软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(1):455−480 [doi: 10.13328/j.cnki.jos.006822]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

多副本分布式事务处理关键技术及典型数据库系统综述*

黄纯悦, 彭 起, 张拂晓, 王声溢, 罗 成, 张岩峰, 于 戈

(东北大学 计算机科学与工程学院, 辽宁 沈阳 110169)

通信作者: 张岩峰, E-mail: zhangyf@mail.neu.edu.cn

摘 要: 数据复制是分布式数据库提高可用性的重要手段, 通过在不同区域放置数据库的部分副本, 还可以提高本地读写操作的响应速度, 增加副本数量也会提升读负载的线性扩展能力. 考虑到这些优良特性, 近年来国内外都出现了众多多副本分布式数据库系统, 包括 Google Spanner、CockroachDB、TiDB、OceanBase 等一系列主流的工业界系统, 也出现了包括 Calvin、Aria、Berkeley Anna 等一系列优秀的学术界系统. 然而, 多副本数据库带来诸多收益的同时, 也带来了一致性维护、跨节点事务、事务隔离等一系列挑战. 总结分析现有的复制架构、一致性维护策略、跨节点事务并发控制等技术, 对比几个代表性多副本数据库系统之间在分布式事务处理方面上的差异与共同点, 并在阿里云环境下搭建跨区域的分布式集群环境, 对几个代表性系统的分布式事务处理能力进行了实验测试分析.

关键词: 分布式事务处理; 分布式数据库; 数据复制; 确定性数据库; 跨区域复制数据库

中图法分类号: TP311

中文引用格式: 黄纯悦, 彭起, 张拂晓, 王声溢, 罗成, 张岩峰, 于戈. 多副本分布式事务处理关键技术及典型数据库系统综述. 软件学报, 2024, 35(1): 455–480. http://www.jos.org.cn/1000-9825/6822.htm

英文引用格式: Huang CY, Peng Q, Zhang FX, Wang SY, Luo C, Zhang YF, Yu G. Survey on Key Techniques of Multi-replica Distributed Transaction Processing and Representative Database Systems. Ruan Jian Xue Bao/Journal of Software, 2024, 35(1): 455–480 (in Chinese). http://www.jos.org.cn/1000-9825/6822.htm

Survey on Key Techniques of Multi-replica Distributed Transaction Processing and Representative Database Systems

HUANG Chun-Yue, PENG Qi, ZHANG Fu-Xiao, WANG Sheng-Yi, LUO Cheng, ZHANG Yan-Feng, YU Ge

(School of Computer Science and Engineering, Northeastern University, Shenyang 110169, China)

Abstract: Data replication is an important way to improve the availability of distributed databases. By placing multiple database replicas in different regions, the response speed of local reading and writing operations can be increased. Furthermore, increasing the number of replicas can improve the linear scalability of the read throughput. In view of these advantages, a number of multi-replica distributed database systems have emerged in recent years, including some mainstream systems from the industry such as Google Spanner, CockroachDB, TiDB, and OceanBase, as well as some excellent systems from academia such as Calvin, Aria, and Berkeley Anna. However, these multi-replica databases bring a series of challenges such as consistency maintenance, cross-node transactions, and transaction isolation while providing many benefits. This study summarizes the existing replication architecture, consistency maintenance strategy, cross-node transaction concurrency control, and other technologies. It also analyzes the differences and similarities between several representative multi-replica database systems in terms of distributed transaction processing. Finally, the study builds a cross-region distributed cluster environment on Alibaba Cloud and conducts multiple experiments to study the distributed transaction processing

* 基金项目: 国家自然科学基金 (62072082, U2241212, U1811261, 62202088); 辽宁省重点研发计划 (2020JH2/10100037); 中央高校基本科研业务费 (N2216015, N2216012)
黄纯悦和彭起为共同第一作者.
收稿时间: 2022-05-09; 修改时间: 2022-07-26, 2022-09-19; 采用时间: 2022-10-19; jos 在线出版时间: 2023-05-24
CNKI 网络首发时间: 2023-05-26

performance of these several representative systems.

Key words: distributed transaction processing; distributed database; data replication; deterministic database; cross-region replication database

1 引 言

数据库系统是计算机系统中的关键组成部分. 随着数据量的增加, 跨区域业务变多, 单机数据库很难满足实际业务场景需求, 分布式数据库应运而生. 由于业务在地理上分散场景, 大型商业公司一般都需要构建高可用的全球分布式数据库系统. 通常, 数据复制是保证可用性和分区容错性最常用的手段.

分布式数据库的数据复制通常是在不同区域放置数据库的部分副本. 第一, 通过增加副本数支持并行的读操作, 使数据复制可以提供读负载的线性扩展能力. 第二, 读负载可重定向至就近服务器执行, 这提高了读操作的响应速度. 第三, 数据复制能够实现数据库的高可用. Spanner[1]、CockroachDB[2]、OceanBase (https://dbdb.io/db/oceanbase) 等都采用了分片-主从的数据复制模式, 由此构建地理上分布式的高可用高扩展性数据库系统, 从而支撑了公司的跨地域业务. OceanBase 的三地五副本复制策略大幅提高了系统的可靠性、可用性和性能, 因而在以其为基础的金融支付系统中广泛使用, 支撑了庞大的业务需求. openGauss (https://opengauss.org/en/)、MySQL 等数据库系统采用了主从数据复制策略, 但未采用分片策略, 即放置多个全副本但只有主节点接受写操作请求. 在 openGauss 中, 从副本通常用来做容灾, 并不对用户提供实际读写业务支持, 并且未采用分片策略使其可扩展性有限. 另外, 还有一部分数据库系统采用了多主数据复制策略, 即在多个区域放置数据库的全副本, 每个全副本都支持读写操作请求. Aria[3]、Calvin[4]是典型的多主复制数据库系统, MySQL、PostgreSQL 也可以通过附加插件的方式提供多主支持. 多主数据库系统在实际场景中能够提供高可用和高性能, 但不可避免存在写放大[5]问题.

近年来, 云计算技术[6,7]为分布式数据库系统提供了近乎无限的计算和存储资源. 高速主干网络的快速发展为跨区域大规模部署数据库集群创造了有利条件. 借助于云计算, 分布式数据库实现了性能上的跨越式提升, 大大促进了多副本数据库的发展. 那么多副本数据库系统都有哪些架构模式? 多副本数据库系统应采用何种数据模型? 多副本数据库系统如何与实际业务场景相匹配? 多副本数据库系统如何保证高可用、数据一致性以及如何提高系统事务处理的并发度? 这些系统在跨区域环境下部署的事务处理实际运行效果怎样? 本文将通过总结现有的学术界和工业界的主要工作及所采用的关键技术, 分析典型系统的事务处理方法, 通过实际大规模实验评估测试, 来尝试解答这些问题.

目前, 已有多篇文章对数据库事务处理技术进行了综述: Harding 等人[8]、赵泓尧等人[9]均综述了内存数据库的并发控制算法; Hu 等人[10]从并发控制、日志记录、索引等维度对内存数据库进行了综述, 其包括了部分分布式和单机系统; Song 等人[11]综述了数据复制的一致性模型; Tarun 等人[12]综述了分布式数据库的分片、分配和复制策略; Moiz 等人[13]综述了数据复制工具. 近年来, 出现了许多基于多主复制架构和无主复制架构的实际系统, 以更好地支持分布式数据库多读多写的性质, 与此同时复制架构的演变带来了诸多新的挑战. 本文综合了复制架构、数据存储模型、操作一致性模型、并发控制算法等维度, 结合具体的多副本分布式数据库系统进行分析, 并部署跨地理区域的分布式环境对典型系统进行了大规模实验测试. 这些对新型系统的分析是对已有综述工作的有益补充, 大规模评测结果也将为研究人员提供有意义的参考.

第 2.1 节将系统介绍复制数据库架构, 以主从、多主和无主为切入点, 分析对比各种架构的区别与联系. 第 2.2 节介绍各种数据模型下的数据复制, 涵盖关系型、KV、文档、列式、图、账本等数据模型. 第 2.3 节着重介绍分布式数据库的一致性模型, 并选取常用的典型一致性模型做系统阐述. 第 2.4 节着眼于明晰复制架构和并发控制融合技术的特点. 第 3 节选取典型的数据库系统, 结合架构、一致性和并发等 3 个方面, 深入对比和分析它们之间的差异与共同点. 第 4 节选取部分典型系统做跨区域的大规模分布式实验评估, 使用 YCSB 和 TPC-C 基准评估它们的分布式事务处理能力. 最后, 本文总结现有系统架构的优缺点, 并结合云原生存算分离架构、新硬件和端-边-云协同领域研究的最新进展对未来工作进行了展望.

2 多副本分布式数据库的关键技术

2.1 复制架构

2.1.1 主从复制架构及典型系统

主从复制架构[14,15]由一个主节点和多个从节点构成 (如图 1(a) 所示), 主节点可处理读写操作, 而从节点只支持读. 在主节点请求压力非常大的场景下, 可以把大量对实时性要求不是特别高的读请求分摊到从节点上, 实现读写分离和负载均衡. 然而, 系统的写负载并未被分摊, 因而, 主节点也会因为频繁的写请求而过载. 主从复制有异步复制、半同步复制、全同步复制 3 种模式. 这 3 种复制方式的最本质区别是主从副本数据的实时一致性程度. 不同的复制方式影响主从副本间的一致性和主节点的写入操作延迟.

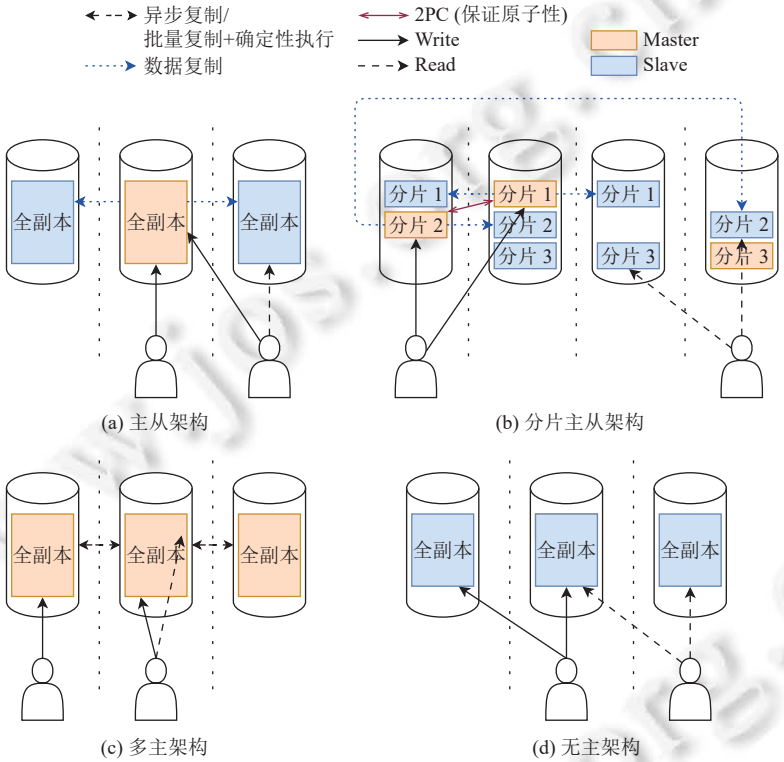


图 1 复制架构系统图示

当前, 许多商业数据库系统采用主从复制结构. MySQL、PostgreSQL 等数据库系统采用的就是主从复制架构, 两者均可以通过一定的配置实现异步、半同步和全同步复制. 例如, MySQL 默认支持异步复制, 但是可通过附加插件提供半同步复制, MySQL Cluster (https://www.mysql.com/cn/products/cluster/) 支持全同步复制. HBase (https://hbase.apache.org/) 支持异步复制和全同步复制. 由于对数据库响应要求的增大, 有时不得不以内存为数据库的存储介质, 大部分数据库 MySQL、PostgreSQL 等都提供了对内存存储引擎的支持, openGauss MOT (https://opengauss.org/en/docs/1.0.0/docs/Developerguide/introducing-mot.html) 也是典型的内存优化型的数据库系统. MOT 将内存作为存储介质, 采用基于非一致内存访问 (NUMA) 的机制, 充分利用了数据的局部性特征. 另外, MOT 采用了乐观并发控制 (OCC) 事务处理机制, 也加快了事务的处理速度. 因此, MOT 实现了软件、存储和硬件方面的优化.

当数据集很大甚至单节点无法存储全部数据或者面临很高查询压力的时候, 需要将数据拆分成分片或叫分区 (如图 1(b) 所示). 分片复制从本质上仍是主从架构, 分片的多个副本可以放在单区域或跨区域不同的节点上. 在

这些节点中有一个为主分片, 所有的写入操作都将路由到这个分片进行. 因此分片复制架构除了具有高扩展性高可用的优点外, 还可以降低跨域的读延迟. 然而它也存在几个缺点. ① 分片主从复制将所有客户端的写请求路由到主节点, 这可能会产生跨节点传输延迟. ② 分片架构中跨分片写入通常采用两阶段提交 (2PC)[16]协议来保证原子性, 这意味着将产生多次往返确认时延. Carousel[17]引入了一种准备共识协议, 通过并行化 2PC 和共识可以在一次广域网往返中完成准备阶段, 在一定程度上可以降低延迟. ③ 增加新的区域服务节点, 可能需要对数据库重新进行分片分配, 使得数据库的维护工作变得复杂. 近年来, 出现了许多商业数据库例如 Google Spanner、CockroachDB、TiDB、OceanBase、YugabyteDB (https://www.yugabyte.com/), 它们都采用了这种分片主从复制架构, 都可以支持超大数据量的全球部署, 具有良好的扩展性和高可用性.

2.1.2 多主复制架构及典型系统

多主复制架构使每个主节点维护一个完整的数据副本, 各个节点都可处理读写请求, 每个主节点同时也充当其他主节点的从节点 (如图 1(c) 所示). 多主复制架构和主从复制架构都存在着写放大挑战, 每个写入操作都在所有副本上冗余执行. 不同的是, 分片主从架构可以有针对性地在访问热点区域放置副本, 无需将所有数据设置副本. 与此相反, 多主架构在每个节点就需要都放置全副本, 因此多主架构的写放大问题更为严重.

与主从架构不同, 多主复制架构中请求只需路由到最近的数据中心即可, 这减少了跨节点的请求路由延迟, 同时也避免了跨节点事务; 然而, 并发更新可能会导致节点间的写-写冲突. 虽然可以通过分布式锁或 TO (时间戳排序) 来处理跨节点冲突, 但在节点间多次往返处理的代价是十分昂贵的, 因此很少有数据库采用这种方式解决跨节点的冲突. 之前, 多主数据库一般都采用避免冲突的策略或用后写胜利来解决冲突. 例如 MySQL Tungsten (https://www.continuent.com/products/tungsten-replicator) 采用“纪律”来避免冲突、PostgreSQL BDR (https://www.enterprisedb.com/docs/bdr/latest/) 采用后写胜利的方式处理冲突. 近年来, 学术界出现了一批确定性多主数据库, 例如 Aria[3]、Calvin[4]、基于 Calvin 的云原生数据库 FaunaDB[18], 它们都采用确定性排序的方法解决冲突并保证所有节点执行的结果相同, 还出现了例如 Anna[19]的无协调多主数据库, 它采用格的 ACI 属性 (结合性、交换性、幂等性) 对冲突进行合并. 这些都将在后文进行详细介绍.

