目前，GeoGauss 不支持 SSI，因为传输读取键的成本很高。

4.4 副本一致性

我们展示了 GeoGauss 如何保证副本一致性。

引理 1。 根据基于纪元的多主 *OCC*，读操作不会影响副本的一致性。

证明。在我们的系统中，每个主节点都会在快照上执行本地读取，然后生成写集。也就是说，写集由单个源工作节点生成，然后发送出去进行写写冲突合并。无论事务读取哪个快照（旧快照或新快照），事务写入的数据在发送出去之前都是确定性的 。读操作的一致性只会影响隔离性。读取不同版本数据的事务违反隔离约束（例如， RR 和 SI），并且会在产生更新之前中止。只有更新才能改变副本的状态。因此，读操作不会影响副本的一致性。□

引理 2。 假设存在初始数据库状态 S_i和一组具有相同 $\{\mathcal{U}(T_1)$、$\mathcal{U}(T_2)$.....、$\mathcal{U}(T_n)\}$的更新 TS =，其中 $\mathcal{U}(T_i)$ = T_i.{sen、csn、cen WS}。无论以何种顺序输入这些更新，算法 *2* 都将生成确定性的状态 S_{i+1}。

证明。合并规则（算法 2）通过比较元组 {sen、csn、cen} 来定义。我们在更新集合 TS上定义严格全序“$\prec$”，如下所示。对于具有相同 cen（即， T_i.cen = T_j.cen）的任意两个更新 U (T_i) 和 U (T_j)，如果满足以下条件之一，则有 U (T_i) $\prec$U (T_j)。

- T_i.sen > T_j.sen;
- T_i.sen = T_j.sen 和 T_i.csn < T_j.csn。

由于 T.csn 是根据 T 的源工作节点 ID 以及由源工作节点分配的提交时间戳生成的，因此它是全局唯一的，任意两个更新之间必然存在严格顺序。对于对同一行 x 的冲突更新，算法 2 允许具有“最小”（根据“$\prec$”）T 的事务的更新提交。也就是说，成功更新的集合始终等于 $\{\{x \in \text{WS} : T_i \in \text{C}(x)\} \mid \text{U}(T_i) \prec \text{U}(T_j)\ \forall T_j \in \text{C}(x)\}$，其中 x 是一行表，WS = {$T_1.WS \cup T_2.WS \cup \ldots \cup T_n.WS$} 是一个纪元中所有事务的写集的并集，而 C($x$) 表示写入行 x 的事务集合。因此，给定初始状态 S_i，无论这些更新以何种顺序输入（即使存在重复更新），在具有唯一元组 {sen,csn,cen} 的更新集合上隐式定义的确定性顺序都将保证确定性输出 S_{i+1}。□

定理 3。 基于纪元的多主 *OCC*（算法 *1*、*2*、*3*）以纪元为粒度确保副本一致性。

证明。引理 1 指出，读操作不会影响副本一致性，因此我们只需研究写操作的影响。引理 2 指出，给定一个更新批次和一个初始状态，无论这些更新的输入顺序是否确定，其输出状态都是确定的 。要实现副本一致性，关键在于确保所有副本都具有 1）相同的初始状态，以及 2）相同的更新批次。首先，算法 1 第24行和算法 3 第19行的同步点确保每个副本执行相同的 更新批次，即合并一个纪元中的所有本地/远程更新。其次，算法 1 第22行的同步点和算法 3 第7行的最新快照编号验证确保纪元 (i + 1) 的更新应用于快照 i，从而保证相同的 初始快照。因此，副本一致性以纪元为粒度得到保证。□

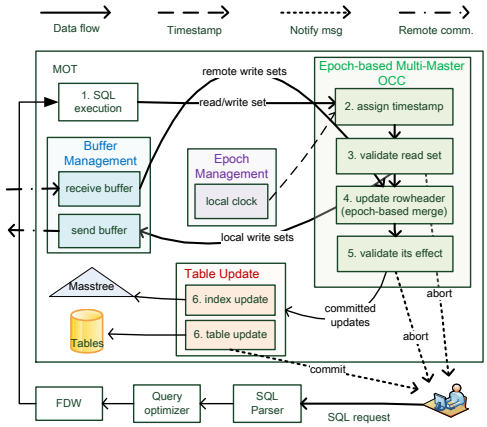


Fig. 4. Data flow of GeoGauss.

本地副本上的读操作不会在副本之间交换。每个副本上一个纪元内写操作的顺序都是确定且一致的（引理 2）。所有副本上的纪元快照依次生成。因此，我们的算法所确保的一致性属于顺序一致性。

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

线程阻塞与通知。大量事务线程可能会被阻塞，等待通知信号（算法 1 的第22行）。线程进入阻塞阶段与通知该线程的信号可能会并发发生。如果线程错过通知信号，就会永久挂起。为避免异常，我们让每个线程休眠，并定期主动检查条件；在线程数量较少时，这种方式效果良好。另一方面，我们引入互斥锁，以防止并发异常。但是，共享同一互斥锁的这些线程必须逐个串行调用。为了最大限度地提高并行性，我们创建多个互斥锁，并使每个互斥锁仅由互不相交的线程子集共享。这样可以有效缓解线程通知的突发情况。

消息队列与流水线化。我们采用 Protocol Buffers [17] 和 Gzip [11] 来压缩缓冲的写集。不同于 openGauss 使用的重量级 gRPC [12]，我们更倾向于使用轻量级 ZeroMQ [23]（采用发布-订阅模式），以实现节点间的高效数据传输。此外，如图 3 所示，每个纪元结束时，发送缓冲区中的写集会被打包并一并发送。这会占用计算资源并导致网络突发。为使通信与计算重叠，我们引入了一种流水线化技术，以流畅的方式分小批次发送写集。请注意，在流畅通信过程中，会发送 EOF 消息以指示纪元的结束（依据物理时间），从而使接收方能够确保一个纪元内的事务完整无缺。此外，我们通过零拷贝发送与接收进一步改进了流水线化。通常，当用户空间进程需要执行从网络设备读取数据或向其写入数据等系统操作时，必须进行一次或多次系统调用，再由操作系统在内核空间执行这些调用。我们通过零拷贝传输写集数据，并通过 Protocol Buffers 直接将其发送出去。这可以降低序列化与反序列化的开销。

