软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.007281]

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

基于确定性并发控制的云原生多写技术 *

洪殷昊[1,2], 赵泓尧[1,2], 王乙霖[1,2], 史心悦[1,2], 卢卫[1,2], 杨尚[3], 杜胜[3]

[1](数据工程与知识工程教育部重点实验室(中国人民大学),北京 100872)

[2](中国人民大学 信息学院, 北京 100872)

[3](北京人大金仓信息技术有限公司, 北京 100872)

通讯作者: 卢卫, E-mail: lu-wei@ruc.edu.cn

摘 要: 云原生数据库具有开箱即用、弹性伸缩、按需付费等优势, 是目前学术界和工业界的研究热点. 当前, 云原生数据库仅支持"一写多读", 即读写事务集中在单一的读写节点, 只读事务分散到多个只读节点. 将读写事务集中在单一的读写节点, 制约了系统的读写事务处理能力, 难以满足读写密集型业务需求. 为此, 本文提出了D3C(deterministic concurrency control cloud database)架构, 通过设计基于确定性并发控制的云原生数据库事务处理机制来突破一写多读的限制, 支持多个读写节点并发执行读写事务. 其中 D3C 将事务分拆为子事务, 并根据预先确定的全局顺序在各节点独立执行这些子事务, 以满足多个读写节点上事务执行的可串行化. 此外本文提出了基于多版本机制的异步批量数据持久化等机制保证事务处理的性能, 并提出基于一致性点的故障恢复机制实现高可用. 实验表明, D3C 在满足云原生数据库关键需求的同时, 在写密集场景下能达到一写多读性能的 5.1 倍.

关键词: 云原生数据库;确定性并发控制;事务处理

中图法分类号: TP311

中文引用格式: 洪殷昊,赵泓尧,王乙霖,史心悦,卢卫,杨尚,杜胜.基于确定性并发控制的云原生多写技术.软件学报. http://www.jos.org.cn/1000-9825/7281.htm

英文引用格式: Hong YH, Zhao HY, Wang YL, Shi XY, Lu W, Yang S, Du S. Deterministic Concurrency Control based Multi-Write Transaction Processing over Cloud-native Databases. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7281.htm

Deterministic Concurrency Control based Multi-Write Transaction Processing over Cloud-native Databases

HONG Yin-Hao[1,2], ZHAO Hong-Yao[1,2], WANG Yin-Lin[1,2], SHI Xin-Yue[1,2], LU Wei[1,2], YANG Shang[3], DU Sheng[3]

[1](Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), Ministry of Education, Beijing 100872, China)

[2](School of Information, Renmin University of China, Beijing 100872, China)

[3](Beijing Kingbase Technology Inc., Beijing 100872, China)

Abstract: Cloud-native databases have emerged as a hot topic in the field of database development in the era of cloud computing, thanks to their advantages such as out-of-the-box functionality, elastic scalability, and pay-as-you-go pricing. However, mainstream cloud-native databases only support a single master node to execute write transactions. This limitation hampers the system's ability to handle write-intensive workloads, making it difficult to meet the demands of businesses with high write requirements. To address this issue, this paper proposes the D3C (deterministic concurrency control cloud database) architecture, which achieves cloud-native multi-writer capabilities by designing a transaction processing mechanism based on deterministic concurrency control. D3C splits transactions into sub-transactions and independently executes them on various nodes according to a pre-defined global order, ensuring serializable isolation for transaction execution on multiple read-write nodes. Additionally, this paper introduces mechanisms such as asynchronous batch data

* 基金项目:国家自然科学基金(61972403, 61732014);
收稿时间: 2024-05-27; 修改时间: 2024-07-16, 2024-08-19; 采用时间: 2024-08-29; jos 在线出版时间: 2024-09-13

persistence mechanisms based on multi-version to ensure the performance of multi-writer transaction processing, and proposes a consistency point-based fault recovery mechanism to achieve high availability. Experiments have shown that D3C can achieve 5.1 times the throughput of a traditional single-master architecture in write-intensive scenarios, while meeting the key requirements of cloud-native databases.

Key words: cloud-native databases; deterministic concurrency control; transaction processing

云原生数据库具有开箱即用、弹性伸缩、按需付费的特点, 是学术界和工业界最为关心的研究热点之一. 报告显示[1], 2022 年数据库系统全球整体的市场规模约为 653 亿美元, 其中云数据库服务规模达到了 135 亿美元, 约占 20.7%[2]. 在国内,2022 年云数据库规模约占国内数据库系统市场规模的 54.3%, 2023 年占比进一步扩大达到 59.8%[3]. 在工业界,互联网巨头如亚马逊、微软、谷歌以及国内的华为、阿里等公司都已推出多款诸如 Aurora[4]、PolarDB serverless[5]的云原生数据库(cloud-native database)产品, 市场占有率正在逐步赶超传统数据库巨头.

当前主流的云原生数据库[4-8]采用一写多读的架构. 在该架构中,读写事务集中在读写节点, 只读事务分散到多个只读节点, 实现读写分离. 一方面, 可以通过增加只读节点的数量来提升只读事务的吞吐; 另一方面, 当读写节点故障时, 只读节点可以快速替换读写节点, 保障服务的可用性. 虽然一写多读具有诸多优点, 但单一的读写节点限制了系统的读写事务处理能力, 使得系统难以应对电商、金融等行业中以读写事务为主的业务负载. 突破云原生数据库一写多读的限制, 支持多个节点同时执行读写事务从而实现"多写", 是当前云原生数据库系统的一项关键技术.

多写, 即在云原生数据库系统中配置多个可以执行读写事务的读写节点, 通过增加读写节点的数量, 可以达到比单个读写节点更高的读写事务吞吐量. 理想上, 系统中的每一个节点都可以支持读写事务, 而不仅仅像一写多读中只有一个读写节点可以支持读写事务. 实现多写的关键技术难点在于: 不同节点如何检测和处理事务冲突以满足隔离级别要求(本文以冲突可串行化作为隔离级别), 如何确保故障下的事务正确性, 以及如何获得良好的事务性能.