2.1.3 无主复制架构及典型系统

无主复制, 意味着各个副本都可以接收用户的写或读操作 (如图 1(d) 所示), 并使用仲裁协议通过比较单调版本号来避免过时的读取. 通常, 无主复制需要通过 Quorum 机制[20]来保证读到最新的副本. 假设有 n 个节点, 用户的写请求发送到 w 个节点, 读请求发送到 r 个节点, 三者数量关系需满足 $w+r>n$. 根据 Quorum 机制, 读请求会接收到来自至少一个节点的最新值和来自其他节点的陈旧值, 因此可以根据最新的版本号来确定最新值. 然而, 这种没有稳定主节点的基于 Quorum 的方法产生不必要的数据传输, 降低系统的吞吐量. 因此, 无主数据库需要在高性能和一致性之间做出平衡. DynamoDB[21]、Riak (https://riak.com/products/riak-kv/index.html) 和 Cassandra[22]都通过无主复制提供高可用键值存储服务. 但是, 所有用户的写入请求都需要发送到多个节点, 可能会产生跨节点延迟. Amazon Aurora (https://aws.amazon.com/rds/aurora/) 采用存算分离架构, 其存储层采用 Quorum 机制进行读写, 通常设置 6 个副本, 写成功 4 个副本即可返回写请求, 但其在计算层默认采用主从架构, 只能在一个数据库实例写入, 其余可用区的实例分担读请求, 其可以通过存储层的 Quorum 机制读到最新版本, 也可允许过时的读. 另外, Amazon Aurora 还提供了多主支持, 可配置最多不超过 4 个的读写节点和若干只读节点, 其在页面级别检测冲突, 并将写入冲突作为死锁错误报告给应用程序.

2.1.4 复制架构小结

本节对复制架构进行总结, 如表 1 所示. 多主架构只需要将读写请求路由到任意一个节点即可, 这个特性使其十分容忍节点失效问题. 无主架构读写请求需要路由到多个节点, 并要满足 Quorum 机制, 对于不超过半数的节点失效具有中等的容忍能力, 而主从架构通常通过主从切换才可以容忍主节点失效问题, 容忍能力较低. 分片主从架构读写请求只需路由到事务涉及分片, 因此某些节点失效对不在失效节点上分片的请求没有影响. 分片主从架构的分片属性使其具有最好的拓展性, Spanner 和 CockroachDB 都可以做到全球跨域的部署并支撑超大规模数据, 但其分片属性相较于其他架构具有最高的系统复杂度. 多主和分片主从架构的冲突处理策略相比于一般采用的后写胜利策略的无主架构更加复杂. 另外, 多主和无主架构存在着更加严重的写放大问题.

表 1 对复制架构的总结

配置	主从	分片主从	多主	无主
写请求路由节点	唯一主节点	事务涉及分片主节点	任意节点	多个节点
读请求路由节点	从节点/主节点	事务涉及分片从节点/主节点	任意节点	多个节点
冲突处理	容易	困难	困难	中等
可拓展性	一般	强	一般	一般
系统复杂度	低	高	中	中
写放大问题	存在	存在	严重	严重
节点失效问题	低容忍	中容忍	高容忍	中容忍
典型系统	openGauss	Google Spanner	Calvin	DynamoDB

2.2 各种数据模型下的数据复制

数据模型分为关系型和非关系型. 关系模型是指用二维表的形式表示实体和实体间联系的数据模型[23]. 非关系型支持更灵活的数据模型和较弱的一致性, 常用于存储非结构化的数据. 非关系型模型有: 键值对 (key-value, KV) 模型、文档类模型、图模型、列式模型等 (如表 2 所示). 实质上, 文档型和列模型都可以看作 KV 型的子类. 关系型数据库支持事务, 为保证事务的正确可靠, 事务必须具备 ACID 特性[24]. 一般来说, 只有关系型数据库具有事务的概念, 但随着近几年的发展, 一些非关系型数据库也开始支持事务和 ACID 特性, 如图数据库 Neo4j 和只支持行级的事务的列式数据库 HBase. 关系型数据库侧重保证数据的完整性和安全性, 而非关系数据库更侧重于可扩展性和灵活性[25].

表 2 对数据模型的总结

配置	关系型	KV	文档	列式	图	账本
数据组织	二维表	键值对 (key-value)	文档 (key-document)	列簇 (key-column)	图结构	区块
查询方式	SQL	Get (Key)	Find (查询条件)等多种方式	Get (Key)	Cypher/Gremlin等图查询语言	用区块哈希值查找区块
存储结构复杂度	中	低	低	低	高	低
事务支持度	支持	部分支持	部分支持	部分支持	基本支持	支持 (智能合约)
复制单元	日志/SQL语句	键值对	操作日志	列簇	日志	区块
典型代表	MySQL	RocksDB	MongoDB	HBase	Neo4j	Ethereum
特点	数据结构化	复制快高并发读写	海量数据访问	便于分布扩展和大数据查询	快速处理复杂多级检索	拜占庭容错
挑战	海量数据访问	用LSM-Tree实现的KV存储的写放大问题			图结构的分布式存储	高并发事务执行

因为数据组织形式不同, 各种数据模型在做数据复制时复制单元有所不同 (如表 2 所示): 关系型数据库的复制单元多为二进制日志 (binary log 也称 binlog, 存储相应 SQL 语句的逻辑日志) 或重做日志 (redo log, 存储数据库变更的物理日志). 例如 MySQL, 主节点将变更记录在 binlog 中, 从节点通过独立的线程拷贝这些记录, 然后重放. 在确定性数据库中, 节点直接复制 SQL 语句来进行确定性执行. 文档型与关系型类似, 其复制单元为操作日志 (oplog), 如 MongoDB (https://www.mongodb.com/), 在做数据复制时, 主节点将在其上的所有操作记录在 oplog 中, 从节点定期轮询主节点获取 oplog, 进行重放. 图数据库在做数据复制时, 复制单元也是日志, 比如 Neo4j (https://neo4j.com/), 只读节点异步复制核心节点的事务日志进行数据更新. 图数据库的一大特点是可以快速进行复杂多级的检索, 但也面临着图结构的分布式存储问题.

列式数据库 HBase 的复制是以列簇为单位的, 每个列簇都可以设置是否进行复制, 列式结构最大的特点就是以列存储, 便于分布扩展. KV 型数据库复制单元为键值对, 在做数据复制时比关系型复制速度快, 理论上 KV 型多副本数据库主从同步要更容易. 但 KV 型存储也依旧面临着一些挑战. RocksDB (http://rocksdb.org/)、LevelDB

(https://github.com/google/leveldb) 等都是基于 LSM-Tree 实现的 KV 型存储. LSM-Tree[26]是专门为 KV 系统设计的面向磁盘的数据结构, 能将离散的随机写请求都转换成批量的顺序写请求, 以此提高写性能, 但会造成写放大严重[27]. LSM-Tree 的写放大问题实质上是: 压缩 (Compaction) 过程中为了保证数据有序进行大量数据重写, 每写 1 单位数据带来 k 单位的数据写盘. 如 RocksDB 中的 level style compaction 机制, 每次拿上一层的所有文件和下一层合并, 假设下一层大小是上一层的 k 倍, 这样单次合并的写放大就是 k 倍, 整体写代价大幅提高.

账本 (ledger) 数据模型主要用于支持拜占庭容错的区块链应用中, 如 Bitcoin[28], Ethereum[29]和 Fabric[30], 但它们的记账模式又有所不同. Bitcoin 采用 UTXO 的记账模式, Ethereum 使用账户-余额的记账模式. Fabric 中的账本由世界状态和区块链两部分组成. 前者以键值对形式组织数据, 后者以区块形式组织数据. Ethereum 的账本里的数据项以区块的结构组织, 区块内包括区块头, 交易信息等. 它们都支持事务处理, 链上节点间进行数据复制时以区块为复制单元, 所有节点维护一个共同的账本[31]. 然而, 区块链要求不但对交易内容支持拜占庭容错, 也要求对交易顺序进行拜占庭容错, 往往需要由单一节点 (如 Bitcoin 等公有链) 或排序服务共识组的 leader 节点 (如 Fabric 等联盟链) 串行执行交易生成区块, 其挑战在于高并发事务执行和如何获得高吞吐量.

2.3 副本一致性

一致性在不同应用场景下有着很多的解释, 单机数据库系统中代表 ACID[24]属性的 C (consistency), 它是指在事务执行前后, 数据库的完整性约束没有被破坏. 分布式系统的 CAP[32]定理中的 C (consistency), 它是指分布式系统中所有节点对于客户来说似乎只在一台机器上, 看到的读取和写入执行都是按照操作到达的顺序响应操作, 这也是线性一致性的定义. 线性一致性是程序能实现的最高一致性级别, 也是用户期望的一致性级别. 顺序一致性[33−35]相较于线性一致性不要求操作顺序和全局时钟下操作发生的顺序一致, 但要保证所有节点读写操作的结果等价于这些操作按照某个特定顺序执行的结果并且内部事务发生的顺序符合节点内时间顺序. 因果一致性级别下, 各节点对于具有因果关系[36]的读写操作按顺序执行, 而对于没有因果关系的操作没有顺序执行要求. 最终一致性则没有节点上数据操作行为的顺序要求, 它更多描述的是节点间最终数据是否相同的状态, 是一种以用户为中心的一致性模型. 提供最终一致性的系统随着时间的推移, 不同节点上的数据总是在向趋同的方向变化. 在一段时间后, 多副本间的数据会最终达到一致状态.

在单机上事务发生的逻辑全序关系是可以通过同一个系统时钟确定, 因此达到线性一致性并不困难. 然而, 在分布式环境下, 各个节点可以在很短的时间间隔内先后启动不同的事务, 并且读取各自的时钟. 由于不同的时钟之间有偏移, 甚至时钟的频率也并不完全相同, 因此确定这些事务的先后顺序就变得困难. Google Spanner、OceanBase、TiDB (https://docs.pingcap.com/tidb/stable/tidb-architecture) 均提供线性一致性. Google Spanner 依靠 TrueTime (GPS 和原子钟)[1]获取精确的物理时间以确定事务全局顺序从而达成线性一致性, Paxos[37]和 Raft [38]等基于仲裁的协议使主从之间达成共识. OceanBase 和 TiDB 依靠集中式时间戳分配服务 TSO (timestamp Oracle) 来确定读写操作发生的全局时序从而实现线性一致性. Shamis[39]基于 RDMA (远程内存访问) 提出了一种时钟同步方案以及基于时间戳的强一致事务提交协议 FaRMv2, 满足线性一致性.

Calvin[4]和 Aria[3]将时间以批次 (batch) 为单位进行切分, 所有发生在相同批次内的事务将被收集以生成一个确定的执行顺序, 从而保证所有节点读写操作顺序相同以满足顺序一致性. 区块链系统是一种特殊的多主复制系统, 具有拜占庭容错能力. 达到一致性的关键是如何在去信任的环境下对交易的顺序达成共识, Hyperledger Fabric[30], 利用排序服务来确定交易的全局顺序. 这样区块链就可以实现顺序一致性. CockroachDB[2]依靠混合逻辑时钟 (HLC)[40]可以判断发生间隔超过一个最大时钟偏移量 (默认值为 500 ms) 的两事务的先后顺序从而达成顺序一致性.

顺序一致性一定满足因果一致性. 因为节点内的操作顺序暗含着因果关系, 遵守节点内的操作顺序并达成全局共识意味着所有节点执行的操作顺序都满足因果关系. Lamport 时钟[41]通过逻辑时间, 可以判断不同事件的因果顺序关系; 向量时钟[42]是 Lamport 时钟的改进算法, 每台机器维护一个时间戳, 用于分布式系统中描述事件因果关系. Anna[19]通过向量时钟可以记录一个数据项更新的因果序并解决并发冲突.

最终一致性的本质在于通过放松一致性的要求, 提高分布式系统的可用性和性能. 采用异步或半同步复制的主从数据库如 MySQL、HBase 和 openGauss 在提供高读写性能时, 牺牲了强一致性级别, 满足最终一致性; MySQL Tungsten 和 PostgreSQL BDR 是多主异步复制架构, 他们的每一个主节点都将日志异步复制到其他主节点执行, 只满足最终一致性; 无主复制架构例如 DynamoDB、Riak 和 Cassandra 没有一个节点拥有全副本的最新数据, 每个节点只拥有部分最新数据, 通过读修复和反熵过程使各个副本向最新的数据版本方向变化, 只满足最终一致性.

还有一类问题他们满足 ACI 属性 (associativity 结合性, commutativity 交换性, idempotence 幂等性) 如计数器和购物车 (求并集), 他们输入的前后顺序对结果没有影响, 因此称之为无冲突的复制数据类型 (conflict-free replicated data types, CRDT)[43,44]. 这类问题可以通过无协调分布式方式实现, 满足逻辑单调性的一致性 (consistency as logical monotonicity, CALM)[45], Anna 利用了这个特性, 使用格 (Lattice) 避免冲突, 满足最终一致性.

2.4 多副本并发控制

近年来, 有多个工作对传统并发控制加以改进, 提高了并发控制效率. 例如 Guo 等人[46]对两阶段封锁协议 (two phase locking, 2PL)[47,48]进行了优化, 减少了锁等待时间. Sundial[49]引入了 Cache 机制, 在内存型分布式数据库上表现优秀. CoCo[50]划分事务为 epoch 并以 epoch 为基本单元提交, 同时其使用乐观并发控制 (optimistic concurrency control, OCC)[51,52]变体降低延迟. Eris[53]通过将一部分并发控制功能放入网络中来保证事务的顺序, 由网络原语对事务排序, 同时采用轻量级的事务协议来保证原子性, 从而减少协调开销.

确定性并发控制[54,55]与采用传统并发控制 (2PL、OCC) 不同, 其本身是使用了某种优化策略的重放机制. 其保证事务在各个机器上以一种确定性的顺序执行. 采用确定性并发控制的确定性数据库是采用确定性排序进行事务并发控制的一类数据库. 但确定性数据库在各个节点上的排序执行, 并不是简单的串行执行, 而是可以通过确定性锁排序、冲突依赖检测等策略提高事务并发度, 减少被终止的事务数. 确定性数据库的研究也在快速发展, Aria、Calvin 是最典型的代表, 其中也有基于 Calvin 的云原生数据库 FaunaDB[18]. 确定性数据库通过 Paxos 或 Raft 家族的共识算法保证各个节点间的收到的重放日志保持一致, 进而保证各个节点最终的确定性排序相同. Harding[8]、赵泓尧等人[9]均在一个统一平台实现了各个并发控制算法, 均得到了确定性并发控制性能稳定, 在高冲突场景下适用的结论.

并发控制方法或其变种种类繁多, 但其通常与系统复制架构联系密切. 2PL、基于时间戳排序 (timestamp ordering, TO)[56]等传统的并发控制技术常与多版本并发控制 (multi-version concurrency control, MVCC)[57]结合并可直接应用于主从架构. 因为所有并发写入操作都在主节点本地处理, 同时对读请求较为友好, 例如 openGauss 使用 MV2PL 进行并发控制. 对于分片主从数据库, 跨分片事务需要使用两阶段提交 (2PC) 保证原子性和一致性, 即一般使用 2PC 与传统并发控制技术结合来控制并发, 例如 2PC+MV2PL 或 2PC+MVTO. 对于无主复制, 且使用 KV 存储的数据库, 例如, DynamoDB 和 Cassandra, 它们都使用 last-write-wins 来处理对同一个 key 的多个并发写入, 这种方法简单且快速. 对于多主复制, 存在多种用于并发更新的冲突解决方法. 无协调数据库 Anna 中 lattice 的 ACI 属性可以避免无协调多主复制因副本以不同的顺序观察和处理消息而产生的副本不一致. 在 Bitcoin[28]中, 事务一般是串行执行, 因为一个区块的执行时间在毫秒级, 相比于 10 min 的区块产生时间, 执行部分几乎可以忽略不计. Fabric 使用了可序列化快照隔离 (SSI) 和 OCC, 所有对等节点 peer 基于先前的块快照并行执行其本地事务, 然后基于顺序共识它们, 验证其执行, 并中止冲突事务. MySQL Tungsten 只能主动防止冲突, 而 PostgreSQL BDR 采用后写胜利 (last-write-wins) 来合并冲突. 确定性数据库需要根据预定义的串行顺序执行事务, 这对并发性有限制, 因为操作系统以一种根本上非确定性的方式调度并发运行的线程. 而现有的确定性数据库依赖于依赖图 (例如 Aria) 或有序锁 (例如 Calvin) 来提供更多并行性, 同时确保确定性.