5.2 容错

在多主架构下，如果某个主节点无法提供服务，其本地事务请求可以路由到另一个主节点。我们采用 Raft 协议 [5]，以对存活节点的状态达成共识，从而避免永久阻塞（整个系统可能会因等待故障节点的写集而陷入阻塞）。由于只有在存活节点的状态发生变化时才会调用该协议，因此开销较低。在这种纪元一致的复制系统中，多个主节点作为强一致副本运行，因此所有主节点同时发生故障的风险很小。我们也可以像现有商业数据库那样部署更多副本作为备份，以提高稳健性。然而，这种架构还有一个需要解决的独特问题，即必须防止更新丢失问题，也就是说，某些更新已在本地节点成功提交，却未能在远程节点提交。为避免此类故障，GeoGauss 提供了三种从轻量级到重量级的容错选项。

本地写集备用服务器。 我们在每个区域的每个副本旁部署一个（或多个）写集备用服务器，用于缓存生成的本地写集以进行备份。每当服务器节点向远程节点发送本地写集时，也会向本地写集备用服务器发送一份副本，后者将返回确认消息作为响应。只有在收集完所有提交纪元编号为 cen 的远程写集，并备份好本地写集 cen 后，我们才能向用户返回 cen 对应事务的已提交/已中止状态。一旦检测到节点故障并将其移除，其他远程节点会要求相应的写集备用服务器通过比较单调递增的 cen 来检查是否存在更新丢失。如果存在，远程节点就需要拉取丢失的写集，并将其合并，以在继续执行之前推进到一致快照 cen。由于本地备份与发送本地更新同时进行，且本地网络比跨区域网络快得多，因此本地备份的成本可以被隐藏，不会影响性能。

远程写集备份服务器。 如果某个区域发生故障，本地备份服务器也会随之故障，因此上述方案可能无法奏效。同样，我们可以在其他区域部署一个（或多个）远程备份服务器来缓存写集。在这种情况下，发送备份更新并接收确认消息需要一次跨区域往返时间（RTT）。这比本地备份方法成本更高，后者只需 0.5 次跨区域 RTT 即可接收远程更新。该方案可在一个或多个区域发生故障时确保容错能力。

写集的 Raft 复制。 为最大限度地提高容错能力，可以按纪元通过 Raft 共识协议复制写集。这与依赖 Raft 复制 SQL 输入的确定性数据库类似。每个节点（作为 Raft 主节点）都会将生成的写集发送给其他接收方主节点（作为 Raft 从节点），后者会返回确认消息。如果主节点收集到超过半数的确认消息，就会向接收方发送提交请求，指示可以将写集应用到现有数据库状态。因此，接收方需要 ~1。5 个 RTT 才能从发送方节点接收写集。

6 DISCUSSION

基于事务 vs. 基于批次 vs. 基于纪元。传统观点倾向于逐个处理事务 [41, 67]。在高争用工作负载和地理分布式环境下，这种方式并非最优，因为每个独立事务都需要付出高昂的协调开销。我们的多主OCC算法也可以稍作修改，以支持基于事务的冲突合并。不过，这会增加确保副本原子性和一致性的复杂性。另一种方案是基于批次的处理（例如，确定性数据库 [51, 70]），它限制每个批次的更新数量，而不是采用固定时间段。基于批次的方案可能因事务之间的负载不均衡而导致性能不理想，尤其是】【。

对于长时间运行的事务，由于批次间存在屏障，因此会受到影响。我们的基于纪元的多主乐观并发控制根据本地时间和提交时间（即 cen）将事务划分为多个独立子集。长时间运行事务的更新将在后续纪元中合并，且不会阻塞合并过程。此外，基于纪元和基于批次的方法在容错方面成本较低，因为共识和持久性是在一组事务上建立的，而非单个事务。

纪元长度。直观而言，纪元长度应设置得长于往返时间（RTT）。但由于确定性执行具有无协调特性，因此没有这个必要。与依赖远程服务器确认远程数据有效性的基于协调的方法（例如 两阶段提交）不同，这种确认通常通过锁定等方式实现；我们的多主乐观并发控制只需验证一个纪元的更新是否完整（即 从接收方的角度看，是否已收集到所有对等节点的更新），在继续执行之前无需与远程对等节点进行协调（无论单程还是多程往返）。在我们的跨区域实验中，用于生成读写集合的最新快照通常会比当前物理时间落后**3-5 个纪元**（对应 30-50 毫秒的单程延迟）。尽管纪元长度可以不受网络往返时间的限制，但仍应受到我们的服务模型约束。我们的系统沿用 openGauss 的服务模型：为每个已提交的事务分配一个线程，直到事务提交或中止后才释放其资源。在此之前，这些服务线程可能会被阻塞。缩短纪元长度可以提高更新交换频率，从而缩短确认时间；这不仅能减少阻塞时间、提升吞吐量，还能降低延迟。然而，过于频繁的通信和更新合并会增加调度开销，抵消其带来的收益（见第 7.5.1 节）。

相较于确定性数据库的优势。 现有的确定性数据库 [9, 24, 51, 52, 57, 61,62, 68–70] 会将事务请求的 SQL 语句批次复制到多个主节点。在收集完同一批次中的所有事务后，每个副本都必须按照预定义的串行顺序执行这些事务。这一要求比可串行化执行所需的要求更严格（后者只要求事务按照某个 串行顺序执行），因为操作系统以本质上非确定性的方式调度线程 [24]。这可能需要借助锁机制来实现并发，同时确保串行执行顺序，但会导致较高的调度开销。已有多种方法被提出以降低调度开销，例如，Aria [51], QueCC [58], PWV [45], 和 LADS [74]。然而，确定性数据库存在一些缺点，包括：1）不支持交互式 SQL，因为必须在执行 SQL 语句之前先传播它们，这限制了其应用场景；2）为保证确定性而产生额外的调度开销；3）由于事务之间的负载不均衡而导致性能不理想，尤其是在存在长时间运行事务时（因为必须等前一批事务执行完毕后，才能开始新的一批事务）。与交换 SQL 语句的确定性数据库相比，传播副本生成的写集的 GeoGauss 具有明显优势。1）我们通过在事务提交时传播写集，支持完整的 SQL 引擎（例如，互操作性）。2）借助适用于写集的无冲突复制数据类型合并规则（算法 2），我们无需额外的调度开销即可确保并发处理的确定性。3）在 GeoGauss 中，事务以乐观方式执行，并按照其提交时间 组织成批次，以进行同步验证。在长时间运行事务执行期间，仍可处理新事务，因此长时间运行事务的影响得到了大幅缓解。