目前, 主要有两种方案来实现多写.

- 方案一: 基于两阶段提交协议(two-phase commit protocol, 2PC)[9]的多写技术. 该技术通过将数据拆分为多个数据分片, 并由不同的节点处理不同数据分片上的事务冲突来做到每个节点独立处理事务. 为此, 当一个读写事务需要访问多个数据分片上的数据时, 这个事务会被划分为多个子事务, 并交由不同节点来处理对应数据分片上的冲突. 因此同一个事务的不同子事务在不同节点上的执行结果可能不同, 需要 2PC 来协调这些子事务的执行结果, 从而保证事务的原子性. 当节点发生故障时, 该机制通过事务日志确保已经提交的事务的持久化, 从而能够通过日志回放保证节点故障前后的一致性.
- 方案二: 基于远程内存直接访问技术(remote direct memory access, RDMA)的多写技术[10-16]. 该技术要求在低延迟高带宽的数据中心内网络中, 将并发控制的元数据维护在专门的节点内存当中, 并引入 RDMA 技术来跨节点访问该内存. 因此每个节点可以以类似于单机事务处理的方式, 直接通过 RDMA 用同一个并发控制的元数据来检测和处理冲突. 低延迟高带宽的网络环境使得系统更容易获得较高的事务性能. 该技术通过持久化内存(persistent memory, PM)以及多机备份的方式, 保证节点故障时数据和事务状态不会丢失.

上述两个方案在性能或适用性上具有明显的限制. 方案一需要利用两阶段提交协议来协调同一个事务在不同节点上的子事务来保证整个事务的原子性, 由于需要至少两轮的网络开销, 与单机事务相比, 事务性能下降显著; 方案二则需要利用 RDMA 特殊设备. 但是 RDMA 设备只能在单个数据中心内使用, 无法部署在跨数据中心网络, 因此无法做到数据中心、城市级别的故障恢复, 无法保证复杂故障场景下的事务正确性. 所以, 现如今并不存在一个能够同时解决以上三个技术难点的多写方案.

确定性并发控制具有和基于 2PC 类似事务处理方法，将数据拆分为多个数据分片，并由不同的节点负责

处理，然而确定性并发控制无须使用 2PC，因而能够消除事务提交时的网络开销。如图 1 所示，在分布式数据库中，确定性并发控制的事务吞吐量始终高于基于 2PC 的并发控制的事务吞吐。此外，确定性并发控制无须使用 RDMA 设备。没有跨数据中心的使用限制。

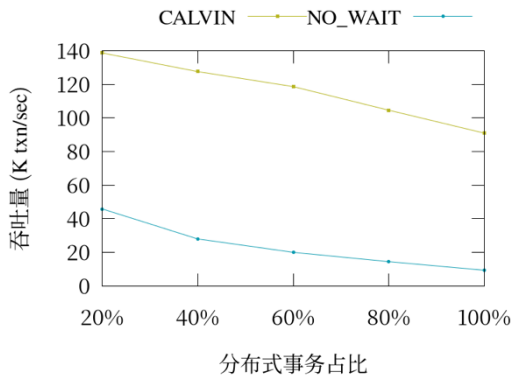


图 1 分布式数据库场景下确定性并发控制和基于 2PC 的并发控制性能对比

本文提出了一个基于确定性并发控制的云原生数据库架构(deterministic concurrency control cloud database, D3C). 通过设计基于确定性并发控制的云原生数据库事务处理机制, 令各个读写节点按照预先确定的事务顺序独立地处理不同数据分片上事务之间的冲突, 保证了所有读写节点上事务都按照同一个顺序执行和提交, 从而满足可串行化隔离级别的要求. 具体来说, 在 D3C 中, 事务的处理遵循以下流程: 首先, 读写节点会收集来自客户端的所有事务请求, 并根据它们的到达顺序为其排序、打包成批. 接下来, 读写节点将这一批事务的逻辑日志写入存储. 之后, 这些事务会根据数据分片被拆分为多个子事务并发送到对应的读写节点上, 然后读写节点则按照事先排定的顺序各自独立地执行这些事务的子事务.

传统的云原生数据库采纳"日志即数据"的思想, 即在事务执行结束时只持久化日志来保证事务的持久性[17,18], 而不会同步地将数据写入存储, 从而保证了事务执行效率. 然而, 为了保证事务的持久性和事务执行的确定性, D3C 需要在事务执行之前记录逻辑日志. 由于逻辑日志仅记录事务的执行逻辑, 并且云原生数据库架构中的存储缺少足够的计算资源来回放逻辑日志, 因此事务需要额外将其执行结果同步写入存储. 这一机制会严重影响事务执行性能. 为此, 本文提出了基于多版本的异步批量数据持久化机制来实现异步的数据写入, 从而保证良好的事务执行效率. 此外, 本文还设计了基于多调度器的读写事务优化和基于数据一致性点的只读事务优化, 进一步提升 D3C 的性能.

由于本文采用了新的云原生数据库架构, 并设计了基于确定性并发控制的云原生数据库事务处理机制. 原有确定性并发控制和云原生数据库的高可用方案都不再能够使用. 确定性并发控制算法通常会为每个读写节点配备多个副本节点, 并且在每个副本上执行与读写节点相同的事务来保证副本间数据的一致性. 然而这一方案在云原生数据库下会显著增加成本. 如果使用云原生数据库原本的方法, 则需要数据库冗余写入事务的逻辑日志和物理日志, 显著增加存储节点的压力. 本文因此设计了基于一致性点的故障恢复机制, 通过识别数据一致性点和日志一致性点来确定故障恢复的起始点和终点, 根据起始点和终点来恢复故障的节点, 从而确保节点故障下的事务正确性.