2.5 本节小结

最后, 对本节提到的常见系统进行了总结 (如表 3 所示). Spanner, CockroachDB, TiDB, YugabyteDB, OceanBase 都属于关系型分片主从架构数据库, 它们的不同时钟排序规则使它们达成了不同的一致性级别.

openGauss, MOT, MySQL 属于全副本主从架构数据库, 其中 MOT 为内存数据库. 它们的默认异步复制模式可认为满足最终一致性. MOT 采用乐观并发控制, 而 openGauss 则采用 MV2PL 的可串行化并发控制方式. HBase, DynamoDB, Cassandra 都为 KV 型磁盘数据库, 不同的是 HBase 采用了主从集群架构, 通过 HFile 进行复制, 而 DynamoDB, Cassandra 为无主架构, 通过读修复和反熵进行数据复制保证最终一致性. Anna 是无协调多主内存数据库, 通过格和 ACI 特性实现了冲突合并. Calvin 和 Aria 是关系型确定性多主数据库, 其中 Aira 使用内存存储. 确定性执行的特点使它们具备顺序一致性. 不同的是, Aria 通过依赖图来中止冲突事务, 而 Calvin 通过有序锁进行并发控制. MySQL Tungsten、PostgreSQL BDR、Amazon Aurora 是异步复制多主架构磁盘数据库, 都不具有强一致性, 并主要采用冲突避免和后写胜利的并发策略. Bitcoin 和 Fabric 是多主架构的账本数据库, 复制时都以区块为单位, 分别通过 POW 算法和排序节点达到了顺序一致性, Bitcoin 中的交易串行执行, 而 Fabric 使用 SSI 和 OCC 实现了一个乐观的并发.

表 3　对常见系统复制架构、一致性、并发控制等的总结

系统	数据模型	存储	复制架构	复制单元	排序	一致性	并发
Spanner	SQL	磁盘	分片主从	redo log	TrueTime	线性一致性	MV2PL
CockroachDB	SQL	磁盘	分片主从	binary log	HLC	顺序一致性	MVTO
TiDB	SQL	磁盘	分片主从	binary log	TSO	线性一致性	MV2PL
YugabyteDB	SQL	磁盘	分片主从	redo log	HLC	顺序一致性	MVTO
OceanBase	SQL	磁盘	分片主从	binary log	TSO	线性一致性	MV2PL
openGauss	SQL	磁盘	主从	redo log	TSO	最终一致性	MV2PL
openGauss MOT	SQL	内存	主从	redo log	TSO	最终一致性	OCC
MySQL	SQL	磁盘	主从	binary log	TSO	最终一致性	MVTO
HBase	KV	磁盘	主从	HFile	—	最终一致性	MV2PL
DynamoDB	KV	磁盘	无主	KV	—	最终一致性	last write wins
Cassandra	KV	磁盘	无主	KV	—	最终一致性	last write wins
Anna	KV	内存	多主	KV	commutative	因果一致性	conflict free
MySQL Tungsten	SQL	磁盘	多主	binary log	—	最终一致性	prevent conflict
PostgreSQL BDR	SQL	磁盘	多主	redo log	—	最终一致性	last write wins
Amazon Aurora	SQL	磁盘	多主	binary log	—	最终一致性	prevent conflict
Calvin	SQL	磁盘	多主	batch of SQLs	deterministic	顺序一致性	ordered locks
Aria	SQL	内存	多主	batch of SQLs	deterministic	顺序一致性	dep.graph
Bitcoin	Ledger	磁盘	多主	block of txs	POW	顺序一致性	serial
Fabric	Ledger	磁盘	多主	block of txs	order service	顺序一致性	SSI & OCC

总而言之, 本文将多副本分布式数据库分为以 Spanner 为代表的分片主从架构数据库、以 openGauss 为代表的全副本主从数据库、以 Calvin 为代表的确定性多主数据库、以 MySQL Tungsten 为代表的异步复制多主数据库、以 Anna 为代表的无协调多主数据库和以 Fabric 为代表的多主架构账本数据库.

3 代表性多副本数据库分布式事务处理技术

3.1 分片主从架构数据库

本节主要对 Google Spanner[1]、CockroachDB[2]、TiDB 的复制架构、一致性保证、并发控制进行对比说明.

3.1.1 Google Spanner

从复制架构上看, Spanner 家族采用了分片主从的复制架构. 在 Spanner 中, Spanserver 用来管理分片, 每个 Spanserver 上存放有 100–1 000 个分片 (Tablet). Tablet 有多个副本, 被存放在多个 Spanserver 并位于不同的物理隔离单元 (zone) 中. Spanner 提供了单区域 (region) 和全球多区域的部署. Tablet 的多个副本之间构成主从架构. Spanner 用 Paxos 共识协议管理这一复制组 (Paxos group). Paxos 是一种基于消息传递的分布式一致性算法, 具有一致状态

同步、组成员变更、强容错这些主要特点. Spanner 在确定的某一时刻, 一个复制组只能有一个领导者 (leader). 因此, 对某一分片的写操作必须由该分片的 leader 完成.

Google Spanner 通过 TureTime 实现线性一致性. Spanner 希望每一个服务器获取尽可能高精度的物理时间, 以此来确定各服务器上事务的先后发生顺序. 为了实现高精度的时间同步, 谷歌在每个数据中心内部署了许多时间服务器 (time master), 大部分采用 GPS 获取时间; 少部分则配备了原子钟. 需要获取真实时间的机器将会运行 time slave 程序, 它将使用一个 Marzullo 算法[58]的变种来探测并拒绝欺骗. 最终, TrueTime API 返回的时间是一个形如 [earliest, latest] 的元组, 保证当前真实时间不会早于 earliest, 不会晚于 latest. 客户端对时间的不确定性源于当前计算机 CPU 的时钟漂移, 在间隔 30 s 的投票周期之间, 时钟漂移平均为 4 ms.

Spanner 实现了两种事务, 分别是读写事务 (read-write) 和只读事务 (read-only). 对于一个跨分片的读写事务, Spanner 采用 SS2PL+2PC 来保证事务的隔离性和原子提交, 并需要有一个协调者对该事务负责. 如图 2 所示, 在事务开始时, 客户端会对将要读取数据的分片加读锁并读取数据, 事务内的所有写操作将被缓冲在本地. 当事务执行完毕, 客户端开始发起 2PC 提交. 在 2PC 的第 1 阶段, 客户端将给所有涉及的分片的领导者发送写缓冲, 以及当前事务的协调者信息. 每个非协调者的领导者需要先在本地对将要写的数据加写锁, 并通过 Paxos 把准备提交记录写入日志, 然后将一个最新的、比之前签发给其他事务都要大的时间戳发给协调者. 协调者的过程稍有不同: 需要先给将要写的数据加写锁, 但跳过准备阶段, 接着收集所有参与本事务的领导者发来的时间戳, 随后从这些时间戳以及当前时间的最迟估计中选取一个最大的作为本事务的提交时间戳 (S) 并通过 Paxos 写入一个提交记录. 等待提交时间成为过去的时间常与 Paxos 通信时间重叠. 在提交等待之后, 协调者会发送提交时间戳给所有的客户端和参与的领导者. 每个参与的领导者接收到协调者发来的提交时间戳之后, 通过 Paxos 将事务结果写入日志并释放锁, 并统一使用此时间戳作为当前事务提交的时间. 因此, 发生在提交时间之后的事务都可以读取到当前事务的写, 这就保证了线性一致性.

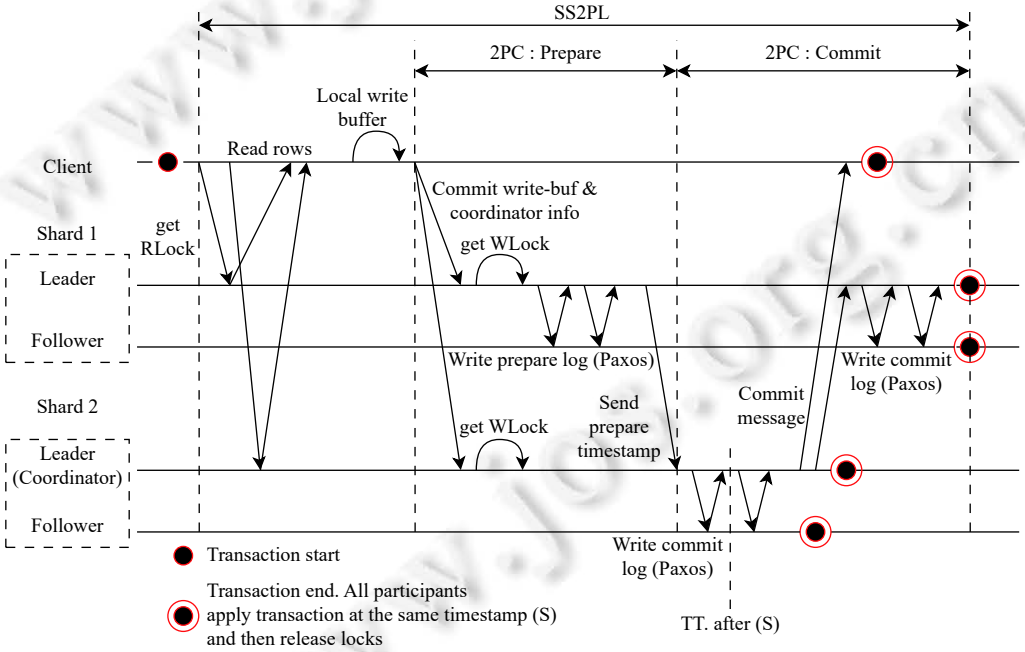


图 2 Google Spanner 跨分片事务处理流程图

对于只读事务, Spanner 不会加锁, 而会选择 TureTime API 返回元组中的 lastest 作为该只读事务的读取时间. 在该只读事务中的所有读操作可以在任何足够新的副本上执行. 同时, Spanner 也提供了基于快照的读, 客户端将提供快照点时间的值或一个时间范围. 对于拥有 TrueTime API 的 Spanner 而言, 分布式的快照读很容易实现.

3.1.2 TiDB

TiDB 中的存储层是 TiKV , KV 键值对按照 key 切分成很多分片 (TiDB 称之为 region). 每个 TiKV 存储多个 region, 每个 region 在不同 TiKV 上有多个副本. 与 Spanner 略有不同的是, TiDB 使用 Raft 同步 region 内的数据. Raft 是 Paxos 的改进版, 比 Paxos 更容易理解, 并且可以完成相同的任务.

TiDB 通过 TSO (timestamp Oracle) 实现线性一致性. TiDB 的思路则是使用集中式的全局授时服务来给所有事务分配全局唯一且单调递增的版本号, 以此确定全局上事务发生的先后顺序, 达到线性一致性级别. 在 TiDB 上, TSO 服务部署在 PD 集群中, 所有的事务在开始时都会向 TSO 获取事务开始时间戳 (start TS). 因此, 事务的执行顺序取决于从 PD 获取时间戳的顺序. TiDB 采用批量为事务分配时间戳的方式节约网络资源. 并且, PD cluster 由 3 个节点组成, 通过 Raft 协议保证容错, PD leader 产生的时间戳严格大于之前产生的任何时间戳. 在 TSO 的保证下, TiDB 可以实现线性一致性. 另外, TiDB 额外支持因果一致性事务, 事务在提交时无需向 PD 获取时间戳, 因此提交延迟更低. 在此一致性级别下, 只有当两个事务加锁或写入的数据有交集时, 才能保证事务的提交顺序与事务的发生顺序保持一致. 目前 TiDB 暂不支持传入数据库外部的因果关系.

TiKV 的事务采用的是 Google 在 BigTable[59]中使用的 Percolator 模型[60], 并默认采用悲观事务模式. TiDB 默认使用快照隔离 (snapshot isolation), 支持读已提交 (read committed). 事务获取 TSO 时间戳的作用之一就是作为快照时间戳, 使其可以读取到一个一致的快照. 在 TiDB 3.0 中, 悲观事务模型被引入进 TiDB. 在最新的 TiDB 5.0 中, 默认开启对于非分布式事务的 Async Commit 和 1PC 提交策略, 使得平均延迟降低了 42%.

3.1.3 CockroachDB

CockroachDB 在复制层实现了与 Spanner 相似的分片主从复制架构, 使用 Raft 管理各个复制集群.

从一致性上来说, Cockroach 既没有使用 TrueTime, 也没有使用集中式的 TSO, 这意味着它无法对齐各个服务器上的时间. 因此, 它不能真正实现线性一致性, 但为了尽可能接近线性一致性, Cockroach 使用混合逻辑时钟 (hybrid logic clock, HLC) 来优化读取性能并且达到顺序一致性. CockroachDB 采用了 MVTO 的并发控制方式, 并实现了可串行化隔离级别. 对于非因果并发事务, HLC 无法判断事务发生的先后顺序, 但可以做到对同一个 key 的操作保持全局时序. 为了达成这一点, CRDB 规定了一个集群最大时间差 $maxClockOffset$, 如果一个事务的开始时间戳为 $t0$, 读取到某数据的写入时间戳 $t1$ 满足 $t0 < t1 < t0 + maxClockOffset$, CockroachDB 并不认为是读到了来自未来的数据, 而是认为因为可控范围内的时钟偏移使得当前事务获取到的时间戳不准确. 此时会引发一个错误, CockroachDB 将把 $t0$ 更新为比 $t1$ 刚好大一点的一个值, 并重启事务, 但最大不确定范围不变, 以防无限制的刷新时间戳.

HLC 并不完美, 它因上文所述的无法对齐时间的问题达不到线性一致性. 无法探测外部因果关系, 这可能引发因果倒置. 考虑 3 个事务 $T1$、$T2$ 和 $T3$, $T2$ 是被 $T1$ 所引发执行, 具有外部因果关系. 但若 $T1$ 和 $T2$ 在短时间内先后在两个服务器上启动, HLC 无法精确分辨 $T1$、$T2$ 发生的先后顺序, 因此 $T3$ 可能会读取到 $T2$ 的执行结果但读不到 $T1$ 的执行结果.

3.2 全副本主从架构数据库

和上述分片主从数据库 Google Spanner、CockroachDB 和 TiDB 不同的是, openGauss 是全副本主从数据库系统. openGauss 是华为开发了新一代开源数据库系统, 支持单机和一主多备部署方式, 主要用于部署区域内数据库集群. openGauss 数据库采用主从复制架构, 主节点将业务请求处理完成后, 才将数据库处理后的最新数据向从副本同步. 在实际场景中, openGauss 的从副本对外通常并不提供服务, 仅用作容灾. 这样的架构使得 openGauss 能够像单机数据库那样处理用户请求, 不必处理多节点时钟同步问题, 也不存在跨节点业务, 因为所有的请求都会转发至主节点进行处理. 然而, 其缺点在于主节点处理压力过大, 难以进行业务分流, 同时从节点不支持读操作, 只作为备份, 所以性能更差.

openGauss MOT (后文简称 MOT) 是一种基于 openGauss 数据库的内存引擎, 两者通过外部数据包装器 (FDW) 进行交互, 和 openGauss 拥有相同的复制架构. MOT 是一种典型的内存优化表, 它针对多核 NUMA 架构进

行了优化. 为了进一步提高事务处理效率, MOT 使用乐观并发控制 (OCC) 作为其事务并发控制算法. MOT 的事务处理分为 3 个阶段: ① 读取阶段: FDW 将 openGauss 解析好的业务请求转换成对 MOT 内存优化表的操作, 每个事务在其私有内存中保存其读集. ② 验证阶段: 事务将其写集加锁, 如果能够成功加锁, 则进入提交阶段, 否则事务将被迫终止. ③ 提交阶段: 验证事务读集, 若事务读集被更改, 则说明其他事务与该数据发生冲突, 该事务应该被终止, 否则事务则提交到数据库. 同时, 主节点周期性地将主节点上的写结果同步到从节点上, 以保证副本一致性和系统的容灾能力. 这样的事务处理机制简洁高效, 易于进行事务处理. 但其缺点也非常明显, 这种事务处理机制也只适用于冲突率低的负载, 在高冲突场景下数据库性能表现不佳.