7 EVALUATION

本节通过跨区域实验评估 GeoGauss。

集群设置。 我们的跨区域集群包含 3 个地理分布式节点（对应 3 个副本），分别位于张家口市（华北）、成都市（中国西南）和深圳市（华南）。每个节点（阿里云 ecs.r6e.8clarge 实例）配备 32 个虚拟 CPU、256GB 动态随机存取存储器，运行 CentOS 7.6 操作系统。跨区域节点之间的网络带宽约为 100 Mbps/s。我们在每个区域单独设置一个节点（ecs.c6.8xlarge），以模拟发送 SQL 语句的本地客户端。
竞争系统与配置。我们选择 CockroachDB (CRDB) [67], Calvin [70], Aria[72], CalvinFS [69], Q-Store [57], SLOG [61], 以及无协调 KV 数据库 Anna [72] 进行比较。为考察乐观异步执行和同步验证带来的性能提升，我们还基于 GeoGauss 实现了两个变体 GeoG-S 和 GeoG-A。

• CRDB：为确保公平性，CRDB 配置为内存存储，并配置为从读取行的主区域之外进行陈旧读。按照 CRDB 文档的建议，我们在每个区域额外部署 2 个节点，以最大限度地提升其性能。CRDB 支持强一致性和可串行化隔离。

• Calvin、Aria、CalvinFS 和 Q-Store：Calvin 和 Aria 是两种典型的采用多主复制的确定性数据库。它们复制 SQL 语句，而非写集，并且不提供完整的 SQL 引擎，因此不支持交互式查询。它们支持强一致性和可串行化隔离。Calvin 利用有序锁实现并发，Aria 则依靠依赖分析和事务重排序。对于这两个系统，我们使用 [4] 中的实现，并采用其默认配置。CalvinFS 是一种扩展 Calvin 以实现元数据管理的分布式文件系统。我们使用[6] 中的实现，并采用其默认配置。Q-Store 基于 Calvin 实现，采用面向队列的事务处理方法而非有序锁，以减少调度开销。

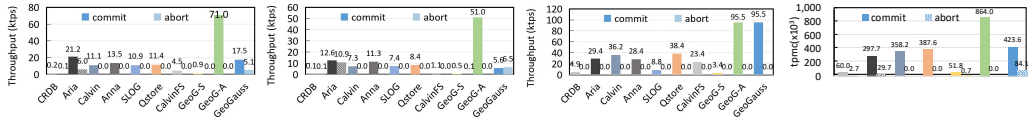
• SLOG：SLOG 采用分片主从架构。输入的 SQL 语句包含在日志中，并复制到从节点。对于跨分片事务，会将其发送到单个节点，以确定全局顺序，该顺序用于在每个分片副本上进行确定性执行。分片副本通过确定性执行重放接收到的日志，从而实现可串行化。

• Anna：Anna 通过合并基于格的复合数据结构（类似于无冲突复制数据类型）实现无等待执行。它仅支持因果一致性和最终一致性（默认），因此已提交的 SQL 请求不会返回已提交或已中止响应。它还支持多种弱隔离级别，默认配置为读已提交隔离。

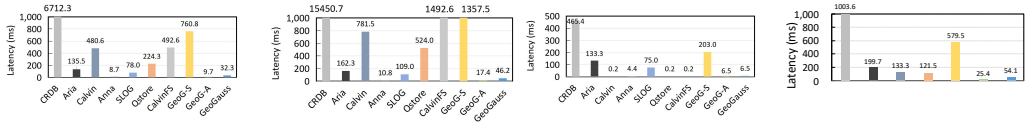
• GeoGauss：我们的系统默认配置为 10ms 的纪元长度。它支持强一致性和多种弱隔离级别（默认采用读已提交）。CRDB 和 GeoGauss 使用标准的基准测试接口，其中每个连接每次只发送一个事务。在这两个系统中，每个节点均配置了 256 个连接。

• GeoG-S：采用同步执行和同步验证，即，在生成快照（$i-1$）之前，不会开始执行纪元 i的事务。

• GeoG-A：移除了纪元概念，并采用异步执行和异步验证。与 Anna 类似，它提供最终一致性，且不保证在
g粒度上 y of ep保持一致性。
工作负载。 我们使用两种流行的基准测试：YCSB [40] 和 TPC-C [21]。对于 YCSB，我们使用一张包含 10 列和 1,000,000 行的表，并将连接数配置为 256。为了将其设为事务基准测试，我们将操作封装在事务中，并让每个事务包含 10 个操作。我们评估 YCSB 工作负载的三种不同变体：1) YCSB-MC（中度争用）： 80% 读和 20% 写，其中 10% 的元组构成热点，被所有查询中的 ~60% 访问（$\theta = 0.8$ 按齐夫分布）。2) YCSB-HC（高度争用）： 50% 读和 50% 写



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



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

Fig. 5. Comparison results on throughput and latency.