总体而言, 本文的主要贡献包括:

- 提出了基于确定性并发控制的云原生数据库架构 D3C, 并在架构中提出了基于确定性并发控制的云原生数据库事务处理机制, 实现了云原生数据库的多写机制.
- 提出了基于多版本的异步批量数据持久化机制来解决 D3C 中实现高效的数据持久化, 并设计了基于多调度器的读写事务优化和基于数据一致性点的只读事务优化两种优化, 进一步提升 D3C 中事务处理的性能.

- 提出了基于一致性点的故障恢复机制来实现 D3C 架构中各类节点的故障恢复, 保证系统的高可用.
- 对 D3C 以及基于确定性并发控制的云原生数据库事务处理机制进行了充分的实验, 实验结论表明 D3C 明显优于传统的一写多读方案, 在写密集场景下, 性能能达到一写多读方案的 5.1 倍.

本文第 1 节介绍了云原生数据库和确定性并发控制的研究现状. 第 2 节介绍了本文所需的基础知识. 第 3 节介绍云原生多写事务处理架构 D3C. 第 4 节介绍 D3C 下的三种事务优化策略, 基于多版本的异步批量数据持久化机制, 基于数据一致性点的只读事务优化和基于多调度器的读写事务优化. 第 5 节介绍了 D3C 中基于一致性点的故障恢复机制. 第 6 节介绍本文的系统实现, 第 7 节通过对比实验验证了 D3C 的有效性和高性能. 最后总结全文.

1 相关工作

1.1 云原生数据库

近些年来, 云服务技术的普及和发展推动了数据库系统上云的进程. 为了充分利用云计算环境中不同服务独立伸缩的能力, 云原生数据库通过采用计算存储分离(简称存算分离)的架构, 实现计算资源, 存储资源的独立, 从而为数据密集型应用提供高性能、高可用和弹性伸缩的能力, 提高应用的资源利用率并降低成本.

Aurora[4]是首个提出基于存算分离架构的云原生数据库, 包括计算节点和存储节点两类节点. 计算节点基于云计算服务, 负责数据库中事务处理、查询处理等计算密集型工作, 存储节点基于云存储服务, 负责数据库中日志存储、数据持久化、数据备份等存储密集型工作. 此外, Aurora 遵循了"日志即数据"的原则, 将事务的物理日志写入到存储节点来完成数据的持久化. 脏页面不会写入存储节点, 减少了计算节点和存储节点之间页面传输的开销. Socrates[6]、Taurus[7]等数据库在存算分离架构的基础上, 进一步将存储节点划分为日志存储节点和页面存储节点. 日志存储节点负责持久化日志, 页面存储节点负责回放日志为计算节点提供页面访问服务. 这种架构进一步解耦资源, 提升系统弹性. PolarDB[8]则在此基础上, 提出了智能化回放日志的策略来加速日志回放以及并行 Raft 机制保证数据一致性. PolarDB Serverless[5]在存算分离的架构基础上引入了共享缓存层, 以提升计算节点访问数据页面的效率以及计算节点之间数据同步的效率. 计算节点的缓存未命中时可以首先从共享缓存层读取页面, 降低了页面读取的时延. 同时由于多个计算节点共同访问缓存节点, 可以避免多个计算节点访问相同数据页面时需要重复从存储节点请求相同数据页面的额外开销.

此外, 业界有三个工作提出了支持多写操作的云原生数据库架构设计. 其中, ScaleStore[19]通过融合 DRAM 缓存、NVMe 高速存储以及 RDMA 技术, 构建了一个高效的分布式存储引擎, 显著提升了数据处理能力. 而 PolarDB-MP[20]则采用多个计算节点架构, 利用 RDMA 技术直接访问内存层中的锁信息, 有效管理了跨节点事务的并发冲突, 确保了数据一致性和高并发性能. 然而, 这两项工作均高度依赖 RDMA 技术, 这一局限性阻碍了它们在跨数据中心网络环境中的应用, 限制了数据中心间乃至城市级、区域级的故障恢复能力, 不适于跨地区部署. 相比之下, Taurus-MM[21]则通过多个计算节点与一个中央的锁管理节点进行交互, 实现了不同节点上事务冲突的协调与管理. 尽管无需引入 RDMA, 但单一锁管理器的设计也引入了新的挑战: 一方面, 它可能成为系统性能的单点瓶颈; 另一方面, 在跨数据中心部署时, 所有计算节点均需与远程数据中心的锁管理器通信以获取锁资源, 这不可避免地增加了网络通信的复杂性和开销, 对系统整体性能和可扩展性构成了潜在影响.

1.2 确定性并发控制算法

与非确定性并发控制[22-31]随机调度事务的做法不同, 确定性并发控制使用确定的顺序来调度事务. 算法的核心是将事务的并发控制调度和事务的具体执行解耦开来, 从而将事务处理分为对应的两个阶段, 事务调度阶段和事务执行阶段. 根据事务调度阶段和事务执行阶段的前后顺序不同, 确定性并发控制算法可以被分为两类. 一类较为悲观, 先执行事务调度后执行事务逻辑[32-35], 本文将其称为悲观确定性并发控制算法; 另一类较为乐观, 先执行事务逻辑后执行事务调度[36-39], 本文将其称为乐观确定性并发控制算法.

悲观确定性并发控制算法首先对事务进行定序, 并按照确定的顺序进行事务调度. Calvin[32]使用了类似两阶段加锁机制(two-phase locking protocol, 2PL)[22]的锁机制来实现事务调度. 其区别是 Calvin 是按照定序后的事务顺序对事务依次进行加锁的有序锁, 而 2PL 是事务随机调度加锁的抢占锁. 然而, 一些工作[38,40]已经表明, Calvin 的单线程加锁产生了明显的性能瓶颈. BOHM[33]则提出了单机多版本的确定性并发控制. 然而存在单一线程在数据分片创建数据项版本的限制. Caracal[34]去除了 BOHM 的限制, 设计了多线程并发创建版本的算法, 同时设计了确保执行线程负载均衡的事务拆分方法, 减少事务整体延迟. 然而, Caracal 要求一批事务都执行完事务调度阶段才能进入事务执行阶段增加了事务的延迟. PWV[35]为悲观确定性并发控制提出了提前写入可见性. 通常, 确定性并发控制要求事务的写入在事务完成时才可见. 该方法指出在满足提前写入可见性前提的情况下, 确定性事务可以通过读未提交的数据提高系统的事务吞吐量.