3.3 确定性多主架构数据库

3.3.1 Aria

从复制架构来看, Aria[3]是多主架构模式. Aria 采用全副本策略放置数据, 在每个节点上存放整个数据库的全副本. 不同于主从架构下的从副本, Aria 的每一个副本都可以处理写操作. 对于每个区域的读写操作, 只需要转发至最近的副本进行处理即可, 大幅减少网络通信开销. 然而, 由于同一事务需要在多个副本上执行, 故 Aria 不可避免存在着冗余执行 (写放大) 问题.

Aria 实现了副本顺序一致性. 如图 3 所示, Aria 是基于批次 (batch) 周期性地进行复制同步的事务处理机制, 即各个主节点将一个 batch 的事务读写集打包并广播给除自己外的其他节点. 同时, 主节点也会收到其他节点发来的事务请求. 当一个 batch 的事务收全之后, 各个节点的排序层根据特定的确定性排序规则 (如根据时间戳排序等) 对收到的事务请求进行确定性排序. 确定性排序的结果是在各个主节点生成顺序一致的事务和操作的执行顺序. 对于相同的副本, 经过相同的事务操作顺序, 各个主节点间将保持最终在 batch 边界时达到强一致性. 从 batch 内部角度来看, 各个副本在一个 batch 内的副本是不能保持一致的; 但从 batch 外部角度看, 各个副本间能够基于 batch 达到强制收敛一致, 即 Aria 实现了基于 batch 的顺序一致性.

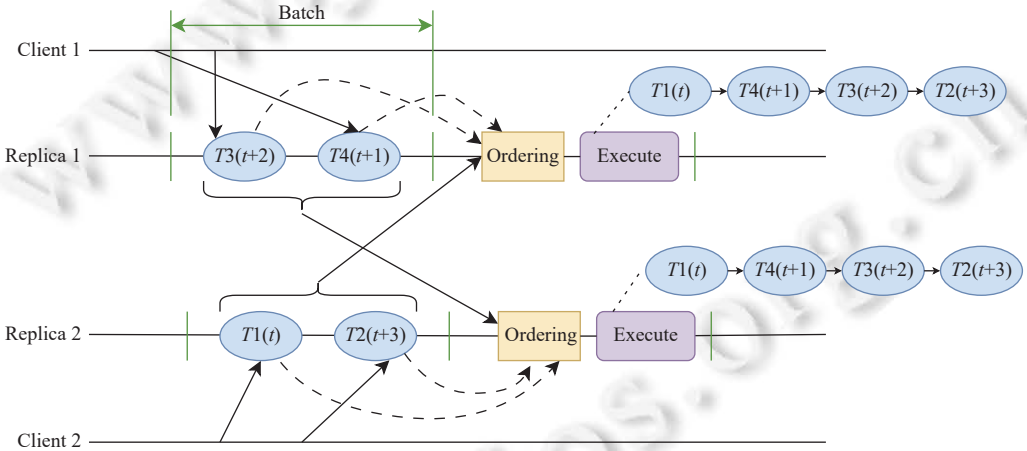


图 3 Aria 事务处理流程图

Aria 的事务执行分为两个阶段: ① 执行阶段: 事务在私有内存中存放读写集, 所有的更改都在私有内存中进行, 不提交给数据库. 同时, 在执行阶段, 还会将写集与全局写集合并在全局写集中保存事务 ID 较小的那个数据记录, 以便在同一批次内保证先写胜利. ② 提交阶段: 在本批次内, 根据冲突检测机制分析事务之间的依赖关系. 如果事务之间存在冲突, 则把该事务延后到下一批次执行, 否则, 将数据提交到数据库中. 因此, 对于每一个副本上的事务来说, 所有事务的操作顺序都是确定性和一致的.

另外, Aria 检测事务本地读写集与全局读写集是否存在 WAW (写后写) 依赖和 RAW (读后写) 依赖. 总体来说, 当本事务与其他事务不存在①WAW 依赖; ②存在 RAW 依赖, 但不存在 WAR (写后读) 依赖, 则事务可以提交. 在同一个批次内, Aria 采用了事务重排序算法, 使得一个批次内, 可以提交更多的事务. 对于该 batch 被终止的

事务, 将采用事务重启机制把冲突的事务推迟到下一批次执行, 避免一部分事务被终止.

3.3.2 Calvin

Calvin[4]属于多主复制架构, 即每个节点都是全副本, 都可以接收读写请求. 在复制架构上, Calvin 和 Aria 采用的复制模式是相同的. 但从复制数据类型的角度看, Calvin 则是对用户发送来的 SQL 请求进行复制. 从架构层次来看, Calvin 除了在各个副本分片外, 还实现了对请求的排序层, 即所有的事务请求都需要经过排序层进行确定性排序后才能开始真正执行. 同时, Calvin 也实现了基于 batch 的顺序一致性.

与 Aria 不同的是, Calvin 的主节点间复制的数据时客户端请求, 而不是事务的读写集. Calvin 在一个 batch 内收集客户端发来的请求, 并把请求通过复制层发送到各个主节点的排序层进行排序. Calvin 对于一个 batch 的并发事务采取确定性锁排序机制进行排序. 由于排序层事先知道并发事务的读写集, Calvin 可以对事务的顺序以及相应事务获取锁的顺序进行确定性排序, 避免事务占用锁的时间过长, 从而提高系统的并发度. 与多主架构类似, 确定性数据库不可避免地存在写放大的问题, 这也是其和多主数据库难以克服的难题. 但确定性数据库也因为能够保证事务全局顺序保证了事务的一致性和全局多副本的一致性, 故而也是分布式数据库的一大发展.

3.4 异步复制多主架构数据库

3.4.1 PostgreSQL BDR

PostgreSQL BDR (bi-directional replication) 是由 2ndQuadrant 开发的多主逻辑复制解决方案. BDR 具有高可用的优点, 同时多主架构可以在任意节点读写的特点使它更加适合地理分布式部署. BDR 默认采用异步逻辑复制, 逻辑复制是基于数据行的复制方式, 它不同于使用块地址逐字节的物理复制方式, 逻辑复制不会复制索引更改从而减少写放大的问题. 异步复制即在本地提交之后再向其他主节点进行复制, 这意味着在某些时间上, 各个主节点存储的数据并不相同.

对于多主复制的架构来说, 如何处理冲突尤为重要. PostgreSQL BDR 使用后写胜利 (last write wins, LWW) 策略来处理冲突: 当在多个节点同时写入造成行级别冲突的时候, 多个节点会根据事务提交的时间戳选择最后写入的那一个. 采用这种策略可能会造成丢失修改, 并且会在不同的主节点看到不一致的状态, 因此不满足强一致性. 而多个副本在一段时间之后一定会被更改为相同的值, 因此 PostgreSQL BDR 满足最终一致性. 为了避免冲突, PostgreSQL BDR 还提供了无冲突数据类型 (CRDT) 和 Eager Replication 两个解决方案, 前者对数据库的业务场景的性质有要求 (如第 2.3.4 节介绍), 后者会带来昂贵的开销.

3.4.2 MySQL Tungsten

Tungsten Replicator 常作为 MySQL 实现多主复制架构的工具, 除此之外, 他甚至可以支持不同数据库之间的复制 (Oracle, MongoDB, Vertica 等). 不同于 PostgreSQL BDR 采用 LWW 的冲突处理方式, MySQL Tungsten 采用 Discipline[61]的方式来预防冲突. Discipline 策略的关键在于授予每个节点不同的权限以避免冲突, 此时每个主节点保存全副本的特性并没有发生改变. 例如, 在一个具有 3 个主节点的 MySQL Tungsten 复制架构中, 每个主节点都有 3 个数据库, 其中每个主节点各持有一个数据库的更新权限和全部数据库的读权限. 在 Tungsten 中, 从主节点发出的每个复制流都是一个单独的服务, 当一个节点收到复制流时可以查看事件的起源, 如果这个事件是由被授权的主节点发起, 那么执行该事件, 否则, Tungsten 将会按用户预先设置好的规则来处理, 如显示错误. MySQL Tungsten 和 PostgreSQL BDR 同样是满足最终一致性的, Tungsten 这种基于 Discipline 的配置虽然可以避免冲突, 但对于进行权限划分数据库管理人员提出了更高的要求, 同时在地理分布式数据库中, 用户可能难免会路由到远端主节点, 造成延迟增大.

3.5 无协调多主架构数据库

Anna[19]是伯克利大学研究的一个分布式键值对存储数据库. 它用格结构 (Lattice) 组合来存储和异步合并状态, 以实现一系列的无协调的一致性级别, 并使用 Actor 模型来构建系统, 是一个具备高度可扩展性、高可用、高吞吐量的无协调分布式数据库.

Anna 的 Actor 使用一致性哈希来划分键的空间, 使用有可调复制因子的多主复制来跨 Actor 复制数据分区.

Actors 定时进行键交换, 将给定 Actor 的键更新传播给副本对应的其他主机. 每个 Actor 的状态都保持在基于 Lattice 的数据结构中, 由于 Lattice 有 ACI 属性 (associativity 结合性, commutativity 交换性, idempotence 幂等性), 这保证了 Actor 的状态在消息延迟、重新排序和重传的情况下保持一致. 例如, 假设有 3 个集合 S_1, S_2, S_3, 对于求并集运算满足: ①交换律: $S_1 \cup S_2 = S_2 \cup S_1$, ②结合律: $(S_1 \cup S_2) \cup S_3 = S_1 \cup (S_2 \cup S_3)$, ③幂等性: $S_1 \cup S_1 \cup S_1 \cup \ldots = S_1$. 因此集合求并集满足 ACI 属性, 分别保证了消息延迟、重新排序和重传情况下的运算结果相同.

具体来说, Anna 的每个副本反复检查来自客户端代理的传入请求, 为这些请求提供服务, 并将结果添加到本地的变更集 changeset, 也就是说该变更集存放的是一段时间内更新的键值对. 这段时间结束时, 每个副本将其变更集中的键更新进行合并, 并将其合并结果多播给负责这些键的相关主机, 并清空变更集. 它还检查从其他副本传来的多播消息, 并将这些消息中键更新合并到其本地状态. 而 Lattice 对合并更新的顺序不敏感, 因此即使各个副本以不同的顺序接收更新, 它们也可以保证副本之间的一致性, 满足最终一致性. 如图 4 所示, 副本 2 (replica 2) 收到来自客户端 2 (client 2) 的写操作请求 $w1$、$w4$, 副本 1 (replica 1) 收到来自客户端 1 (client 1) 的写操作请求 $w2$、$w3$, 此时它们处于冲突状态. 一段时间后 replica 1 和 replica 2 把各自本地的变更集发给对方, 对方分别利用 Lattice 的 ACI 属性进行冲突合并后, 它们最终达到一致的状态.

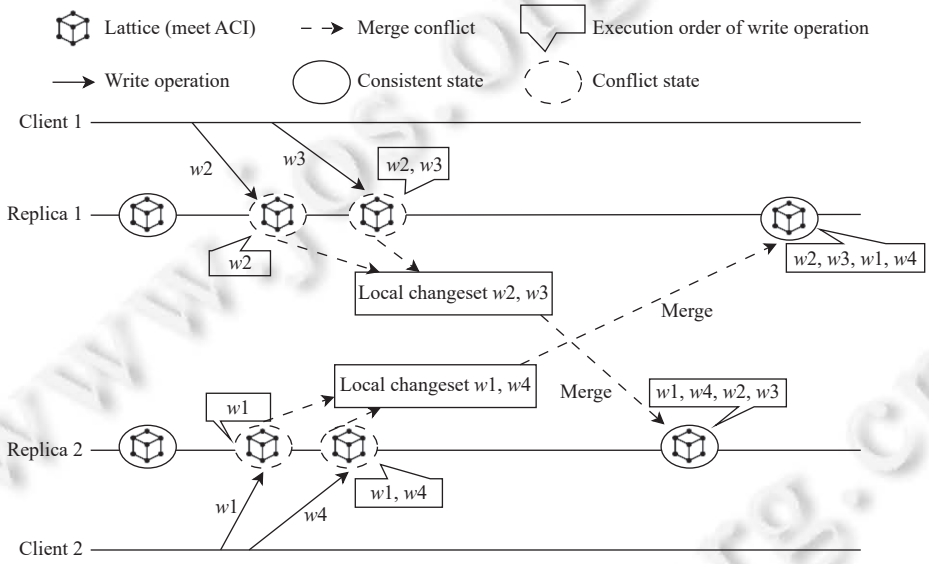


图 4 Anna 事务处理流程图

Anna 使用 Lattice 的方式允许用户自定义冲突解决逻辑并嵌入到 ValueLattice 的合并函数中, 进而可以自定义一致性级别, 在特定场景下极大的提升系统性能. 对于因果一致性, Anna 用 MapLattice (键为客户端代理 ID, 值为与每个代理 ID 关联的版本号) 表示向量时钟; 其中版本号用 MaxIntLattice (元素为整数, 合并函数是在输入和当前元素之间取最大值) 表示, 因此与 MaxIntLattice 关联的整数总是递增的, 可以用来表示单调递增的版本号. 向量时钟和用户定义的 Lattice 构成 PairLattice, 即 (MapLattice, ValueLattice).

当客户端代理执行读-修改-写操作时, 它首先找到当前的向量时钟, 增加与代理 ID 对应的版本号, 并将新向量时钟与更新的对象 (ValueLattice) 一起写入服务器. 然后 PairLattice 的合并函数按字典顺序作用于 PairLattice. 假设有两个 PairLattice $P(a, b)$ 和 $Q(a, b)$, 如果 P 的向量时钟大于 Q 的向量时钟, 那么 $P(a, b)$ 因果跟随 $Q(a, b)$, 结果就是 $P(a, b)$, 反之则为 $Q(a, b)$. 如果 P 和 Q 的向量时钟是不可比较的 (发生冲突), 那么这两对对应于并发写入, 结果合并为 $(P(a) \cup Q(a), P(b) \cup Q(b))$ (冲突合并).

Anna 通过客户端缓冲来支持多种隔离级别. 对于读已提交, Anna 通过在客户端代理上缓冲事务的写操作直到提交来防止脏读, 这确保了未提交的写操作永远不会出现在键值系统中. 和实现因果一致性的 Lattice 组合相比

较, 实现读已提交的 Lattice 组合用表示事务时间戳的 MaxIntLattice 替换了表示向量时钟的 MapLattice. 其 PairLattice 的 merge 函数对 MaxIntLattice (时间戳) 进行比较, 并将 ValueLattice 修改为与较大时间戳对应的 ValueLattice. 如果时间戳相等, 则意味着这些写操作是在同一事务中发出的, 在这种情况下, 将调用 ValueLattice 的合并逻辑. 对于读未提交, 由于不需要防止脏读, Anna 通过禁用客户端缓冲得以实现. 对于可重复读, Anna 缓冲在客户端读取的记录, 当事务试图读取同一记录时, 它调用客户端缓存而非查询服务器. 实现这些隔离级别的 Lattice 组成与实现读已提交的 Lattice 组成相同. Anna 还可以满足单调读、单调写、写跟随读、读自己写等以用户为中心的一致性.

3.6 区块链: 多主架构的账本数据库

区块链本质上也是多副本分布式数据库, 常见的代表有 Bitcoin[28]和 Fabric[30]. 从复制架构上看, 两者都是多主复制架构, 由多个节点执行被共识协议验证过的事务来各自维护一个账本的副本, 并分担读写负载. 但与传统分布式数据库不同的点在于复制的粒度[31]. 一般的多副本分布式数据库由于节点是相互信任的, 复制的具体内容为: 读写操作的有序日志. 因此节点不知道事务逻辑, 一次只看到一个操作. 这种情况要求协调事务执行的事务管理器必须是可信任的. 区块链没有这样的受信任实体, 它的节点中有一些可能是恶意节点, 因此它会复制整个事务, 包含事务上下文、客户端签名、执行时间戳等, 便于执行交易验证.