其中 10% 的元组构成热点，被所有查询中的 ~75% 访问（$\theta = 0.9$）。3) YCSB-RO（只读）：所有查询均为读取，且采用均匀访问分布。对于 TPC-C 基准测试，我们将仓库数配置为 800，并设置 120 个客户端连接来发送查询请求。由于 Calvin 和 Aria 不提供 SQL 引擎（不支持交互式查询和复杂查询，例如，连接和范围扫描），它们仅支持 New-Order 事务和 Payment 事务。我们按照 [51] 的方式，通过混合 50% 的 New-Order 事务和 50% 的 Payment 事务来构建 TPC-C 基准测试。CalvinFS、SLOG 和 Anna 不支持此 TPC-C 基准测试，因此我们不在这些系统上运行 TPC-C 基准测试。

7.1 跨区域比较结果

我们在跨区域集群上进行了对比实验。图 5 报告了成功提交和中止的事务吞吐量以及平均延迟。在 YCSB-MC 工作负载下，GeoGauss 的吞吐量高于大多数其他系统，并在读密集型工作负载下实现了最高吞吐量。GeoG-A 的速度比 GeoGauss 快，但无法保证强一致性。Anna 和 GeoG-A 都无法向用户响应已提交或中止通知，因为它们无法在时间约束内确定事务状态（它们没有中止事务）。严格来说，它们不是事务系统。GeoG-S 是 GeoGauss 的高度同步变体，它保证强一致性，但代价是性能下降。由于在地理分布式环境中的协调开销高昂，CRDB 速度较慢。Calvin 和 Aria 的吞吐量远高于 CRDB。Calvin 的扩展版本 CalvinFS 使用法定人数协议实现副本一致性，因此性能低于 Calvin。Q-Store 降低了 Calvin 的调度开销，但性能提升有限，因为在地理分布式环境中，主要开销来自协调而非调度。SLOG 的性能低于 Calvin 和 Aria，因为它需要将跨分片事务发送到单个节点以确定全局顺序；在缺乏感知局部性的分片方案的跨区域场景下，这会严重影响性能。SLOG 中对某个数据项的写入和线性化读取必须定向到其主副本，因此 YCSB-RO 上的吞吐量与 YCSB-MC 上的相近。由于采用乐观执行逻辑，GeoGauss 和 Aria 的中止率较高。GeoGauss 基于过时快照进行乐观执行，并在提交前验证执行结果；而 Aria 则依赖依赖图分析来解决冲突。与 Calvin 等悲观锁定方法相比，这些方法的中止率往往较高。需要注意的是，确定性数据库及其扩展不支持标准 TPC-C 基准测试。它们的互操作性有限，而且需要预先知道将访问哪些数据。

在延迟方面，GeoGauss 在 YCSB-MC、YCSB-HC 和 TPC-C 上均优于其他强一致系统。Anna 是一种无需协调、具有最终一致性的键值存储系统（KVS）。GeoG-A 仅支持 CRDT 合并，不支持基于纪元的一致性，因此仅支持最终一致性。

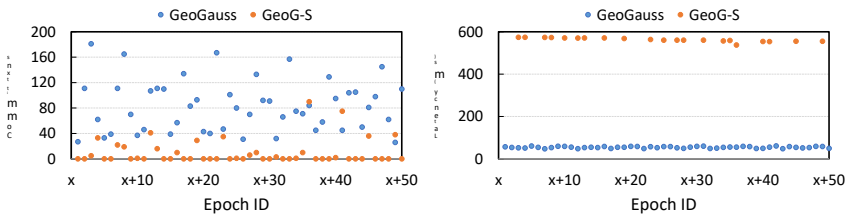
Table 2. Runtime breakdown of a transaction (TPC-C).

	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

一致性。它们都不需要协调，因此延迟低于其他系统。由于在地理分布式环境中等待时间较长，CRDB 和 GeoG-S 的延迟较高。Calvin 和 Aria 需要额外时间来调度这些批处理事务，以实现确定性执行，因此其延迟高于 GeoGauss。在 YCSB-RO 工作负载下，Calvin 及其扩展（CalvinFS 和 Q-Store）的读取延迟非常低，因为它们直接返回本地读取数据，且没有输入解析开销。Calvin 不支持完整的 SQL，而 GeoGauss 的输入查询需要经过 SQL 解析器和优化器处理，因此其延迟高于 Calvin 及其扩展。Aria 需要执行预处理步骤来对输入查询进行依赖分析，因此其延迟远高于 Calvin 及其扩展。

7.2 Performance Gain Analysis

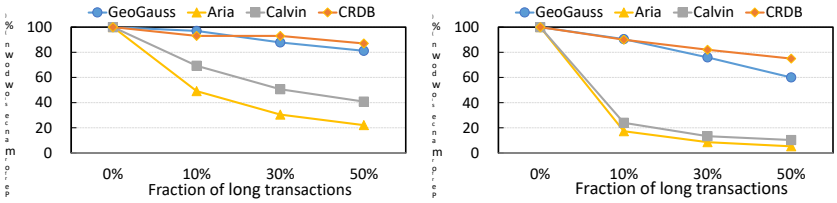
在我们的系统中，事务处理主要包含五个阶段：SQL解析、事务执行、等待最新快照生成（算法1 中的第22行）或同一 cen 的所有远程/本地事务应用完成（算法 1 中的第24行）、合并（算法2）以及持续时间内的日志记录。我们研究了不同阶段的开销，以探究瓶颈。表 2 展示了一个成功提交的 TPC-C 事务在各阶段所花费的平均时间。我们可以看到，在 GeoG-S中，等待阶段占据了运行时间的主要部分。通过采用乐观异步执行（仍然需要同步验证），GeoGauss 大幅减少了等待时间。这是在保证每个纪元基础上的强一致性（通过同步验证）的同时提升性能的关键。相反，GeoG-A 无需等待同步，因此无法提供强一致性。GeoG-S 的合并时间和日志记录时间更短，是因为 GeoG-S 中处理的事务远少于另外两个系统。



(a) Number of commit transactions (b) Latency

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

如上文所述，GeoG-S 中的同步执行需要基于快照（$i-1$）处理纪元 i 的事务，这会对性能产生重大影响。为了研究同步执行的影响，我们进一步测量了 GeoGauss 和 GeoG-S 中每个纪元的已提交事务数量和平均延迟。图 6a 显示了连续多个纪元中的已提交事务数量。在 GeoG-S 中，许多纪元内没有事务提交，并且每隔 2-4 个纪元才会出现一次事务提交。这是因为单程时间约为 30 ms，而 GeoG-S 在等待远程写入时会停止提供服务。