乐观确定性并发控制算法也首先对事务进行定序, 然后事务将按任意的顺序完成执行阶段, 并随后执行事务调度确保确定性顺序. Senthil Nathan 等人的工作[36]在数据库中在区块链中实现了乐观确定性并发控制. 该工作使用可串行化快照隔离(serializable snapshot isolation, SSI)[30,31]在事务提交前按事先确定的顺序分析事务依赖, 并在检测到依赖环时按规则确定性地回滚其中的某个事务. DOCC[37]同样按事先确定的顺序进入事务调度阶段. 该方法验证事务读到的数据项是否是期望的版本, 并在发现事务执行顺序错误时立即重做事务. 在 Aria[38]中, 事务按批进入事务调度阶段, 并发地执行事务依赖验证. 通过重新设计的确定性提交协议, 该工作能够允许事务不按照事先确定的顺序提交, 同时仍然保证提交顺序是具有确定性的. Harmony[39]优化了 Aria 重排序事务的算法, 减少了事务的回滚率, 并能在基于磁盘的区块链环境下取得更好的性能.

悲观确定性并发控制算法的优点是所有事务都首先经过了事务调度阶段, 因此在事务执行阶段, 事务不会因为冲突回滚, 无须提交协议保证事务的原子性. 其缺点是首先执行事务调度阶段限制了事务的并发度, 导致在低冲突的场景下事务吞吐量较低. 乐观确定性并发控制算法的优点是事务执行阶段在前, 事务可以不受任何限制地并发执行. 其缺点是由于事务执行阶段可能没有按照预先确定的顺序读到正确的数据项版本, 因此需要类 2PC 的确定性提交协议来确保或确定性地调整事务顺序. Ziliang Lai 等人[39]指出乐观确定性并发控制算法的性能瓶颈在确定性提交协议上, 因而更容易受到高冲突的负载影响.

确定性并发控制在数据库[32-35,37,38]和区块链[36,39,41,42]上都能够得到应用. 其中 SChain[41,42]在区块链上提出了与本文类似的存算分离架构. 其区别是, SChain 针对的是许可区块链, 对于其中的每个节点, 都只有部分其他节点可以信任. 因此, SChain 将确定性并发控制的分布式执行改为了单机执行, 避免执行流程依赖不可信任的节点. 本文则利用确定性并发控制来解决云原生数据库的多写问题, 并在云原生数据库的架构下优化确定性并发控制的事务吞吐量, 以及根据云原生数据库的特点来设计新的确定性并发控制的高可用方案.

讨论 确定性并发控制当前在分布式数据库中应用较少, 本文认为主要是以下三个原因: 1. 现有分布式数据库与确定性并发控制的执行架构完全不同, 这使得现有系统难以改造以引入确定性并发控制; 2. 确定性并发控制的高可用方案需要冗余的副本; 3. 确定性并发控制存在事务已知读写集的前提假设. 对于第一个问题, 多写云原生数据库是一个新兴的领域, 针对确定性并发控制的架构和特点, 构建新的云原生数据库中存在更大的可能性. 对于第二个问题, 云原生数据库存算分离的特点使得可以在只为数据提供副本的情况下实现高可用, 本文在第 5 节中基于云原生数据库的存算分离提出了不需要冗余读写节点的基于一致性点的故障恢复机制. 对于第三个问题, 确定性并发控制的前提假设已有乐观锁位置预测(Optimistic lock location prediction)[32]的方法, 在事务执行前侦察事务的读写集.

2 预备知识

这一节主要介绍云原生数据库本身的事务处理机制, 并介绍 Calvin 确定性并发控制算法在事务处理上与云原生数据库的异同.

2.1 云原生数据库中的事务处理机制

图 2 展示了云原生数据库存算分离的典型架构. 系统包含负载均衡器、计算层和存储层. 计算层包含一

个读写节点和多个只读节点. 负载均衡器会将读写事务发送给读写节点, 将只读事务发送给只读节点或读写节点. 存储层存储了日志和数据. 数据页面通过日志回放的方式生成. 为了保证数据库的持久性和可用性, 日志和数据通常均采用多副本机制进行备份, 副本之间使用 Paxos 等一致性协议来保证数据一致性.