区块链通过共识机制保证每个副本的一致性. 与一般分布式数据库相比特别的是, 由于区块链节点中有恶意节点, 在共识机制中, 必须使用成本更高的拜占庭式容错 (BFT)[62]的共识协议来保证共识, 维持副本一致性, 如工作量证明 (POW) 和实用拜占庭协议 (PBFT)[63]. Bitcoin 的共识机制主要是使用 POW. POW 协议使得对每一个节点, 诚实地参与到主链的创建中比篡改区块能获得更大的收益. Bitcoin 通过这样的协议一定程度上避免了恶意节点的攻击. 而 Fabric 的共识机制做了插销处理, 可以根据不同业务场景, 支持实用性拜占庭容错 (PBFT)、Raft 等多种协议. 区块链系统达到一致性的关键是如何在去信任的环境下对交易的顺序达成共识, 从本篇所讨论的一致性角度来讲, Fabric 是利用排序服务来确定交易的全局顺序, 实现顺序一致性. 而 Bitcoin 中当一个节点证明了自己的工作量, 将新区块加入链上时, 所有其他节点均按照该节点所排列的事务顺序来执行相应事务, 因此也是顺序一致性.

在部分区块链中, 交易仍然是串行执行的. 例如在 Bitcoin 中, 一个区块的执行时间在毫秒级, 一个区块的产生需要约 10 min 的时间, 所以事务的执行部分的时间几乎可以忽略不计. 在一些支持智能合约的区块链中, 交易之间共享合约的状态, 所以为了保证交易执行结果的确定性, 往往采用顺序执行, 这样执行智能合约的结果一定是确定的, 否则, 可能永远达不到共识. 但 Fabric[64]支持一定程度的并发技术, 使用了可序列化快照隔离 (SSI) 和 OCC, 让所有对等节点 peer 基于先前的块快照并行执行其本地事务, 然后基于顺序共识它们, 验证其执行, 并中止冲突事务. 这样可以实现一个乐观的并行执行, 提高了系统的整体性能和规模, 但是这样的乐观并发模型, 导致了 Fabric 的事务 abort 率很大[64].

另外, Bitcoin 和 Fabric 都不支持分片. 分片作为提高分布式数据库可扩展性的常用技术, 在区块链中实现有几个挑战. 在区块链中实现分片, 必须考虑网络中诚实节点的比例与拜占庭节点的攻击[31], 比如在 POW 中要求总算力的 50% 是诚实的, 而 PBFT 则要求超过 2/3 的节点数是诚实的. 因此分片策略需保证分片后每个分片的绝大多数节点也是诚实的. 在一些支持分片的数据库 (如 Spanner) 中, 跨分片的事务提交需要使用 2PC 来保证原子性, 但 2PC 要求有可信任的协调者, 而区块链需要容忍拜占庭错误, 即可能存在恶意节点, 没有可以充当协调者的受信任实体, 需要引入一些新的协议来统筹跨分片事务来保证原子性[65]. 比如 Ethereum2 引入了一个使用 Casper[66]协议作为共识机制的独立链, 称为信标链, 来协调跨分片事务.

近几年来涌现出了众多基于复制架构的区块链系统. 从事务处理角度分析, 区块链系统的事务处理需要经过排序 (order), 执行 (execute) 和验证 (validate) 几个阶段. 一般来说, 验证阶段是必须要在各个副本上冗余进行的, 而排序和执行阶段可能会在这些系统中进行各种不同的优化. 根据这几个阶段的编排方式, 我们将这些系统归类为以下 4 种架构, 包括 order-execute-validate (OEV), execute-order-validate (EOV), order-execute-parallel-validate (OEPV), execute-validate (EV). (1) 在 OEV 架构下事务都按照相同顺序串行执行, 不支持并发事务, 使用该架构的系统有 Bitcoin, Quorum[67], ResilientDB[68]等. 许可制区块链 Quorum 在以太坊基础上将原有的 PoW 改为基于 Raft

的共识. 每一个节点在执行共识之前将事务组装成块, 然后基于 Raft 共识协议执行共识, 维持一致性. 高性能事务性区块链 ResilientDB 提出了一种地理尺度的拜占庭容错共识协议 (GeoBFT), 节点根据地理位置被划分为多个集群, 每个集群使用 PBFT 对一批本地事务达成共识, 然后与其他集群交换数据, 最终达成共识, 维持一致性. ByShard[69]是一个研究分片拜占庭容错系统的统一框架, 它使用 shard+OEV 的架构, 基于 PBFT 协议实现共识, 并且创新性地提出了一种 orchestrate execute 模型 (OEM) 来处理多分片事务, 在拜占庭式环境中实现了 2PC 和 2PL. 此外, SChain[70]提出了一种可伸缩的 SOEF (order-execute-finalize) 架构, 实质上仍属于 OEV 架构. 块内的事务分布多个执行器上并行执行, 并且支持跨多个块并发执行事务. 克服了逐块执行缺陷, 提高了不同对等点的资源利用率. (2) 在 EOV 架构下, 一组事务会先并行执行, 然后进行排序, 最后用 OCC 检测冲突. 使用 EOV 架构的系统有 Fabric 和 SlimChain[71]等. SlimChain 使用链外存储和并行处理增强系统可扩展性. 它设计了一种链下智能合约执行、链上交易验证和状态承诺的新方案, 并且使用了 PoW 和 Raft 两种共识方法. (3) 在 OEV 架构的基础上, 实现排序阶段和执行阶段并行的架构即为 OEPV 架构, Fabric SSI[72]就使用了这种架构, 事务会被同时发送给排序节点和执行节点, 最后根据排序和执行结果进行验证, 并且使用 SSI 来解决事务冲突. (4) 此外, 许可区块链 NeuChain[73]提出了一种 order-free 的 EV 架构, 它将确定性执行引入了区块链中, 利用确定性排序定义块内事务顺序, 利用 epoch 定义块间事务顺序, 消除了具有单点瓶颈的显式排序步骤, 大大提高了系统的性能.

4 多副本分布式数据库的评测与分析

4.1 实验设置

(1) 集群配置

● 跨区域集群配置: 本文的跨区域实验运行在张家口、深圳和成都的阿里云服务器上. 为了公平起见, 所有的数据库系统都运行在 ecs.r6e.8xlarge, 32 核, 64 GB, CentOS 7.6 服务器上, 与之相对应的测试机运行在 ecs.r6.8xlarge, 32 核, 64 GB, CentOS 7.6 服务器上. 区域间的数据库节点通过公网连接, 公网带宽统一配置为 100 Mb/s, 每个区域的测试机和服务器间通过私网连接, 其带宽大约为 10 Gb/s (见表 4).

表 4 实验配置表

区域	张家口、深圳、成都
服务器配置	ecs.r6e.8xlarge, 32核, 64 GB
测试机配置	ecs.r6.8xlarge, 32核, 64 GB
操作系统	CentOS 7.6 OS
带宽	公网100 Mb/s, 私网10 Gb/s

● 跨 AZ 集群配置: 为了测试数据库在网络条件良好条件下的性能指标, 本文在跨 AZ (availability zone, 即一个地理区域内的多个可用区) 场景下对数据库进行了测试, 数据库系统和测试机分别配置在与跨区域场景下相同的对应配置的服务器上. 为了减少跨 AZ 数据库节点间的网络延迟, 本文中, 数据库节点通过私网连接, 网络带宽为 10 Gb/s. 默认情况下, 采用服务器与测试机相分离的策略, 其网络带宽也为 10 Gb/s, 尽可能避免负载生成过程、请求发送反馈过程对服务器的影响.

(2) 测试负载

● YCSB (Yahoo! Cloud serving benchmark)[74]是一个用 Java 实现的用于服务端的基准测试工具, 包含一个“usertable”表, 表中含有 10 个字段. 值得注意的是, YCSB 可以通过 JDBC 提供对多种数据库的支持. YCSB 提供了多种工作负载, 本文选取了 YCSB-A, YCSB-B, YCSB-C 进行了测试. 其中 YCSB-A 为 50% 读, 50% 写; YCSB-B 为 95% 读, 5% 写; YCSB-C 为 100% 读. 测试过程中, 表数据设置为 100 万, 操作的数量设置为 50 万, 每个区域的连接数默认设置为 256.

● TPC-C[75]是一种针对联机事务处理的规范, 提供了更加复杂的事务并发执行. 本文使用 BenchmarkSQL[76]对上述数据库进行测试, 将 warehouses 的数量设置为 800, 每个区域的连接数默认设置为 200, 其中 new order 和

payment 事务的比例各占 50%, 这意味着本文的 TPC-C 将是全写负载.

(3) 对比系统

● 分片主从架构数据库: 本文选取了 CockroachDB (后文简称 CRDB) 和 TiDB 这两个典型的分片主从数据库系统, 两者都是较为成熟的数据库系统. 本文参照 CRDB 官网给出的配置建议, 将 CRDB 以 9 节点 3 副本的方式部署, 最大程度减少跨区域部署时网络带宽对系统运行的影响. TiDB 也以 9 节点 3 副本的方式部署.

● 全副本主从架构数据库: 全副本主从数据库系统中, 本文选取了 openGauss MOT 进行实验, 由于 MOT 无法通过绑定公网的形式进行跨区域一主两备部署, 在张家口部署了 MOT 的一主两备集群. 测试端的请求则分别从张家口、深圳和成都发送至张家口所在的集群, 其中为了保持实验时服务器和测试机之间的连接方式, 张家口的测试机将通过私网的方式连接到 MOT 集群.

● 多主确定性数据库: 本文选择了 Aria 和 Calvin 两个典型的确定性数据库系统, 因为它们在所采用的确定性并发控制以及同步方式极具代表性. 在测试 Aria 和 Calvin 时, 本文采用了默认的 3 节点配置, 使用了 28 个线程用来处理事务请求, 并预留一部分 CPU 用来生成负载、发送和接收节点间的请求. Aria 和 Calvin 使用一定数量的线程生成负载后直接向系统发送, 执行的过程中, 系统并不会与负载生成的线程进行交互. 因此, 他们的执行机制与 CRDB 等数据库有着本质的区别, 即 CRDB 等数据库时基于数据库驱动的事务交互处理机制, 只有当上一个事务完成之后, 客户端才会发送下一个事务, 而 Aria 和 Calvin 的事务执行并无交互, 且通过测试线程直接向系统注入负载. 值得注意的是, 本文选择在批次大小 (batch size) 为 2 000 的配置下进行 YCSB 测试, 在 batch size 为 500 的配置下进行 TPC-C 测试. 在这样的配置下, 数据库服务器的 CPU 利用率基本可以达到 100%, 即达到机器的充分利用.

● 异步复制多主架构数据库: 本文选择了 PostgreSQL BDR (后文简称 PG-BDR) 作为异步复制多主架构的代表, 在本文实验中, 采用 3 节点配置策略. 使用的是 9.4.12 版本的 PostgreSQL 和 1.0.2 版的 BDR 进行配置.

● 无协调多主架构数据库: 本文选择了 Anna 数据库系统, 其采用的基于无冲突复制数据类型 (CRDT) 的无协调事务处理机制具有一定的代表性. 在本文实验中, 系统默认采用 3 节点 (副本) 配置策略. 特别地, 由于 Anna 不支持事务, 也并不支持标准 YCSB 负载, 所以我们对 Anna 源代码进行了修改, 添加了对事务的支持, 并使其足以支撑 YCSB 测试.

● 区块链数据库系统: 本文选取 Fabric 作为区块链数据库系统的代表. 为了和其他对比系统保持一致, Fabric 也采用 3 节点配置策略.

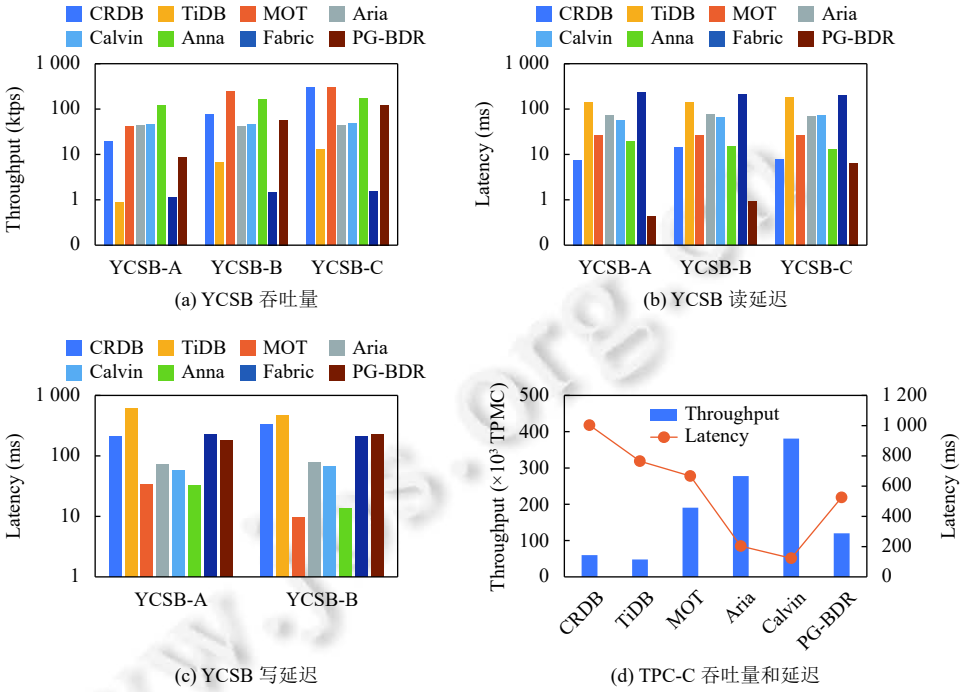
值得注意的是, 由于多副本分布式数据库系统在架构、操作一致性模型和事务并发处理上的差距, 很难将这些系统的公共组件进行抽取. 文献 [57] 用实验方式研究了 MVCC 的多种实现方式. 文献 [77] 研究了现今主流的并发控制算法. 由于研究内容相对集中, 能够很容易抽取不同系统中的公共部分. 由于算法的相似性, 各个数据库的组件大致都相同, 所以只需要对并发控制算法进行重写即可. 然而, 本文的研究对象: 多副本分布式数据库系统, 其在事务处理方式、数据同步策略和系统架构上千差万别, 所以难以提取这些事务处理系统间的公共特性. 因此, 在测试的过程中, 本文并没有开发一个系统模拟各种数据库系统的工作机制. 但是, 我们在测试过程中, 为每个数据库服务器节点选取了相同配置的服务器和与之对应的测试机. 另外, 数据库节点间的网络通信带宽统一为 100 Mb/s, 确保不同系统在测试时的网络条件保持一致. 本文实验统一在 3 副本下进行, 以保证副本数的一致和数据库节点吞吐量相对均衡. 除此之外, 各个系统配置和测试负载也都和表 4 保持一致, 各个系统连接数都相同. 通过以上配置, 尽可能保证了各个系统测试环境的公平性.

4.2 跨域性能对比

为了评估系统在跨区域环境下的性能指标, 本文在跨区域 (张家口、深圳和成都) 场景下分别使用第 4.1 节中提到的负载对数据库进行了测试, 实验结果如图 5 所示.

对于写入密集型的 YCSB-A 负载, TiDB 在本文对比系统中取得了最低的吞吐量 (比其他对比系统低 22.75–136.42 倍). CRDB 表现出了较低的吞吐量, 比 MOT 低 2.08 倍, 比 Aria 低 2.16 倍, 比 Calvin 低 2.38 倍. 对于 CRDB 来说, 这是因为其使用的 2PC 分布式事务提交机制使得事务提交过程中存在多次往返通信, 在跨区域场景下对系统性能的影响更大. Anna 在写操作比例较大的 YCSB-A 负载下有最高的吞吐量, 这是因为其特有的基于

无冲突复制数据类型 (CRDT) 的事务处理机制, 使得事务的冲突合并更轻量级, 与之相对应的事务读写延迟也相对较低. 另外, 典型的区块链系统 Fabric 性能表现较差, 这是因为区块链系统的并发程度普遍不高. PG-BDR 表现出了较好的吞吐量, 随着写比例的降低, 其性能也逐渐变高, 这是由于其读事务在本地直接执行, 但由于异步复制, 在同一时刻各个主节点执行相同读事务结果可能并不相同.