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

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

上一纪元的集合。而 GeoGauss 利用乐观异步执行，能够持续提交事务。图 6b 展示了包含已提交事务的各纪元的平均延迟。由于涟漪效应，延迟远高于单程延迟（~580 对比 ~30 毫秒）。

7.3 Long Running Transactions

如第 6 节关于确定性数据库的区别讨论所述，在处理长时间运行事务时，GeoGauss 应优于确定性数据库，因为确定性数据库的批次之间存在屏障。在本实验中，我们手动为随机选取的 YCSB-MC 事务添加固定的 20 ms/100 ms 延迟，以模拟长时间运行事务。图 7 展示了随着长事务占比变化，系统在吞吐量方面的性能下降情况。Aria 和 Calvin 在处理更多长事务时，吞吐量均大幅下降。如图 7b 所示，在处理更长时间运行的事务时，确定性数据库的性能下降甚至更加显著，例如，当长事务占比为 10% 时，性能下降达到 80%。GeoGauss 对长事务具有更强的鲁棒性。这是因为 GeoGauss 中的事务采用乐观方式读取数据，并将更新写入其私有缓存，同时，新事务可以与长时间运行事务的执行并发处理。值得一提的是，我们算法中的同步点旨在确保远程更新的完整性，而不是在事务批次之间设置全局屏障。CRDB 对长事务也具有较强的鲁棒性，尽管其延迟远高于我们的方法，因为（手动设置的）延迟被隐藏在其 并行提交 阶段中。

表 3. 每笔事务的平均广域网流量（KB/事务）

	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

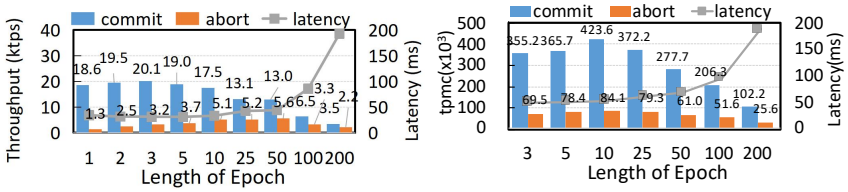
7.4 广域网流量

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 可变配置

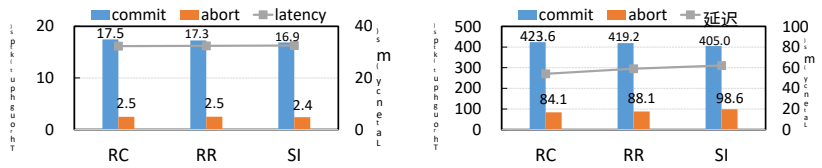
7.5.1 纪元长度。 如第 5.1 节所述，GeoGauss 为每个事务分配一个线程（也就是说，每个连接分配一个线程），并且在事务提交或中止之前不会释放连接。因此，较长的纪元可能导致所有连接都被等待确认的事务占用，系统可能会在一段时间内停止服务。另一方面，这也会影响延迟，因为提交的写入请求必须等待生成纪元快照后才能响应用户。较长的纪元会导致延迟增加，而较短的纪元则可能导致冲突合并过于频繁，同样会降低性能。



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

Fig. 8. Effect of epoch length.

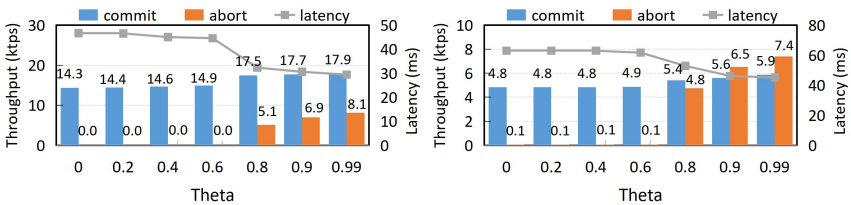
图 8 展示了 YCSB-MC 和 TPC-C 工作负载在不同纪元长度（范围为 1 ms 至 200 ms）下的吞吐量和延迟结果。正如上文分析的那样，吞吐量结果低于预期。延迟一方面取决于纪元长度较短时的单程延迟，另一方面则取决于纪元长度较长时的纪元长度设置。



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

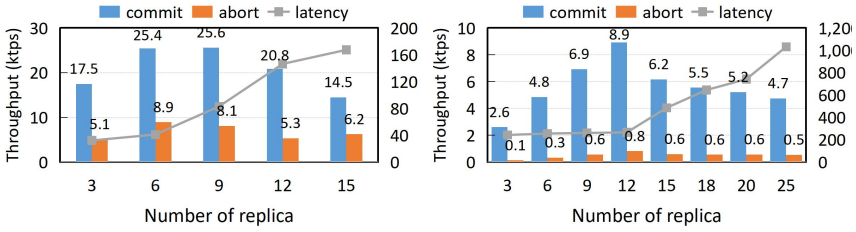
Fig. 9. Performance with different isolation levels.

7.5.2 隔离。 我们在本实验中研究了采用不同隔离级别时的性能。图 9 展示了 YCSB-MC 和 TPC-C 工作负载下的实验结果。不同隔离级别下的吞吐量和延迟结果并没有太大差异，唯一的区别是，隔离级别越高，中止率也越高。这符合预期，因为 RR 和 SI 都需要执行读集验证步骤，这可能会提高中止率。



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

Fig. 10. Performance with different contention levels.



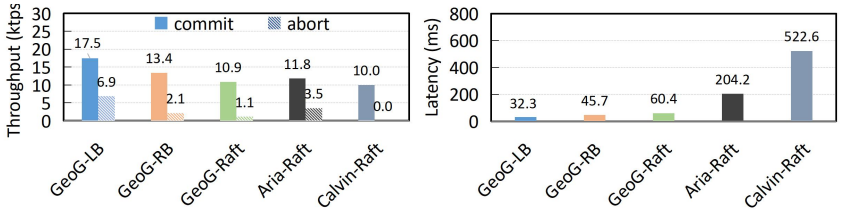
(a) YCSB-MC (China) (b) YCSB-MC (WorldWide)

Fig. 11. Scalability (YCSB-MC).