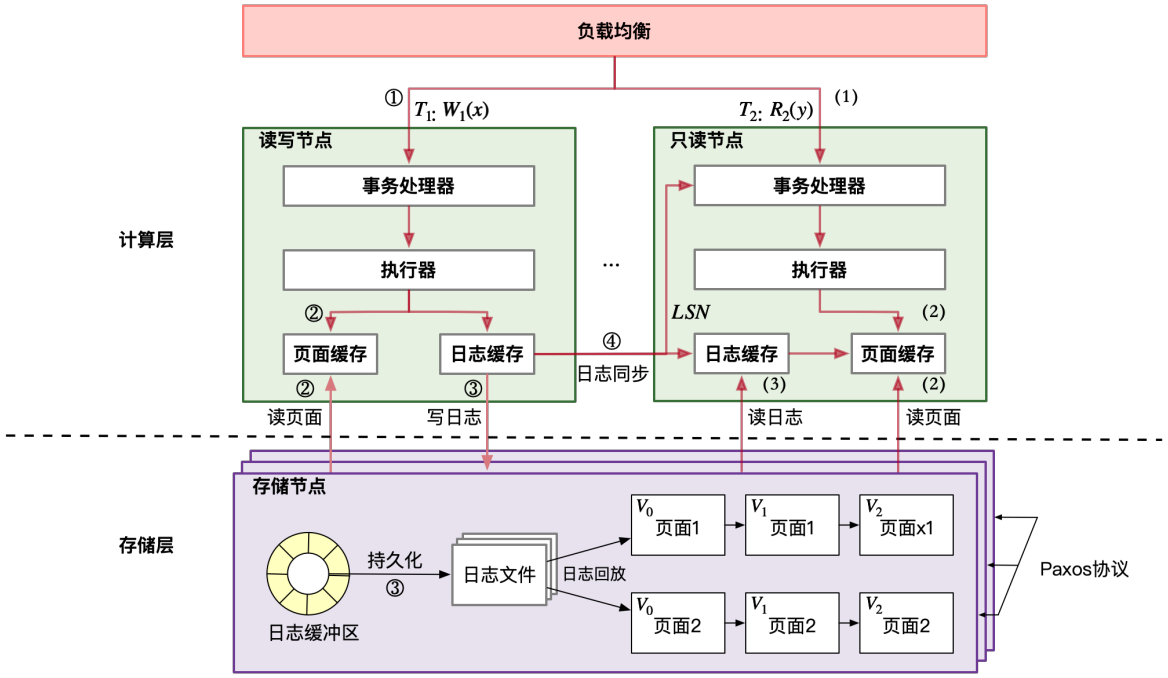


图 2 云原生数据库中的事务处理和高可用

读写和只读节点接收到事务后, 首先在本地缓存查找相关数据, 如果缓存未命中, 则需要从存储层请求指定版本的数据页面. 事务提交前, 需要本地日志缓存持久化到存储层才能进行事务的提交. 读写节点和只读节点通过日志来保证缓存一致性. 本文以图 2 中的读写事务 T_1 和只读事务 T_2 为例, 分别介绍云原生数据库读写事务和只读事务的处理逻辑. ①读写事务 T_1 在进行读写操作时, 需要通过并发控制算法来处理数据冲突, 例如在进行 x 的写操作之前首先需要获取数据项 x 上的锁. 在完成加锁后, ②读写节点首先在本地缓存中查找 x 并进行读写, 如果缓存未命中,则需要从存储层获取最新版本的数据页面, 在完成写操作后, 需要将重做日志写入日志缓存中. ③事务 T_1 提交前, 需要将日志缓存中的日志持久化到存储层, 系统通常采用一致性协议来保证日志节点高可用, 即只有大多数的日志副本写入成功后, 事务才可以成功提交. 读写节点并不会将数据页面写入存储层, 而是依赖存储层的日志回放线程回放日志生成新版本的页面. ④读写节点会将新产生的重做日志的日志信息异步广播给所有的只读节点, 日志信息通常包括日志序列号和日志在存储层中的存储位置, 只读节点将收到的日志信息写入本地缓存, 并更新本节点接收到的最新日志序列号. 需要注意的是, 上述方法中无法保证读写节点和只读节点的强一致性, 即只读节点读取不到读写节点最新修改的数据, 只能满足计算节点之间的最终一致性. 为了保证强一致性, 可以采用同步更新等方案, 即读写节点上的事务提交时, 需要保证该事务的修改已经成功发送给了所有只读节点才能成功提交, 但这样会牺牲读写节点的性能, 因此云数据库中通常采用前一种方法来保证节点间的最终一致性. 只读节点(1)接收到只读事务 T_2 后, 首先获取当前节点上接收到的最新日志序列号来作为当前读操作的快照点, (2)T_2 在读取数据项 y 时, 首先在本地缓存中进行查找, 如果缓存未命中, 只读节点从存储层使用快照点来请求指定版本的数据页面, 如果缓存中有数据项 y 的所需版本, 则直接读取, 如果缓存中数据项 y 所在的页面落后于 T_2 所需版本, (3)则根据日志缓存中的日志信息从日志节点中读取所需日志进行回放, 生成指定版本.