(a) YCSB 吞吐量 (b) YCSB 读延迟

(c) YCSB 写延迟 (d) TPC-C 吞吐量和延迟

图 5 跨域实验结果

如图 5(c) 所示, 确定性数据库写延迟比 CRDB 低 4.35–5.07 倍, 这是因为其只需要一轮网络通信, 大大加快了事务处理能力. 确定性数据库在读密集型负载下的吞吐量却差于 CRDB 和 MOT, 这是因为确定性数据库需要周期性地向各个节点发送读写集, 造成了必要的网络通信开销, 增大读延迟; 而对于 CRDB 和 MOT 来说, 由于读负载的增多, 多个节点间甚至不需要进行通信即可以完成事务. TiDB 在各个负载下均有较高的事务延迟 (写延迟比其他对比系统高 1.66–58.59 倍、读延迟比其他对比系统高 1.82–322.91 倍), 这是因为其 2PC 事务提交机制所决定的. 区块链系统 Fabric 无法将读写延迟区分开, 但其相比其他系统总体延迟较高, 这是由于拜占庭环境下区块链事务处理过程较其他多副本分布式数据库更为复杂, Fabric 中事务处理需要执行、排序、验证几个阶段, 且 Fabric 中事务的顺序串行验证也会导致块在提交前产生堆积, 因此平均的事务延迟会比较高. 多主异步复制数据库 PG-BDR 的写延迟处于较高水平, 这是因为 PG-BDR 采用逻辑复制并需要更多的时间进行冲突的处理. 值得注意的是, 多主异步复制数据库 PG-BDR 和无协调多主数据库 Anna 的读延迟远小于其他数据库系统, 这是其多主架构的优势所在. 因为每个节点都持有整个数据库副本, 所有读事务不需要其他节点参与, 这使得所有的读请求都转化为了区域内的读, 大大降低系统的读延迟, 但由于 Anna 需要代理转发, 所以其读延迟要比 PG-BDR 高 7–19 ms. 另外, 由于 YCSB 类型的负载过于简单, 使得确定性数据库难以发挥其优势.

对于更加复杂类型的负载 TPC-C, 如图 5(d) 所示, 确定性数据库的性能要明显优于 CRDB 和 MOT, 比 CRDB 高 4.63–6.35 倍, 比 MOT 高 1.46–1.99 倍, 比 PG-BDR 高 2.31–3.2 倍, 相应的事务延迟也明显降低. 相应地, MOT 由于集群部署在张家口, 所以主从副本间的网络通信开销基本可以忽略不计. MOT 的 TPC-C 负载延迟主要表现在深圳和成都两地的测试机和张家口间的网络通信开销 (一次往返), 这使得 MOT 的性能也要明显优于 CRDB 和 TiDB, 比 CRDB 高 3.18 倍, 比 TiDB 高 3.98 倍. 对于确定性数据库 Aria 和 Calvin, 他们都表现了较好的性能, 这得益于他们未实现较为庞大的数据库系统的原因, 也取决于使用了确定性排序这种较为新颖的并发控制方式.

更重要的一点是, Aria 和 Calvin 事务执行机制不同, 它们可以在短时间内生成大量负载, 并将负载尽可能注入数据库系统, 这大大提升了数据库的性能. 但它们也存在没有完整 SQL 引擎, 商业应用不成熟等问题. 另外, Aria 展示出比 Calvin 更高的延迟, 这是因为 Aria 通过依赖图来中止冲突事务, 而 Calvin 通过有序锁来实现确定性执行. 值得注意的是, PG-BDR 在 TPC-C 负载下也比 CRDB 和 TiDB 的性能要高, 比 CRDB 高 2 倍, 比 TiDB 高 2.51 倍, 比 MOT、Aria 和 Calvin 低 1.58–3.17 倍. 这是因为 PG-BDR 虽然是多主架构的数据库系统, 但是在全写负载下, 频繁的区域间逻辑复制和较高水平的冲突会消减其多主优势, 同时, 由于异步复制, 在 TPC-C 负载执行完成后, 各个节点仍需要相当长的一段时间处理冲突达到最终一致性.

对于读写延迟, Calvin 和 Aria 各自的读写延迟都十分接近. 而 CRDB 因为允许陈旧读取在读取时延迟很低, 在写入时延迟很高, 这是因为对于跨分片的写入事务, CRDB 多次往返的协调成本十分昂贵. 而 MOT 由于是中心化的单主配置, 无需协调, 因此它的写入延迟很低. 但 MOT 牺牲了高可拓展性, 当数据量过大时, MOT 会不满足业务要求.

4.3 跨 AZ 性能对比

为了评估系统在跨 AZ 环境下的性能指标, 本文在张家口跨 AZ 环境下进行了跨 AZ 实验. 如图 6(a) 和图 6(d) 所示, 确定性数据库 Aria 和 Calvin 的性能要比 CRDB 等数据库高出近两个数量级. 一方面, 这是因为事务执行机制的不同导致的, 即对于 CRDB、TiDB 和 MOT 来说, 测试机与服务器之间通过数据库驱动创建会话保证通信、且一个连接对应于一个数据库会话. 同时, 一个连接一次只能发送一个事务请求, 待数据库向客户端发送回应之后, 才会开启下一个事务. 这意味着, 事务是以请求回应模式注入数据库系统, 导致了数据库系统能够处理事务的能力是有上限的. 然而, 对于 Aria 和 Calvin 来说, 他们是通过简单的负载生成机制, 在短时间内生成大量负载, 并将负载尽可能注入数据库系统, 这大大提升了数据库的性能. 另一方面, CRDB 等数据库需要记录必要的日志信息, 以及大量的磁盘 IO 等工作拖慢了系统的性能. 然而, Aria 和 Calvin 并没复杂的日志系统, 且数据存放于内存中, 减少了磁盘 IO, 大大提高的系统的性能. 另外, 多主异步复制数据库 PG-BDR 比跨域环境下吞吐量提高了 2 倍, 这得益于更加良好的网络条件. 然而, 相比于 MOT 而言, PG-BDR 的每个节点需要逻辑复制增大了网络通信开销, 因此其吞吐量比 MOT 略低, 相应的读延迟和 MOT 相当, 写延迟比 MOT 高 56.2–141.8 ms.

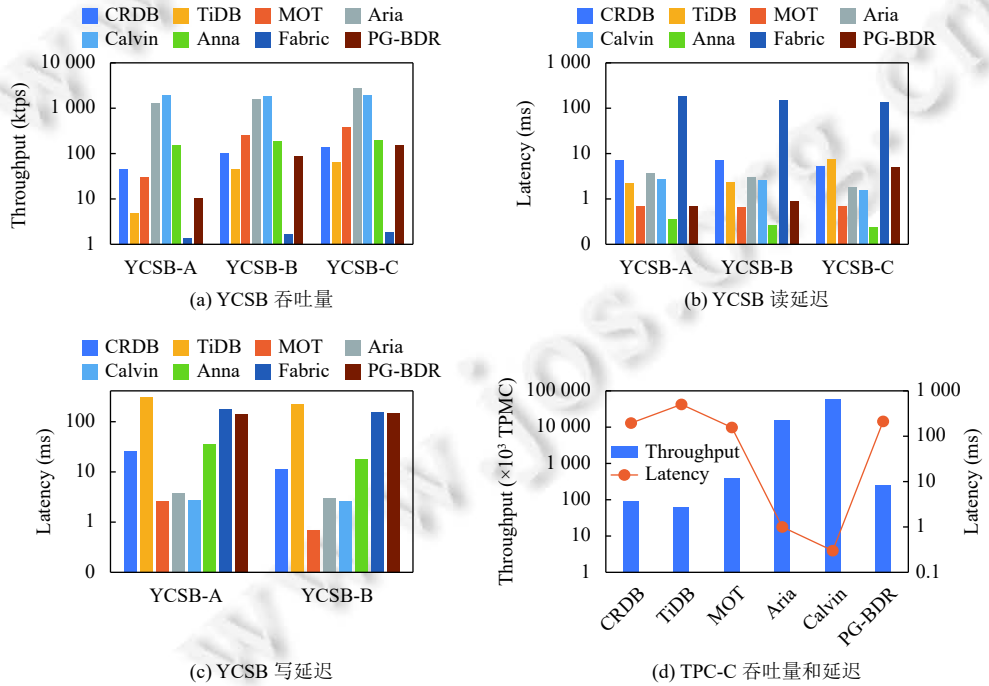


图 6 跨 AZ 实验结果

在写入密集型负载下, Fabric 表现出了最差的吞吐量, 这是由于拜占庭环境下区块链事务处理过程较其他多副本分布式数据库更为复杂. 对于读取密集型事务, CRDB、TiDB、MOT 的性能有了明显的提升, 对于只读事务, TiDB 吞吐量是 CRDB 的 1.92 倍. 确定性数据库 Calvin 吞吐量没有明显提升, 这是因为对于只读事务, Calvin 仍需要周期性地交换读写集. 而 Aria 吞吐量仍有明显提升, 这是因为 Aria 采用分析依赖的手段检测冲突, 只读事务基本没有依赖, 因此不需要分析依赖; 相反, Calvin 基于确定性锁排序的特性使其对于读事务也需要加锁, 因此性能要低于 Aria.

4.4 事务处理时间分解

为了能够更深层次理解各个数据库系统性能指标的差异, 在本节中, 本文在跨区域场景下对 Aria、Calvin 和 CRDB 的 TPC-C 事务执行的各个阶段进行了详尽的分析统计. 如图 7 所示, 所有的数据库都需要进行执行事务和更新数据库操作.

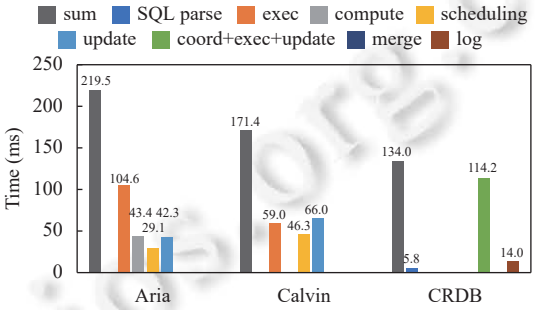


图 7 运行时间分解细目

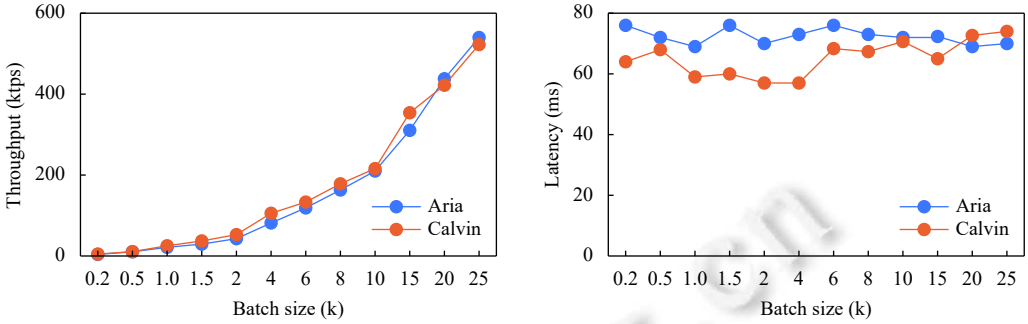
图 7 中, 日志 (log) 和 SQL 解析 (SQL parse) 是 CRDB 所特有的, 也正是因为其成熟的商业数据库系统, 所以它们具有完整的 SQL 引擎和日志机制. 对于 CRDB 来说, 协调 (coord)+执行 (exec)+更新 (update) 紧密耦合在一起, 也即事务执行阶段. 协调和写入紧密耦合在一起, 这也使得 CRDB 的事务执行机制变得复杂, 开销较大. 在 Aria 和 Calvin 中, exec 和 update 阶段分别代表了事务的执行阶段和数据写入阶段. 值得注意的是, Aria 和 Calvin 都因为需要对事务进行确定性排序而都具备了调度阶段, 即图中的 Scheduling 阶段. 然而, 由于 Aria 需要进行事务依赖分析, 即图中的 compute 阶段, 所以 Aria 比 Calvin 多了分析依赖的开销.

与预期的一样, 在 CRDB 中, SQL 解析和写日志过程并不是事务执行过程的瓶颈所在, 其主要的开销在于协调、执行和更新过程, 这也说明了 CRDB 吞吐量较低的原因. 对于事务执行各个阶段花销较为均衡的确定性数据库 Aria 和 Calvin 来说, 其调度开销最高不超过事务总用时的 27%, 这使得数据库避免因协调导致资源浪费, 提高的系统资源利用率. 另外, 也可以很清楚地看到, 确定性数据库的少部分开销用于调度, 大部分时间则用于进行事务处理, 这也能够间接提高数据库的有效性. 另外, 可以看到, 对于确定性数据库, 在跨区域场景下调度并不是影响系统性能的主要因素, 可以看到 Aria 的调度开销是 Calvin 的一半, 但是 Aria 的 TPC-C 性能却要略微低于 Calvin, 这说明调度开销并不是系统性能的决定性因素. 在跨区域场景下, 节点间传递的信息和网络带宽的高低可能是另外的巨大影响因素.

4.5 批次大小对确定性数据库影响

通常, 确定性数据库采用基于 batch 的事务处理机制, 这是为了使得一个 batch 内的事务都能够保证全局一致的顺序, 从而保证事务执行的确定性. 但是, 商业数据库系统如 CRDB 等的事务处理都基于 2PC, 即以单个事务粒度进行数据库事务处理. 因此, 确定性数据库的事务处理方式与其他的数据库系统不同, 故本文对确定性数据库进行了特别研究. 在确定性数据库 Aria 和 Calvin 中, 基于批次处理的事务处理机制使得批次的大小成为系统的关键参数. 为此, 本文使用 YCSB-A 负载, 在跨区域环境下, 变化批次大小 (batch size), 记录系统性能和延迟的变化趋势. 如图 8(a) 所示, 对于 Aria 和 Calvin, 随着 batch size 的增大, 系统的性能逐渐升高. 与此不同的是, 如图 8(b) 所示, 两者的延迟并未随 batch size 的增大有过于明显的变化, 而是在一定区间范围波动. 这是因为两者基于 batch 的

事务处理机制使得事务的延迟主要体现在跨区域通信方面, 即各个节点间写集互相传送的网络通信时间, 因为本地的读写集依赖分析或确定性锁排序都是极快的, 而网络通信时间则占据了大部分的开销. 这也就意味着, 在跨区域环境下, 网络环境的好坏将是直接影响数据库延迟的重要因素.

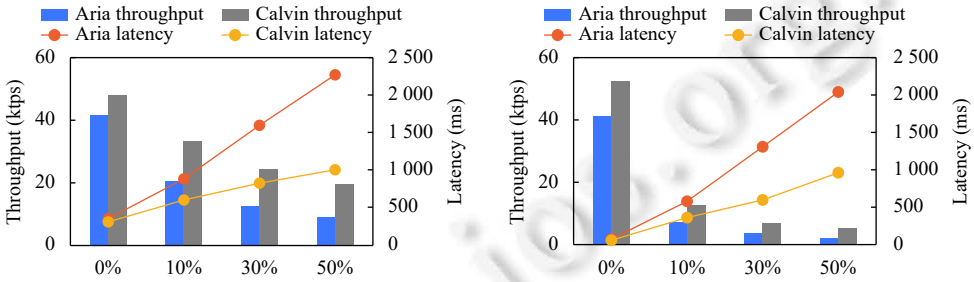


(a) 吞吐量 (b) 延迟

图 8 Batch size 对吞吐量和延迟的影响

4.6 长事务对确定性数据库影响

由于确定性数据库通常采用基于 batch 的事务执行策略, 即总是将事务按照批次进行打包, 并发送至远端节点. 这样的事务处理机制虽然降低了确定性事务处理的复杂度, 但也会受到跨 batch 事务的影响, 因为并不是所有的事务都可以在一个 batch 时间内完成. 然而, 在其他数据库, 如 CRDB、Google Spanner 或是区块链中, 事务的处理都是以单个事务为粒度, 并不存在跨 batch 事务, 因此, 本节我们通过实验研究了长事务对两种典型的确定性数据库的影响. 为了研究长事务对系统性能的影响, 本文在跨区域环境下, 分别按照比例将事务延迟 20 ms 或 100 ms, 相应的实验结果如图 9 所示. 对于每一个事务, 通过随机数的方式按照一定的概率为事务添加延迟, 具体的方式是使得被延迟的事务睡眠相应的时间. 如图 9(a) 所示, 在每个长事务的延迟为 20 ms 的情况下, 随着长事务比例的增加, 系统的吞吐量出现很大程度的下降, 与之对应的延迟也升高, 这都是符合预期的. 这意味着, 对于确定性数据库, 长事务将极大影响系统的性能和延迟, 这也是确定性数据库亟须解决的问题. 同样地, 如图 9(b) 所示, 系统的性能也随着长事务比例的增加而降低得更加明显, 延迟也相应地明显升高. 这说明确定性数据库中, 长事务可能会称为系统性能的瓶颈, 当然, 这也是确定性数据库将来需要解决的问题.