7.5.3 竞争级别。 为了研究不同竞争级别下的性能，我们在 YCSB-MC 和 YCSB-HC 工作负载中，将θ 参数（偏斜因子）从 0 变化到 0.99。如图 10 所示，随着 θ 增大，由于发生冲突的概率更高，中止率也随之升高；并且，在写密集型工作负载 YCSB-HC 下的中止率甚至高于 YCSB-MC。有趣的是，当 θ 增大时，吞吐量略有提高，而延迟略有降低。这是因为，由冲突导致的中止事务会提前释放线程，而不会造成阻塞（算法 1 中的第24行），从而能够处理更多新的事务（这些事务不会与其他事务发生冲突）并使其提交。如果新事务是只读事务，则会直接返回，这有助于降低延迟并提高吞吐量。

7.6 Scalability

在本实验中，我们研究 GeoGauss 的可扩展性。我们将副本节点的数量从 3 增加到 15，并运行 YCSB-MC。如图 11a 所示，当副本节点数量不超过 9 时，吞吐量稳步提升，之后则开始下降。这是因为副本数量增加会带来更多复制开销，从而降低性能。使用更多副本时，延迟也会增加。这是因为基于纪元的合并需要接收所有副本的更新，这可能会导致更长的同步时间。为研究跨洲环境下的扩展性能，我们搭建了一个跨越多个大洲的全球集群，其中包括部署在五个数据中心（伦敦、新加坡、东京、硅谷和弗吉尼亚）的 25 个节点。我们将副本节点的数量从 3 增加到 25，并运行 YCSB-MC。吞吐量和延迟结果如图 11b 所示。我们可以看到，与中国的可扩展性实验相比，趋势相似。全球集群中的峰值吞吐量低于中国，延迟则长得多，这符合预期。



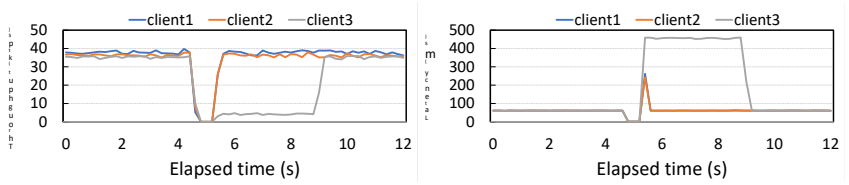
(a) 吞吐量 (b) 延迟

图12. 容错情况下的性能（YCSB-MC）。

7.7 容错

在上述实验中，为了获得最佳性能，关闭了确定性数据库（例如，Calvin 和 Aria）的容错支持；GeoGauss 仅使用本地写集备用服务器，不影响性能。在本实验中，我们在

Calvin、Aria 和 GeoGauss 中启用 Raft复制，并比较这些系统在支持容错时的性能（即，Calvin-Raft、Aria-Raft、GeoG-Raft）。我们还测量了 GeoGauss 在另外两种较弱容错机制下的性能，即本地和远程写集备用服务器（GeoG-LB 和 GeoG-RB）。如图 12 所示，GeoG-Raft 的吞吐量与 Calvin-Raft 和 Aria-Raft 相当。采用较弱的容错机制（例如，GeoG-LB 和 GeoG-RB）时，吞吐量可大幅提高。在延迟方面，由于采用了乐观执行方案，GeoG-Raft 的延迟远低于 Calvin-Raft 和 Aria-Raft。如第 5.2 节所述，GeoG-LB、GeoG-RB 和 GeoG-Raft 分别需要 ~0.5 RTT、 ~1 RTT 和 ~1.5 RTT，因此 GeoG-LB 的延迟低于 GeoG-RB 和 GeoG-Raft。



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

图13. 故障情况下的性能（YCSB-MC）。

为了研究故障情况下的性能波动，我们手动关闭一个节点，并观察 GeoGauss 在故障发生后的表现。图13展示了从各个客户端视角观察到的吞吐量和延迟变化；我们在每个区域都部署了一个连接到服务器节点的客户端。节点故障期间出现的暂时性性能下降，是由于服务被阻塞，即需要等待来自故障节点的更新。得益于基于Raft的成员管理（超时设置为500毫秒），GeoGauss 能够快速响应此故障。此前连接到崩溃节点的 client3 的请求会被路由到其他区域仍在提供服务的节点。但事务请求会在远程区域执行，导致 client3 的吞吐量下降、延迟增加。崩溃节点恢复后，基于Raft的成员管理会检测到这一情况，并让 client3 连接到同一区域的节点。

8 结论

本文提出了 GeoGauss，这是一种具有多主复制架构的强一致、轻协调地理分布式事务数据库。我们已经证明，GeoGauss 通过强一致性和弱隔离，能够显著提升地理分布式事务处理的性能。通过采用基于纪元的输出复制和乐观异步执行，我们的多主OCC算法能够高效地合并冲突，同时在纪元粒度上保证副本的强一致性。GeoGauss 还克服了确定性数据库的缺点，并支持交互式SQL执行，因此具有更广泛的适用性。

致谢

本研究得到国家自然科学基金委员会（62072082、U2241212、62202088、62232009、61925205、62137001、62272093）、国家社会科学基金（21&ZD124）、CCF-华为胡杨林创新研究基金（CCF-HuaweiDB2022003）、中央高校基本科研业务费（N2216012、N2216015）、华为、好未来教育以及北京信息科学与技术国家研究中心（BNRist）的资助。