2.2 Calvin确定性并发控制算法

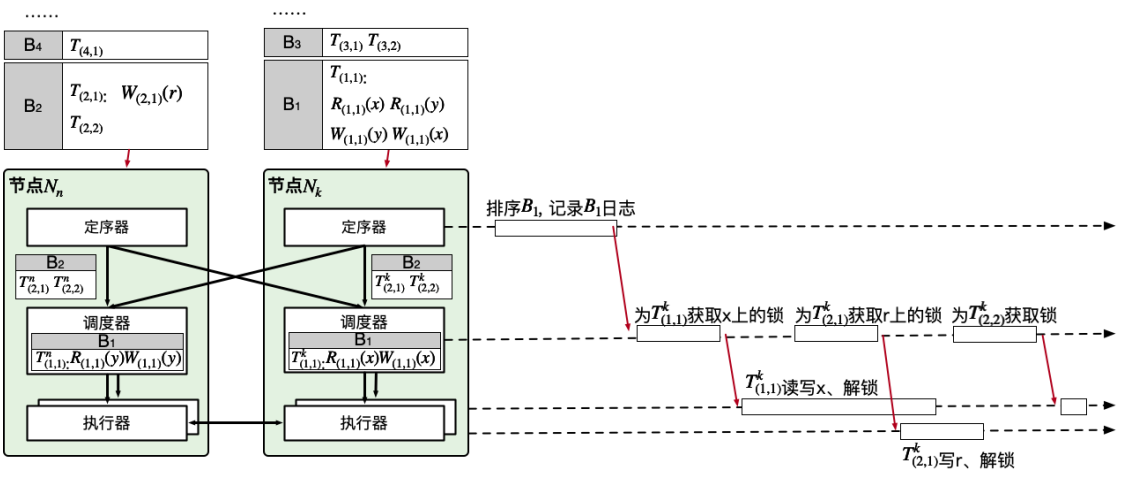


图 3 Calvin 架构中的事务处理

Calvin 是目前学术界最流行并且已经被商业数据库系统采用(如 FaunaDB[43])的确定性并发控制算法. 本文参考 Calvin 算法来实现 D3C 中的事务执行流程. Calvin 确定性并发控制算法的架构图如图 3 所示, 其中每个节点配备有三类组件: 定序器(sequencer)、调度器(scheduler)和执行器(worker). 其中定序器首先会接收来自客户端的事务, 然后将这些事务打包成批, 并按照接收到事务的先后顺序为这些事务确定顺序. 打包完毕之后, 定序器会将这个批中的所有事务作为逻辑日志存入磁盘当中. 定序器根据批中的事务需要访问的数据分片将这个批内的事务拆分为多个子事务, 并将对应的子事务分发给相应节点的调度器进行下一步执行. Calvin 对数据分片的策略没有特定的要求, 因此能够适用于常见的分片策略[44,45]. 调度器在接收到定序器发来的批内子事务之后, 则会依照定序器给定的顺序为这个批中的子事务加锁, 每当调度器为一个子事务加完所有锁, 调度器就会将这个子事务发送给执行器去实际执行. 由于 Calvin 的调度和执行都依赖于事务的读写集, Calvin 要求事务都预知读写集. 对于执行前未知读写集的事务, 可以通过侦察查询来预测读写集. 尽管预测的读写集和实际执行的读写集不符会导致事务回滚, 但 Calvin 已经证明这样的回滚并不会反复发生[32].

以图 3 中的两个节点 N_k 和 N_n 为例, 本文首先介绍 Calvin 是如何为这些事务确定顺序的. 节点 N_k 和 N_n 的定序器各自接收事务, 分别形成 B_1、B_2、B_3 等批. 定序器将为这些批中的事务分配事务标识, 并确定事务顺序. 本文中, 事务标识记作 $T_{(a,i)}$. 它是节点为每个批分配的批 ID 和批中事务分配的事务 ID 所组成的一个二元组. 其中 a 和 i 分别代表批 ID 和事务 ID. 注意, 每个节点会各自为收到的批分配批 ID, 批 ID 的值会以节点的数量为间隔增长, 以保证全局不会重复. 例如图 3 中的节点 N_k 按照 B_1、B_3、B_5...的顺序生成批, 而节点 N_n 按照 B_2、B_4、B_6...的顺序生成批. 同一批内事务的顺序将直接由事务 ID 的大小决定, 事务 ID 较小的事务将排在前面. 例如, B_2 中 $T_{(2,1)}$排在 $T_{(2,2)}$之前.批之间的顺序由批 ID 的大小决定, 批 ID 较小的批内所有事务将排在批 ID 较大的所有事务之前. 例如,B_2 中 $T_{(2,1)}$和 $T_{(2,2)}$都排在 B_3 中的 $T_{(3,1)}$之前. 为保证批之间的先后顺序, 调度器会依照批 ID 的顺序逐个从各个节点的定序器中获取, 即使某一个批中没有事务会在当前调度器上处理, 调度器也必须去尝试获取这个批, 来确保拿批的顺序. 当调度器完整地获取到一批事务后, 调度器才会按照顺序尝试获取批中的每个事务的锁.

接下来本文介绍 Calvin 中的事务是如何执行的. 在节点 N_k 和 N_n 定序器生成批并确定这些事务的顺序之后, 两个节点上的定序器将批内事务的逻辑日志写入磁盘. 随后定序器根据批访问的数据分片, 将批内事务拆分为多个子事务, 并将这些批内的子事务发送给管理对应数据分片的节点上的调度器. 事务 $T_{(1,1)}$分别访问节点 N_k 和 N_n 上的数据分片中的数据项 x 和 y, 事务 $T_{(1,1)}$会被拆分为子事务 $T^k_{(1,1)}$和 $T^n_{(1,1)}$, 其中 $T^k_{(1,1)}$在节点

N_k上读写数据项 x, $T^n_{(1,1)}$在节点 N_n上读写数据项 y. 之后 $T^k_{(1,1)}$被发送给节点 N_k的调度器, $T^n_{(1,1)}$被发送给节点 N_n的调度器. 调度器接收到这些批内的子事务后, 就按照事务的顺序为这些事务加锁. 例如节点 N_k的调度器会先给 $T^k_{(1,1)}$在数据项 x 上加写锁, 随后为 $T^k_{(2,1)}$在数据项 r 上加写锁, 再为 $T^k_{(2,2)}$要访问的数据项加锁. 如果一个子事务试图加锁的数据项已经被其他事务加锁, 它将在该数据项上排队等待. 调度器会继续按序为该事务要访问的其他数据项和后续事务要访问的数据项加锁. 当一个子事务获取到全部的锁之后, 这个事务就会被发送给执行器去执行, 完成读、写、解锁等操作.

注意, 在子事务 $T^k_{(1,1)}$执行结束时, 执行器并不会直接将子事务 $T^k_{(1,1)}$修改的数据项刷入磁盘, 也不会为子事务 $T^k_{(1,1)}$写入日志, 而是采用一种类似于 ZigZag[46]的方式来周期性地创建检查点(checkpoint). 在创建检查点之前, Calvin 会在整个系统中从所有已提交和正在执行的事务中选择一个最大的事务标识(ckpID)做为检查点的一致性点. 此时, Calvin 会对这些数据项设置一个额外的备份. 事务标识大于 ckpID 的事务会将它们的修改更新到这些数据的备份中, 事务标识小于等于 ckpID 的事务则继续更新在原数据项中, 因此检查点的创建只会因为维护数据的备份而影响系统性能, 不会导致事务阻塞. 当所有事务标识小于等于 ckpID 的事务提交后, 此时系统中原数据项的值达到了检查点要求的一致性点. 在这个时候,Calvin 将修改后的数据项刷到磁盘当中. 之后 Calvin 就会用数据项的备份覆盖掉原数据项, 让系统恢复正常执行. 通过使用检查点, Calvin 就可以利用定序器存储的逻辑日志和磁盘中的检查点, 将一个故障节点的状态恢复出来. 但是这样的方案难以做到高可用, 因为检查点的间隔时间通常较长, 通过检查点进行故障恢复需要耗费大量时间重做事务, 无法满足高可用要求. 如果缩短检查点的间隔时间, 则频繁维护数据备份的开销将会降低系统性能.