(a) 吞吐量及延迟 (事务延迟=20 ms) (b) 吞吐量及延迟 (事务延迟=100 ms)

图 9 长事务对吞吐量和延迟影响

5 总结与展望

本文首先介绍了多副本分布式数据库的若干关键技术, 包括复制架构、各数据模型下的复制方法、副本一致性和多副本并发策略等, 分析了各种选型下的优缺点, 并对现有典型系统进行了总结分析. 然后本文选取分片主从架构、确定性多主架构、异步复制多主架构、无协调多主架构数据库、区块链系统中的典型系统的分布式事务处理核心技术进行阐述, 从这些系统的一致性保证、并发策略等出发, 对它们的事务处理流程进行了分析与阐述.

最后在阿里云平台上构建了张家口-深圳-成都的跨域分布式集群环境和在张家口区域内的跨可用区 (AZ) 的集群环境, 分别在跨域和跨 AZ 环境下使用 YCSB 和 TPC-C 负载对 CockroachDB、openGauss、TiDB、Calvin、Aria 这几个代表性多副本数据库的事务处理能力进行了性能测试. 从实验结果来看, 确定性多主架构数据库表现出了超越分片主从架构的性能.

基于本文对多副本分布式数据库的分析和评测结果, 目前多副本分布式数据库的事务处理性能, 特别是在跨域场景下的事务处理性能, 主要受限于节点间频繁通信的协调机制: 第一, 多副本之间的强一致性保证往往需要协调机制 (共识) 来实现, 在跨域场景下这将受到巨大挑战; 第二, 事务处理层面, 主流的分片主从架构利用区域感知的分片技术提升了系统的读写性能, 但是数据分片导致了跨分片事务 (即跨节点事务), 需要节点之间的复杂协调机制 (如 2PC) 来保证事务原子性. 目前主流的分布式数据库, 如 Google Spanner、CockroachDB、TiDB 等, 它们所采用分片主从架构恰恰面临这些节点间协调频繁的问题.

在当前全球化趋势下, 众多数据库厂商构建大规模全球跨区域数据库, 频繁协调的性能问题在跨域场景下被进一步放大. 而本文主要测试分析了另一种重要架构, 即以 Calvin 和 Aria 为代表的确定性多主架构数据库, 它基于批量同步交换复制的方法, 采用无协调节点的复制和事务处理机制, 避免了频繁协调的开销. 并且区别于 Anna 等无协调多主架构, 可以提供强一致性和强隔离性的保证, 具有较好的潜力和发展前景. 但是, 基于本文的实验结果和理论分析, 确定性多主架构的发展和实用落地仍然面临诸多问题, 比如长事务和调度开销问题, 另外其缺少可交互性支持, 无法真正应用到实际生产场景中. 关于分片主从架构和确定性多主架构两种架构选型的孰优孰劣, 也值得更多的关注和讨论.

关于多副本数据库的未来发展, 可结合以下几个相关领域技术的最新进展, 不断优化和改进多副本数据库的扩展性、可用性和事务处理性能.

(1) 结合云原生存算分离架构的多副本数据库. 以存算分离架构[78]为特点的云原生数据库技术为分布式数据库系统提供了更大的发展空间. 在存储和计算耦合的架构中, 很难在存储负载和计算负载之间达成比较好的平衡, 原因在于这两种负载对计算机资源的诉求不同. 存储计算分离架构可以让大数据集群充分利用资源, 计算资源和存储资源可以独立地弹性伸缩, 系统负载均衡调度更加灵活, 更符合云计算的特性. 多副本数据库的副本存储与查询计算适合存算分离架构, 研究多写多读的云原生架构[78,79]; 结合存储节点和计算节点的资源特点, 研究计算下推[80]或者缓存优化[81]; 结合持久化云存储的优势, 研究传统事务处理算法在存算分离架构的实现, 如适应存算分离架构的两阶段提交算法[82]等.

(2) 结合新硬件的多副本数据库. GPU 众核并行技术、非易失性存储、高效通信组件 RDMA、可编程的数据处理器 DPU 等新硬件技术的发展极大地推动了数据库的发展, 同样对于多副本数据库的事务处理, 新硬件技术也带来了诸多机遇和挑战. GPU 众核并行技术已经被证明可以提高数据查询效率[83−86], 但是 GPU 对事务处理的支持较差, 主要由于在 GPU 中实现锁管理机制开销较大, 乐观事务处理也需要较多的分支检测, 也包括对索引的更新, 这些从直观上看并不适合 SIMD 计算架构, 若要实现同时能很好支持 AP 和 TP 的 GPU 数据库尚需要进一步研究. 非易失性存储 (NVM) 技术可以作为数据库持久化数据的存储介质, 提高数据存取速度, NVM 是字节可寻址的, 几乎与 DRAM 一样快, NVM 的容量比 DRAM 大很多 (4–16 倍). 这样的 NVM 特性使得构建混合 NVM 和 DRAM 的 OLTP 引擎成为可能, 但是同时也面临 NVM 写入冗余和 NVM 空间管理等挑战, 有许多工作[87−89]正在尝试解决这些问题, 为多副本数据库事务处理设计带来很多新的思路. 远程直接内存访问 (RDMA) 技术可以绕开 TCP/IP 协议栈、操作系统、远端 CPU, 降低了群集内节点的通信开销, 进而可以利用 RDMA 重写分布式事务两阶段提交和乐观并发控制算法[39,90−93], 在多副本数据库上利用 RDMA 实现事务也较为直观, 也较为适合开发生产级别的数据库系统. 数据处理器 DPU 是近年来非常热门的可编程器件, 它由 Nvidia 推出, 包含高性能及软件可编程的多核 CPU、高性能网络接口以及灵活可编程的加速引擎, 是网卡与处理器的结合, 具备传输与计算的能力, 目前尚没有基于 DPU 的事务处理加速的工作, 但我们认为它将对分布式数据库的设计带来重要的影响.

(3) 结合端-边-云协同的多副本数据库. 近年随着边缘计算的兴起, 边缘计算和云计算结合形成了终端-边缘-云 (端边云) 协同的层次型分布式计算模型, 可以在高计算能力服务、高存储能力服务和低延时服务等方面满足

各类应用的需求. 数据库副本的放置策略可结合端-边-云架构的特点, 在不同位置上部署部分或者完整数据库副本, 终端设备的存储和计算能力较弱, 可以配置缓存来维护少量热数据副本支持快速查询; 边缘设备具有一定的存储和计算能力, 可以结合区域-数据感知策略配置部分数据支持近地域的查询处理; 云中心具有近乎无限的存储和计算能力, 部署完整副本甚至多副本来提供计算和备份能力. 已经有工作对端-边-云协同的部分数据查询进行了研究[94−96], 但是考虑事务处理, 最关键的问题是如何维护端-边-云设备上的副本数据一致性, 目前尚缺乏全面深入的研究. 查询处理和事务处理具有本质的冲突, 前者读数据, 后者写数据, 特别是在多副本场景下, 如何在高并发下维护数据一致性将是端-边-云协同数据库要解决的难点问题.

References:

[1] Corbett JC, Dean J, Epstein M, *et al*. Spanner: Google's globally distributed database. ACM Trans. on Computer Systems, 2013, 31(3): 8. [doi: 10.1145/2491245]

[2] Taft R, Sharif I, Matei A, VanBenschoten N, Lewis J, Grieger T, Niemi K, Woods A, Birzin A, Poss R, Bardea P, Ranade A, Darnell B, Gruneir B, Jaffray J, Zhang L, Mattis P. CockroachDB: The resilient geo-distributed SQL database. In: Proc. of the 2020 Int'l Conf. on Management of Data. Portland: ACM, 2020. 1493–1509. [doi: 10.1145/3318464.3386134]

[3] Lu Y, Yu XY, Cao L, Madden S. Aria: A fast and practical deterministic OLTP database. Proc. of the VLDB Endowment, 2020, 13(12): 2047–2060. [doi: 10.14778/3407790.3407808]

[4] Thomson A, Diamond T, Weng SC, Ren K, Shao P, Abadi DJ. Calvin: Fast distributed transactions for partitioned database systems. In: Proc. of the 2012 Int'l Conf. on Management of Data. Scottsdale: ACM, 2012. 1–12. [doi: 10.1145/2213836.2213838]

[5] Yang JP, Plasson N, Gillis G, Talagala N, Sundararaman S. Don't stack your log on my log. In: Proc. of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads. Broomfield: USENIX, 2014. 1–10.

[6] Dillon T, Wu C, Chang E. Cloud computing: Issues and challenges. In: Proc. of the 24th IEEE Int'l Conf. on Advanced Information Networking and Applications. Perth: IEEE, 2010. 27–33. [doi: 10.1109/AINA.2010.187]

[7] Chen K, Zheng WM. Cloud computing: System instances and current research. Ruan Jian Xue Bao/Journal of Software, 2009, 20(5): 1337–1348 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3493.htm [doi: 10.3724/SP.J.1001.2009.03493]

[8] Harding R, Van Aken D, Pavlo A, Stonebraker M. An evaluation of distributed concurrency control. Proc. of the VLDB Endowment, 2017, 10(5): 553–564. [doi: 10.14778/3055540.3055548]

[9] Zhao HY, Zhao ZH, Yang WQ, Lu W, Li HX, Du XY. Experimental study on concurrency control algorithms in in-memory databases. Ruan Jian Xue Bao/Journal of Software, 2022, 33(3): 867−890 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6454.htm [doi: 10.13328/j.cnki.jos.006454]

[10] Hu HQ, Zhou X, Zhu T, Qian WN, Zhou AY. In-memory transaction processing: Efficiency and scalability considerations. Knowledge and Information Systems, 2019, 61(3): 1209–1240. [doi: 10.1007/s10115-019-01340-7]

[11] Souri A, Pashazadeh S, Navin AH. Consistency of data replication protocols in database systems: A review. Int'l Journal on Information Theory, 2014, 3(4): 19–32. [doi: 10.5121/ijit.2014.3402]

[12] Tarun S, Batth RS, Kaur S. A review on fragmentation, allocation and replication in distributed database systems. In: Proc. of the 2019 Int'l Conf. on Computational Intelligence and Knowledge Economy. Dubai: IEEE, 2019. 538–544. [doi: 10.1109/ICCIKE47802.2019.9004233]

[13] Moiz SA, Sailaja P, Venkataswamy G, Pal SN. Database replication: A survey of open source and commercial tools. Int'l Journal of Computer Applications, 2011, 13(6): 1–8. [doi: 10.5120/1788-2469]

[14] Kleppmann M. Designing Data-intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sevastopol: O'Reilly Media Inc., 2017.

[15] Helal AA, Heddaya AA, Bhargava BB. Replication Techniques in Distributed Systems. New York: Springer, 2002.

[16] Bernstein PA, Hadzilacos V, Goodman N. Concurrency Control and Recovery in Database Systems. Boston: Addison-Wesley Longman, 1986.

[17] Yan XA, Yang LG, Zhang HB, Lin XC, Wong B, Salem K, Brecht T. Carousel: Low-latency transaction processing for globally-distributed data. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 231–243. [doi: 10.1145/3183713.3196912]

[18] Freels M. FaunaDB: An architectural overview. 2018. http://fauna-assets.s3.amazonaws.com/public/FaunaDB-Technical-Whitepaper.pdf

[19] Wu CG, Faleiro J, Lin YH, Hellerstein J. Anna: A KVS for any scale. In: Proc. of the 34th IEEE Int'l Conf. on Data Engineering. Paris: IEEE, 2018. 401–412. [doi: 10.1109/ICDE.2018.00044]

[20] Gifford DK. Weighted voting for replicated data. In: Proc. of the 7th ACM Symp. on Operating Systems Principles. Pacific Grove: ACM,

1979. 150–162. [doi: 10.1145/800215.806583]

[21] Sivasubramanian S. Amazon dynamoDB: A seamlessly scalable non-relational database service. In: Proc. of the 2012 Int'l Conf. on Management of Data. Scottsdale: ACM, 2012. 729–730. [doi: 10.1145/2213836.2213945]

[22] Lakshman A, Malik P. Cassandra: A decentralized structured storage system. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35–40. [doi: 10.1145/1773912.1773922]

[23] Jatana N, Puri S, Ahuja M, Kathuria I, Gosain D. A survey and comparison of relational and non-relational database. Int'l Journal of Engineering Research & Technology, 2012, 1(6): 1–5.

[24] Haerder T, Reuter A. Principles of transaction-oriented database recovery. ACM Computing Surveys, 1983, 15(4): 287–317. [doi: 10.1145/289.291]

[25] Gyorödi C, Gyorödi R, Sotoc R. A comparative study of relational and non-relational database models in a Web-based application. Int'l Journal of Advanced Computer Science and Applications, 2015, 6(11): 78–83. [doi: 10.14569/ijacsa.2015.061111]

[26] O'Neil P, Cheng E, Gawlick D, O'Neil E. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996, 33(4): 351–385. [doi: 10.1007/s002360050048]

[27] Mei F, Cao Q, Jiang H, Li JJ. SifrDB: A unified solution for write-optimized key-value stores in large datacenter. In: Proc. of the 9th ACM Symp. on Cloud Computing. Carlsbad: ACM, 2018. 477–489. [doi: 10.1145/3267809.3267829]

[28] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. 2008. https://nakamotoinstitute.org/bitcoin/

[29] Wood G. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper, 2014, 151: 1–32. [doi: 10.1007/978-3-319-75650-9_14]

[30] Androulaki E, Barger A, Bortnikov V, Cachin C, Christidis K, De Caro A, Enyeart D, Ferris C, Laventman G, Manevich Y, Muralidharan S, Murthy C, Nguyen B, Sethi M, Singh G, Smith K, Sorniotti A, Stathakopoulou C, Vukolić M, Cocco SW, Yellick J. Hyperledger fabric: A distributed operating system for permissioned blockchains. In: Proc. of the 13th EuroSys Conf. Porto: ACM, 2018. 30. [doi: 10.1145/3190508.3190538]

[31] Ruan PC, Dinh TTA, Loghin D, Zhang MH, Chen G, Lin Q, Ooi BC. Blockchains vs. distributed databases: Dichotomy and fusion. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 1504–1517. [doi: 10.1145/3448016.3452789]

[32] Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. ACM SIGACT News, 2002, 33(2): 51–59. [doi: 10.1145/564585.564601]

[33] Abadi D. Correctness anomalies under serializable isolation. 2022. http://dbmsmusings.blogspot.com/2019/06/correctness-anomalies-under.html

[34] Abadi D. An explanation of the difference between isolation levels vs. consistency levels. 2019. http://dbmsmusings.blogspot.com/2019/08/an-explanation-of-difference-between.html

[35] Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. on Computers, 1979, C-28(9): 690–691. [doi: 10.1109/TC.1979.1675439]

[36] Ahamad M, Neiger G, Burns JE, Kohli P, Hutto PW. Causal memory: Definitions, implementation, and programming. Distributed Computing, 1995, 9(1): 37–49. [doi: 10.1007/BF01784241]

[37] Lamport L. The part-time parliament. In: Malkhi D, ed. Concurrency: The Works of Leslie Lamport. New York: ACM, 2019. 277–317. [doi: 10.1145/3335772.3335939]

[38] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: Proc. of the 2014 USENIX Conf. on Annual Technical Conf. Philadelphia: USENIX Association, 2014. 305–320. [doi: 10.5555/2643634.2643666]

[39] Shamis A, Renzelmann M, Novakovic S, Chatzopoulos G, Dragojević A, Narayanan D, Castro M. Fast general distributed transactions with opacity. In: Proc. of the 2019 Int'l Conf. on Management of Data. Amsterdam: ACM, 2019. 433–448. [doi: 10.1145/3299869.3300069]

[40] Kulkarni SS, Demirbas M, Madappa D, Avva B, Leone M. Logical physical clocks. In: Proc. of the 18th Int'l Conf. on Principles of Distributed Systems. Cortina d'Ampezzo: Springer, 2014. 17–32. [doi: 10.1007/978-3-319-14472-6_2]

[41] Lamport L. Time, clocks, and the ordering of events in a distributed system. In: Malkhi D, ed. Concurrency: The Works of Leslie Lamport. New York: ACM, 2019. 179–196. [doi: 10.1145/3335772.3335934]

[42] Fidge CJ. Timestamps in message-passing systems that preserve the partial ordering. Australian Computer Science Communications, 1988, 10(1): 56–66.