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：快速且实用的确定性联机事务处理数据库。https://github.com/luyi0619/aria [5] 2022。百度 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：集群分布式数据库系统。https://www.mcobject.com/cluster/ [9] 2022。FaunaDB。https://fauna.com/ [10] 2022。适用于 MySQL 的 Galera 集群。https://galeracluster.com/ [11] 2022。GNU Gzip。https://www.gnu.org/software/gzip/ [12] 2022。gRPC：高性能、开源的通用 RPC 框架。https://grpc.io/ [13] 2022。MySQL Tungsten。https://www.continuent.com/products/tungsten-replicator [14] 2022。MySQL 的主从复制。https://dev.mysql.com/ [15] 2022。openGauss。https://opengauss.org/ [16] 2022。PostgreSQL BDR。https://wiki.postgresql.org/wiki/BDR_项目 [17] 2022。Protocol Buffers。https://developers.google.com/protocol-buffers [18] 2022。Redis CRDT。https://redis.com/blog/diving-into-crdts/ [19] 2022。Riak：企业级 NoSQL 数据库。https://riak.com/ [20] 2022。Facebook 的半同步复制。http://yoshinorimatsunobu.blogspot.com/ [21] 2022。TPC-C 主页。https://www.tpc.org/tpcc/ [22] 2022。YugabyteDB：分布式 SQL 数据库。https://www.yugabyte.com/ [23] 2022。ZeroMQ：开源通用消息库。https://zeromq.org/ [24] Daniel J Abadi 和 Jose M Faleiro。2018。确定性数据库系统概览。《Commun. ACM》61，9（2018），78–88。[25] Michael Abebe、Brad Glasbergen 和 Khuzaima Daudjee。2020。DynaMast：复制系统的自适应动态主控。载于 2020 年 IEEE 第 36 届国际数据工程会议（ICDE）。IEEE，1381–1392。[26] Michael Abebe、Brad Glasbergen 和 Khuzaima Daudjee。2020。MorphoSys：分布式数据库系统的自动物理设计演变。《VLDB 基金会会议录》13，13（2020），3573–3587。[27] Paulo Sérgio Almeida、Ali Shoker 和 Carlos Baquero。2015。通过增量变异实现高效的基于状态的无冲突复制数据类型。载于网络系统国际会议。施普林格，62–76。[28] Paulo Sérgio Almeida、Ali Shoker 和 Carlos Baquero。2018。增量状态复制数据类型。《J. Parallel and Distrib. Comput.》111（2018），162–173。[29] Peter Alvaro、Neil Conway、Joseph M. Hellerstein 和 David Maier。2017。Blazes：分布式程序的协调分析与放置。《ACM 数据库系统汇刊》42，4，文章 23（2017 年 10 月），31 页。[30] Peter Alvaro、Neil Conway、Joseph M Hellerstein 和 William R Marczak。2011。Bloom 中的一致性分析：一种 CALM 且汇集的方法。载于 CIDR。249–260。[31] Mohammad Javad Amiri、Divyakant Agrawal 和 Amr El Abbadi。2019。Caper：一种跨应用许可型区块链。《VLDB 基金会会议录》12，11（2019），1385–1398。[32] Mohammad Javad Amiri、Divyakant Agrawal 和 Amr El Abbadi。2021。SharPer：在网络集群上对许可型区块链进行分片。载于 2021 年国际数据管理会议论文集。76–88。[33] Elli Androulaki、Artem Barger、Vita Bortnikov、Christian Cachin、Konstantinos Christidis、Angelo De Caro、David Enyeart、Christopher Ferris、Gennady Laventman、Yacov Manevich 等。2018。Hyperledger Fabric：面向许可型区块链的分布式操作系统。载于第十三届 EuroSys 会议论文集。1–15。[34] H. Avni、A. Aliev、O. Amor、A. Avitzur、I. Bronshtein、E. Ginot、S. Goikhman、E. Levy、Lu Levy、I.、F. 和 L. Mishali。2020。利用主存和多核实现工业级联机事务处理。《VLDB 基金会会议录》13，12（2020），3099–3111。[35] Peter Bailis、Alan Fekete、Michael J. Franklin、Ali Ghodsi、Joseph M. Hellerstein 和 Ion Stoica。2014。数据库系统中的协调避免。《VLDB 基金会会议录》8，3（2014），185–196。[36] Peter David Bailis。2015。分布式数据库中的协调避免。加州大学伯克利分校。[37] Michael J Cahill、Uwe Röhm 和 Alan D Fekete。2009。快照数据库的可串行化隔离。《ACM 数据库系统汇刊》（TODS）34，4（2009），1–42。[38] Prima Chairunnanda、Khuzaima Daudjee 和 M Tamer Özsu。2014。ConfluxDB：面向分区快照隔离数据库的多主复制。《VLDB 基金会会议录》7，11（2014），947–958。[39] Neil Conway、William R. Marczak、Peter Alvaro、Joseph M. Hellerstein 和 David Maier。2012。分布式编程中的逻辑与格。载于云计算研讨会论文集（SoCC '12）。1:1–1:14。