Calvin 算法还提出了一套基于备机的策略, 即给每个节点配备有多个备节点, 这些备节点会执行与主节点上完全一致的事务. 因此,当主节点故障后, Calvin 可以立刻将服务切换到备节点上, 从而实现高可用. 然而基于备机的策略带来了大量的计算资源冗余, 和云原生提高资源利用率、减少成本的原则相悖.

3 D3C：基于确定性并发控制的云原生数据库系统架构

设计支持多写的数据库架构的目标是在保证云原生数据库的两个重要特点存算分离和高可用的前提下, 突破云原生数据库一写多读的重要限制. 本文的思路是利用确定性的思想来调度事务处理, 从而保证各个节点可以在无需多节点协调的情况下, 独立检测和处理事务冲突, 以满足可串行化隔离级别的要求.

为了保证云原生数据库架构存算分离的特点, 本文设计的云原生数据库架构 D3C 如图 4 所示, 分为存储层、计算层和负载均衡器. 在存储层中, 本文将日志节点和数据节点区分成两类节点, 分别用于存储日志和数据. 将日志节点和数据节点分开做到了持久性与可用性分离, 使得两类节点可以根据日志存储和数据存储各自不同的特点采用不同的存储和备份策略. 为了保证存储层的高可用, 本文为日志和数据节点设计了多副本机制, 并通过基于 Quorum 的一致性协议[47]和 Gossip 协议[48]来共同保证日志和数据同步.

计算层包含了多个读写节点,负责独立地执行事务以实现多写. 为保证事务做到确定性的调度, 本文参考 Calvin 算法, 将其中的定序器、调度器引入云原生数据库架构. 为了保证事务持久性和事务执行的确定性, D3C 需要在事务执行之前将逻辑日志存入日志节点, 而日志节点难以直接利用逻辑日志回放数据, 因此 D3C 中需要一套高效的持久化数据技术将数据持久化到存储节点中, 以便于事务的高效执行, 为此本文设计了一个追踪器模块用于异步持久化数据, 有关数据持久化的细节本文将在 4.1 节介绍. 此外, 使用多副本来保证读写节点的高可用需要冗余大量的计算资源, 因此本文设计了基于一致性点的故障恢复机制, 使得 D3C 能够在无须冗余部署读写节点多副本的情况下保证读写节点的可用性. 这一点本文将在第 5 节中介绍.

如图 4 所示, 存储层中的日志节点负责持久化事务的逻辑日志. 它包含一个日志缓冲区, 用于接收来自计算层定序器发来的批. 日志缓冲区会将接收到的批以逻辑日志的形式同步刷入持久的日志文件中, 确保事务日志的可靠存储. 日志节点通过日志的批 ID 来保证日志的连续性和完整性, 当日志节点完成日志的持久化之后, 会通过批 ID 来检查日志是否存在空缺, 并使用后台线程定期与其他副本进行 Gossip 通信来填补日志空缺. 数据节点则负责存储用户的实际数据, 其中数据以键值对的形式组织, 每个数据项都维护了多个版本.