[43] Shapiro M, Preguiça NM, Baquero C, Zawirski M. Conflict-free replicated data types. In: Proc. of the 13th Symp. on Stabilization, Safety, and Security of Distributed Systems. Grenoble: Springer, 2011. 386–400. [doi: 10.1007/978-3-642-24550-3_29]

[44] Cui YL, Fu G, Zhang YF, Yu G. Elsa: Coordination-free distributed KVS for cross-region architecture. Ruan Jian Xue Bao/Journal of

Software, 2022. 1–19 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6437.htm [doi: 10.13328/j.cnki.jos.006437]

[45] Alvaro P, Conway N, Hellerstein JM, Marczak WR. Consistency analysis in bloom: A CALM and collected approach. In: Proc. of the 5th Biennial Conf. on Innovative Data Systems Research. Asilomar: CIDR, 2011. 249–260.

[46] Guo ZH, Wu K, Yan C, Yu XY. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 658–670. [doi: 10.1145/3448016.3457294]

[47] Eswaran KP, Gray JN, Lorie RA, Traiger IL. The notions of consistency and predicate locks in a database system. Communications of the ACM, 1976, 19(11): 624–633. [doi: 10.1145/360363.360369]

[48] Thomasian A. Concurrency control: Methods, performance, and analysis. ACM Computing Surveys, 1998, 30(1): 70–119. [doi: 10.1145/274440.274443]

[49] Yu XY, Xia Y, Pavlo A, Sanchez D, Rudolph L, Devadas S. Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system. Proc. of the VLDB Endowment, 2018, 11(10): 1289–1302. [doi: 10.14778/3231751.3231763]

[50] Lu Y, Yu XY, Cao L, Madden S. Epoch-based commit and replication in distributed OLTP databases. Proc. of the VLDB Endowment, 2021, 14(5): 743–756. [doi: 10.14778/3446095.3446098]

[51] Kung HT, Robinson JT. On optimistic methods for concurrency control. ACM Trans. on Database Systems, 1981, 6(2): 213–226. [doi: 10.1145/319566.319567]

[52] Thomasian A. Database Concurrency Control: Methods, Performance, and Analysis. Boston: Springer, 1996.

[53] Li JL, Michael E, Ports DRK. Eris: Coordination-free consistent transactions using in-network concurrency control. In: Proc. of the 26th ACM Symp. on Operating Systems Principles. Shanghai: ACM, 2017. 104–120. [doi: 10.1145/3132747.3132751]

[54] Abadi DJ, Faleiro JM. An overview of deterministic database systems. Communications of the ACM, 2018, 61(9): 78–88. [doi: 10.1145/3181853]

[55] Ren K, Thomson A, Abadi DJ. An evaluation of the advantages and disadvantages of deterministic database systems. Proc. of the VLDB Endowment, 2014, 7(10): 821–832. [doi: 10.14778/2732951.2732955]

[56] Bernstein PA, Goodman N. Concurrency control in distributed database systems. ACM Computing Surveys, 1981, 13(2): 185–221. [doi: 10.1145/356842.356846]

[57] Wu YJ, Arulraj J, Lin JX, Xian R, Pavlo A. An empirical evaluation of in-memory multi-version concurrency control. Proc. of the VLDB Endowment, 2017, 10(7): 781–792. [doi: 10.14778/3067421.3067427]

[58] Marzullo KA. Maintaining the time in a distributed system: An example of a loosely-coupled distributed service [Ph.D. Thesis]. Stanford University. 1984.

[59] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: A distributed storage system for structured data. ACM Trans. on Computer Systems, 2008, 26(2): 4. [doi: 10.1145/1365815.1365816]

[60] Peng D, Dabek F. Large-scale incremental processing using distributed transactions and notifications. In: Proc. of the 9th USENIX Symp. on Operating Systems Design and Implementation. Vancouver: USENIX Association, 2010. 251–264.

[61] Giuseppe M. Multi-master data conflicts part 2: Dealing with conflicts. 2013. https://planet.mysql.com/?tag_search=11814

[62] Lamport L, Shostak R, Pease M. The Byzantine generals problem. In: Malkhi D, ed. Concurrency: The Works of Leslie Lamport. New York: ACM, 2019. 203–226. [doi: 10.1145/3335772.3335936]

[63] Castro M, Liskov B. Practical Byzantine fault tolerance. In: Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation. New Orleans: USENIX Association, 1999. 173–186. [doi: 10.5555/296806.296824]

[64] Ruan PC, Loghin D, Ta QT, Zhang MH, Chen G, Ooi BC. A transactional perspective on execute-order-validate blockchains. In: Proc. of the 2020 Int'l Conf. on Management of Data. Portland: ACM, 2020. 543–557. [doi: 10.1145/3318464.3389693]

[65] Dang H, Dinh TTA, Loghin D, Chang EC, Lin Q, Ooi BC. Towards scaling blockchain systems via sharding. In: Proc. of the 2019 Int'l Conf. on Management of Data. Amsterdam: ACM, 2019. 123–140. [doi: 10.1145/3299869.3319889]

[66] Buterin V, Griffith V. Casper the friendly finality gadget. arXiv:1710.09437, 2017.

[67] Morgan JP. Quorum: A permissioned implementation of Ethereum supporting data privacy. 2016. https://github.com/ConsenSys/quorum.

[68] Gupta S, Rahnama S, Hellings J, Sadoghi M. ResilientDB: Global scale resilient blockchain fabric. Proc. of the VLDB Endowment, 2020, 13(6): 868–883. [doi: 10.14778/3380750.3380757]

[69] Hellings J, Sadoghi M. ByShard: Sharding in a byzantine environment. Proc. of the VLDB Endowment, 2021, 14(11): 2230–2243. [doi: 10.14778/3476249.3476275]

[70] Chen ZH, Zhuo HZ, Xu QQ, Qi XD, Zhu CY, Zhang Z, Jin CQ, Zhou AY, Yan Y, Zhang H. SChain: A scalable consortium blockchain exploiting intra- and inter-block concurrency. Proc. of the VLDB Endowment, 2021, 14(12): 2799–2802. [doi: 10.14778/3476311.3476348]

[71] Xu C, Zhang C, Xu JL, Pei J. SlimChain: Scaling blockchain transactions through off-chain storage and parallel processing. Proc. of the VLDB Endowment, 2021, 14(11): 2314–2326. [doi: 10.14778/3476249.3476283]

[72] Nathan S, Govindarajan C, Saraf A, Sethi M, Jayachandran P. Blockchain meets database: Design and implementation of a blockchain relational database. Proc. of the VLDB Endowment, 2019, 12(11): 1539–1552. [doi: 10.14778/3342263.3342632]

[73] Peng ZS, Zhang YF, Xu Q, Liu HX, Gao YX, Li XH, Yu G. NeuChain: A fast permissioned blockchain system with deterministic ordering. Proc. of the VLDB Endowment, 2022, 15(11): 2585–2598. [doi: 10.14778/3551793.3551816]

[74] Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: Proc. of the 1st ACM Symp. on Cloud Computing. Indianapolis: ACM, 2010. 143–154. [doi: 10.1145/1807128.1807152]

[75] TPC-C. 2022. https://www.tpc.org/tpcc/

[76] BenchmarkSQL. 2022. https://github.com/petergeoghegan/benchmarksql

[77] Tanabe T, Hoshino T, Kawashima H, Tatebe O. An analysis of concurrency control protocols for in-memory databases with CCBench. Proc. of the VLDB Endowment, 2020, 13(13): 3531–3544. [doi: 10.14778/3424573.3424575]

[78] Dageville B, Cruanes T, Zukowski M, Antonov V, Avanes A, Bock J, Claybaugh J, Engovatov D, Hentschel M, Huang JS, Lee AW, Motivala A, Munir AQ, Pelley S, Povinec P, Rahn G, Triantafyllis S, Unterbrunner P. The snowflake elastic data warehouse. In: Proc. of the 2016 Int'l Conf. on Management of Data. San Francisco: ACM, 2016. 215–226. [doi: 10.1145/2882903.2903741]

[79] Verbitski A, Gupta A, Saha D, Brahmadesam M, Gupta K, Mittal R, Krishnamurthy S, Maurice S, Kharatishvili T, Bao XF. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In: Proc. of the 2017 Int'l Conf. on Management of Data. Chicago: ACM, 2017. 1041–1052. [doi: 10.1145/3035918.3056101]

[80] Yu XY, Youill M, Woicik M, Ghanem A, Serafini M, Aboulnaga A, Stonebraker M. PushdownDB: Accelerating a DBMS using S3 computation. In: Proc. of the 36th IEEE Int'l Conf. on Data Engineering. Dallas: IEEE, 2020. 1802–1805. [doi: 10.1109/ICDE48307.2020.00174]

[81] Yang YF, Youill M, Woicik M, Liu YZ, Yu XY, Serafini M, Aboulnaga A, Stonebraker M. FlexPushdownDB: Hybrid pushdown and caching in a cloud DBMS. Proc. of the VLDB Endowment, 2021, 14(11): 2101–2113. [doi: 10.14778/3476249.3476265]

[82] Guo ZH, Zeng XY, Wu K, Hwang WC, Ren ZW, Yu XY, Balakrishnan M, Bernstein PA. Cornus: One-phase commit for cloud databases with storage disaggregation. arXiv:2102.10185, 2021.

[83] Bakkum P, Skadron K. Accelerating SQL database operations on a GPU with CUDA. In: Proc. of the 3rd Workshop on General-purpose Computation on Graphics Processing Units. Pittsburgh: ACM, 2010. 94–103. [doi: 10.1145/1735688.1735706]

[84] Breß S, Heimel M, Siegmund N, Bellatreche L, Saake G. GPU-accelerated database systems: Survey and open challenges. In: Hameurlain A, Küng J, Wagner R, Catania B, Guerrini G, Palpanas T, Pokorný J, Vakali A, eds. Trans. on Large-scale Data- and Knowledge-centered Systems XV. Springer. Berlin: Springer, 2014. 1–35. [doi: 10.1007/978-3-662-45761-0_1]

[85] Rosenfeld V, Breß S, Markl V. Query processing on heterogeneous CPU/GPU systems. ACM Computing Surveys, 2022, 55(1): 11. [doi: 10.1145/3485126]

[86] Wang KB, Zhang K, Yuan Y, Ma SY, Lee R, Ding XN, Zhang XD. Concurrent analytical query processing with GPUs. Proc. of the VLDB Endowment, 2014, 7(11): 1011–1022. [doi: 10.14778/2732967.2732976]

[87] DeBrabant J, Arulraj J, Pavlo A, Stonebraker M, Zdonik S, Dulloor SR. A prolegomenon on OLTP database systems for non-volatile memory. Proc. of the VLDB Endowment, 2014, 7(14): 57–63.

[88] Liu G, Chen LY, Chen SM. Zen: A high-throughput log-free OLTP engine for non-volatile main memory. Proc. of the VLDB Endowment, 2021, 14(5): 835–848. [doi: 10.14778/3446095.3446105]

[89] Liu G, Chen LY, Chen SM. Zen+: A robust NUMA-aware OLTP engine optimized for non-volatile main memory. The VLDB Journal, 2023, 32(1): 123–148. [doi: 10.1007/s00778-022-00737-1]

[90] Kalia A, Kaminsky M, Andersen DG. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016. 185–201.

[91] Dragojević A, Narayanan D, Nightingale EB, Renzelmann M, Shamis A, Badam A, Castro M. No compromises: Distributed transactions with consistency, availability, and performance. In: Proc. of the 25th ACM Symp. on Operating Systems Principles. Monterey: ACM, 2015. 54–70. [doi: 10.1145/2815400.2815425]

[92] Wei XD, Dong ZY, Chen R, Chen HB. Deconstructing RDMA-enabled distributed transactions: Hybrid is better. In: Proc. of the 13th USENIX Conf. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018. 233–251. [doi: 10.5555/3291168.3291186]

[93] Wei XD, Xie XT, Chen R, Chen HB, Zang BY. Characterizing and optimizing remote persistent memory with RDMA and NVM. In:

Proc. of the 2021 USENIX Annual Technical Conf. USENIX Association, 2021. 31–45.

[94] Cai ZP, Shi T. Distributed query processing in the edge-assisted IoT data monitoring system. IEEE Internet of Things Journal, 2021, 8(16): 12679–12693. [doi: 10.1109/JIOT.2020.3026988]

[95] Li H, Yi LJ, Tang B, Lu H, Jensen CS. Efficient and error-bounded spatiotemporal quantile monitoring in edge computing environments. Proc. of the VLDB Endowment, 2022, 15(9): 1753–1765. [doi: 10.14778/3538598.3538600]

[96] Velentzas P, Vassilakopoulos M, Corral A. GPU-aided edge computing for processing the *k* nearest-neighbor query on SSD-resident data. Internet of Things, 2021, 15: 100428. [doi: 10.1016/j.iot.2021.100428]

附中文参考文献:

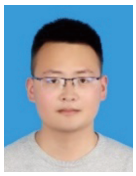
[7] 陈康, 郑纬民. 云计算: 系统实例与研究现状. 软件学报, 2009, 20(5): 1337–1348. http://www.jos.org.cn/1000-9825/3493.htm [doi: 10.3724/SP.J.1001.2009.03493]

[9] 赵泓尧, 赵展浩, 杨皖晴, 卢卫, 李海翔, 杜小勇. 内存数据库并发控制算法的实验研究. 软件学报, 2022, 33(3): 867−890. http://www.jos.org.cn/1000-9825/6454.htm [doi: 10.13328/j.cnki.jos.006454]

[44] 崔玉龙, 付国, 张岩峰, 于戈. Elsa: 一种面向跨区域架构的无协调分布式键值存储系统. 软件学报, 2022. 1–19. http://www.jos.org.cn/1000-9825/6437.htm [doi: 10.13328/j.cnki.jos.006437]

黄纯悦(2000－), 男, 本科生, CCF 学生会员, 主要研究领域为分布式数据库.

罗成(2001－), 女, 本科生, 主要研究领域为大数据处理, 分布式系统.



彭起(1998－), 男, 硕士生, 主要研究领域为数据库, 分布式系统.

张岩峰(1982－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为数据库, 大数据处理, 分布式系统.

张拂晓(2001－), 女, 本科生, 主要研究领域为分布式数据库.

于戈(1962－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为数据库, 分布式系统.

王声溢(2001－), 男, 本科生, 主要研究领域为数据库, 分布式系统.