[40] Brian F. Cooper、Adam Silberstein、Erwin Tam、Raghu Ramakrishnan 和 Russell Sears。2010。使用 YCSB 对云服务系统进行基准测试。载于第 1 届 ACM 云计算研讨会论文集（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 等。2013。Spanner：谷歌的全球分布式数据库。ACM《计算机系统汇刊》（TOCS）31，3（2013），1–22。[42] Hung Dang、Tien Tuan Anh Dinh、Dumitrel Loghin、Ee-Chien Chang、Qian Lin 和 Beng Chin Ooi。2019。通过分片实现区块链系统的可扩展性。载于 2019 年国际数据管理会议论文集。123–140。[43] Giuseppe DeCandia、Deniz Hastorun、Madan Jampani、Gunavardhan Kakulapati、Avinash Lakshman、Alex Pilchin、Swaminathan Sivasubramanian、Peter Vosshall 和 Werner Vogels。2007。Dynamo：亚马逊的高可用键值存储。ACM SIGOPS《操作系统评论》41，6（2007），205–220。[44] Sameh Elnikety、Steven Dropsho 和 Fernando Pedone。2006。Tashkent：将持久性与事务排序相结合，实现高性能、可扩展的数据库复制。载于第 1 届 ACM SIGOPS/EuroSys 欧洲计算机系统会议 2006 论文集（EuroSys '06）（EuroSys '06）。117–130。[45] Jose M Faleiro、Daniel J Abadi 和 Joseph M Hellerstein。2017。通过提前实现写入可见性提升事务性能。VLDB 基金会会议录 10，5（2017）。[46] 蚂蚁集团。2022。OceanBase。https://open.oceanbase.com/ [47] Suyash Gupta、Sajjad Rahnama、Jelle Hellings 和 Mohammad Sadoghi。2020。ResilientDB：全球规模的弹性区块链架构。arXiv 预印本 arXiv:2002.00160（2020）。[48] Rachael Harding、Dana Van Aken、Andrew Pavlo 和 Michael Stonebraker。2017。分布式并发控制的评估。VLDB 基金会会议录 10，5（2017），553–564。[49] Jelle Hellings 和 Mohammad Sadoghi。2021。ByShard：拜占庭环境中的分片。VLDB 基金会会议录 14，11（2021），2230–2243。[50] Avinash Lakshman 和 Prashant Malik。2010。Cassandra：一种去中心化结构化存储系统。ACM SIGOPS《操作系统评论》44，2（2010），35–40。[51] Yi Lu、Xiangyao Yu、Lei Cao 和 Samuel Madden。2020。Aria：一种快速实用的确定性联机事务处理数据库。VLDB 基金会会议录 13，12（2020），2047–2060。[52] Yi Lu、Xiangyao Yu、Lei Cao 和 Samuel Madden。2021。分布式联机事务处理数据库中的基于纪元的提交与复制。VLDB 基金会会议录 14，5（2021），743–756。[53] Yi Lu、Xiangyao Yu 和 Samuel Madden。2019。STAR：通过非对称复制扩展事务处理能力。VLDB 基金会会议录 12，11（2019），1316–1329。[54] 中本聪。2008。比特币：一种点对点电子现金系统。《去中心化商业评论》（2008），21260。[55] Pincap。2022。TiDB。https://pingcap.com/products/tidb [56] Nuno Preguiça。2018。无冲突复制数据类型：概述。arXiv 预印本 arXiv:1806.10254（2018）。[57] Thamir Qadah、Suyash Gupta 和 Mohammad Sadoghi。2020。Q-Store：通过面向队列的执行与通信实现分布式多分区事务。载于第 23 届国际数据库技术扩展会议论文集（EDBT）。73–84。[58] Thamir M Qadah 和 Mohammad Sadoghi。2018。Quecc：一种面向队列的无控制并发架构。载于第 19 届国际中间件会议论文集。13–25。[59] Ian Rae、Eric Rollins、Jeff Shute、Sukhdeep Sodhi 和 Radek Vingralek。2013。F1 中的在线异步模式变更。VLDB 基金会会议录 6，11（2013 年 8 月），1045–1056。[60] Sajjad Rahnama、Suyash Gupta、Rohan Sogani、Dhruv Krishnan 和 Mohammad Sadoghi。2021。RingBFT：分片环形拓扑上的弹性共识。arXiv 预印本 arXiv:2107.13047（2021）。[61] Kun Ren、Dennis Li 和 Daniel J. Abadi。2019。SLOG：可串行化、低延迟的地理复制事务。VLDB 基金会会议录 12，11（2019 年 7 月），1747–1761。[62] Kun Ren、Alexander Thomson 和 Daniel J Abadi。2014。确定性数据库系统优缺点的评估。VLDB 基金会会议录 7，10（2014），821–832。[63] Marc Shapiro、Nuno Preguiça、Carlos Baquero 和 Marek Zawirski。2011。收敛且可交换的复制数据类型的全面研究。博士学位论文。Inria–Centre Paris-Rocquencourt；INRIA。[64] Marc Shapiro、Nuno Preguiça、Carlos Baquero 和 Marek Zawirski。2011。无冲突复制数据类型。载于自稳定系统研讨会。386–400。[65] Marc Shapiro、Nuno Preguiça、Carlos Baquero 和 Marek Zawirski。2011。无冲突复制数据类型。载于自稳定系统研讨会论文集（SSS '11）。386–400。[66] Chrysoula Stathakopoulou、Matej Pavlovic 和 Marko Vukolić。2022。让状态机复制的可扩展性变得简单。载于第十七届欧洲计算机系统会议论文集。17–33。[67] Rebecca Taft、Irfan Sharif、Andrei Matei、Nathan VanBenschoten、Jordan Lewis、Tobias Grieger、Kai Niemi、Andy Woods、Anne Birzin、Raphael Poss 等。2020。CockroachDB：弹性地理分布式 SQL 数据库。载于会议论文集。

《*2020* 年 *ACM SIGMOD* 国际数据管理会议》论文集. 1493–1509. [68] Alexander Thomson 和 Daniel J Abadi. 2010. 数据库系统中确定性设计的优势。*VLDB* 基金会论文集3, 1-2 (2010), 70–80. [69] Alexander Thomson 和 Daniel J Abadi. 2015. CalvinFS：面向分布式文件系统的一致性广域网复制与可扩展元数据管理。载于第 *13* 届 *USENIX* 文件与存储技术会议论文集（*FAST 15*）。1–14。[70] Alexander Thomson、Thaddeus Diamond、Shu-Chun Weng、Kun Ren、Philip Shao 和 Daniel J Abadi. 2012. Calvin：面向分区数据库系统的快速分布式事务。载于*2012* 年 *ACM SIGMOD* 国际数据管理会议论文集。1–12。[71] Alexandre Verbitski、Anurag Gupta、Debanjan Saha、Murali Brahmadesam、Kamal Gupta、Raman Mittal、Sailesh Krishnamurthy、Sandor Maurice、Tengiz Kharatishvili 和 Xiaofeng Bao. 2017. Amazon Aurora：高吞吐量云原生关系型数据库的设计考量。载于*2017* 年 *ACM* 数据管理国际会议论文集（*SIGMOD '17*）。1041–1052. [72] Chenggang Wu、Jose M Faleiro、Yihan Lin 和 Joseph M Hellerstein. 2019. Anna：适用于任意规模的键值存储。*IEEE* 知识与数据工程汇刊33, 2 (2019), 344–358. [73] Chenggang Wu、Vikram Sreekanti 和 Joseph M Hellerstein. 2019. Anna 中分层云存储的自动扩缩。*VLDB* 基金会论文集12, 6 (2019), 624–638. [74] Chang Yao、Divyakant Agrawal、Gang Chen、Qian Lin、Beng Chin Ooi、Weng-Fai Wong 和 Meihui Zhang. 2016. 利用多核内存系统中的单线程模型。*IEEE* 知识与数据工程汇刊28, 10 (2016), 2635–2650.

收稿日期：2022年7月；修订日期：2022年10月；录用日期：2022年11月