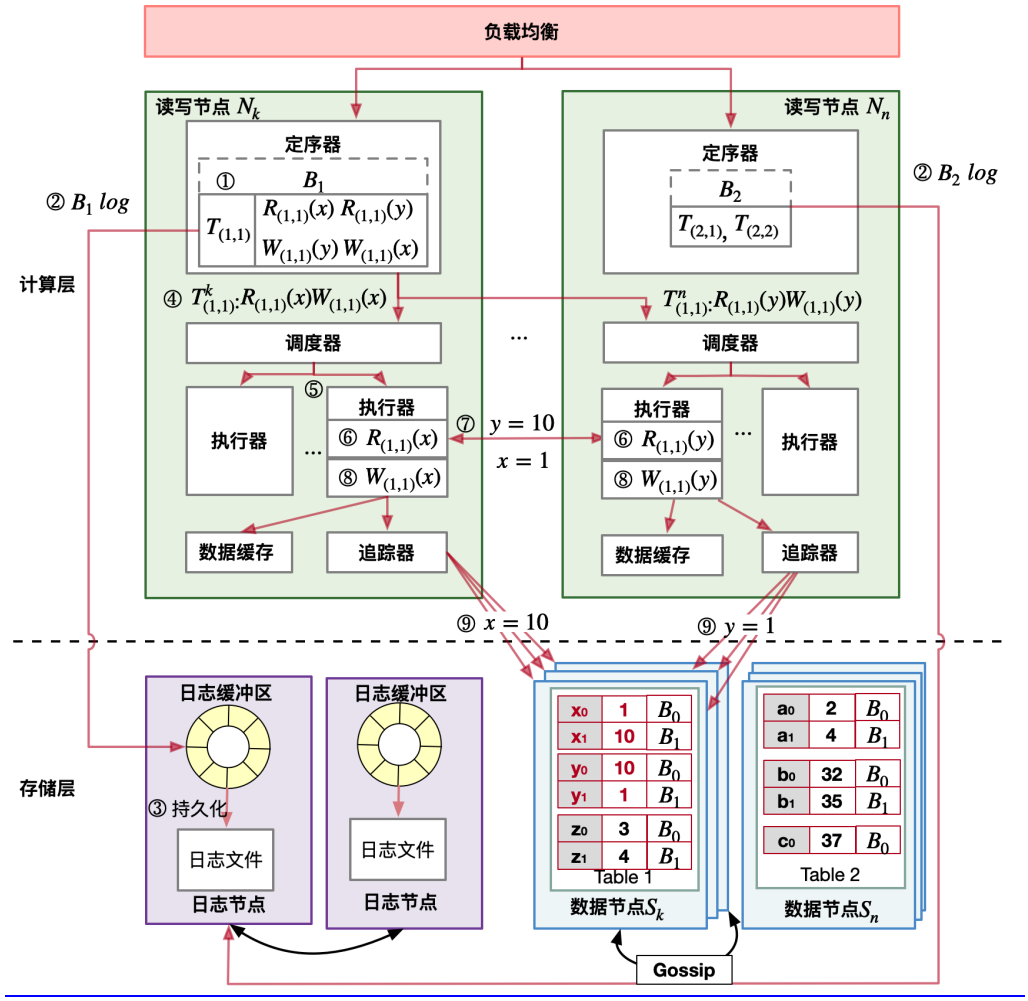


图 4 云原生多写事务处理架构 D3C

计算层中的每个读写节点配备有如下组件: 一个定序器、一个调度器、多个执行器、云原生数据库中需要的数据缓存, 和一个为基于多版本的异步批量数据持久化机制设计的追踪器组件.

- 定序器负责接收来自负载均衡器的事务, 参考 Calvin 算法中的顺序确定机制为这些事务排定先后顺序. 随后定序器会将这些事务打包成批, 并将这些批作为逻辑日志发送到日志节点进行持久化. 大多数日志节点持久化成功代表日志持久化完成. 然后定序器根据这些批内事务访问的数据分片, 将事务拆分为多个子事务, 发送给管理对应数据分片的节点上的调度器处理.
- 调度器则负责按照定序器指定的顺序为每个批中的子事务获取锁. 只有当这些子事务获得了所有的锁之后, 调度器才将这个子事务分派给执行器执行.
- 多个执行器并行的执行被调度器分派的已加锁的子事务, 利用数据缓存中的数据来执行这些事务的读写操作.
- 数据缓存是一个基于多版本的键值对内存存储, 其中缓存了数据节点中的数据.
- 追踪器用于追踪事务的修改, 并异步地将这些修改的数据刷入磁盘.

基于确定性并发控制的云原生数据库事务处理机制. 以图 4 中两个节点 N_k 和 N_n 为例, 事务在 D3C 中的执行流程包括 9 步. 首先, 节点 N_k 和 N_n 从负载均衡器中收集来自客户端的事务,并在定序器中将这些事务打

包成批(第①步). 例如, 在节点 N_k 上,定序器将事务 $T_{(1,1)}$打包进 B_1, 其中 $T_{(1,1)}$需要读取节点 N_k 上的数据项 $x(R_{(1,1)}(x))$, N_n 上的数据项 $y(R_{(1,1)}(y))$, 并交换数据项 x 和 y 的值, 即将 x 的值修改为 y 的值$(W_{(1,1)}(x))$, 将 y 的值修改为 x 的值$(W_{(1,1)}(y))$. 节点 N_n 上的定序器则将事务 $T_{(2,1)}$和 $T_{(2,2)}$打包进入 B_2. 接下来, N_k 和 N_n 的定序器会将 B_1 和 B_2 作为逻辑日志发送到日志节点当中(第②步), 日志节点的日志缓冲区将这些逻辑日志持久化到日志文件中, 读写节点将保证事务开始执行之前多数日志节点接收到日志(第③步). 接下来本文以 B_1 为例介绍后续流程, N_k 的定序器接下来将事务 $T_{(1,1)}$拆分为两个子事务 $T^k_{(1,1)}$和 $T^n_{(1,1)}$, 其中 $T^k_{(1,1)}$在节点 N_k 上读取数据项 x 并修改 x, $T^n_{(1,1)}$在节点 N_n 上读取数据项 y 并修改 y 的值. 之后定序器将这两个子事务发送到 N_k 和 N_n 两个节点的调度器当中(第④步). 接下来节点 N_k 和 N_n 的调度器根据子事务 $T^k_{(1,1)}$和 $T^n_{(1,1)}$的读写集, 对相应的数据项 x 和 y 加锁. 加锁完成后, 将子事务交给各自节点的执行器执行(第⑤步). 执行器将首先执行子事务的读操作(第⑥步). 随后, 执行器将读取的结果发送给其他节点(第⑦步). N_k 和 N_n 上的子事务在获取到另一个节点上子事务的读取结果后再执行写操作, 即 N_k 上的 $T^k_{(1,1)}$需要等待 N_n 上的 $T^n_{(1,1)}$读取 y 的结果后执行$(W_{(1,1)}(x))$, N_n 上的 $T^n_{(1,1)}$需要等待 N_k 上的 $T^k_{(1,1)}$读取 x 的结果后执行$(W_{(1,1)}(y))$(第⑧步). 执行器在收到依赖的读操作的结果之后执行子事务的写操作, 将对数据项 x 和 y 的修改写入数据缓存. 随后 N_k 的执行器释放掉数据项 x 的锁, N_n 的执行器释放掉数据项 y 的锁. 此时子事务 $T^k_{(1,1)}$和 $T^n_{(1,1)}$ 提交. 最后数据缓存中的追踪器将子事务 $T^k_{(1,1)}$对数据项 x 的修改, 子事务 $T^n_{(1,1)}$对数据项 y 的修改发送到数据节点, 在数据节点中写一个数据项 x 的新版本 x_1 和 y 的新版本 y_1(第⑨步).

需要注意的是, 如果事务因为执行时的读写集与预测的读写集不一致, 或因为事务本身的执行逻辑而回滚, 则该回滚将发生在第⑦步. 以图 4 中的事务 $T_{(1,1)}$为例, 假设实际不存在数据项 x, 则 N_k 向 N_n 发送读操作的执行结果后, 两个节点都会知道数据项 x 不存在, 并跳过第⑧步的写操作开始回滚.

4 事务优化方法

本节先后介绍了 D3C 中的基于多版本的异步批量数据持久化机制、基于数据一致性点的只读事务优化和基于多调度器的读写事务优化, 进一步提升 D3C 的事务处理性能.

4.1 基于多版本的异步批量数据持久化机制

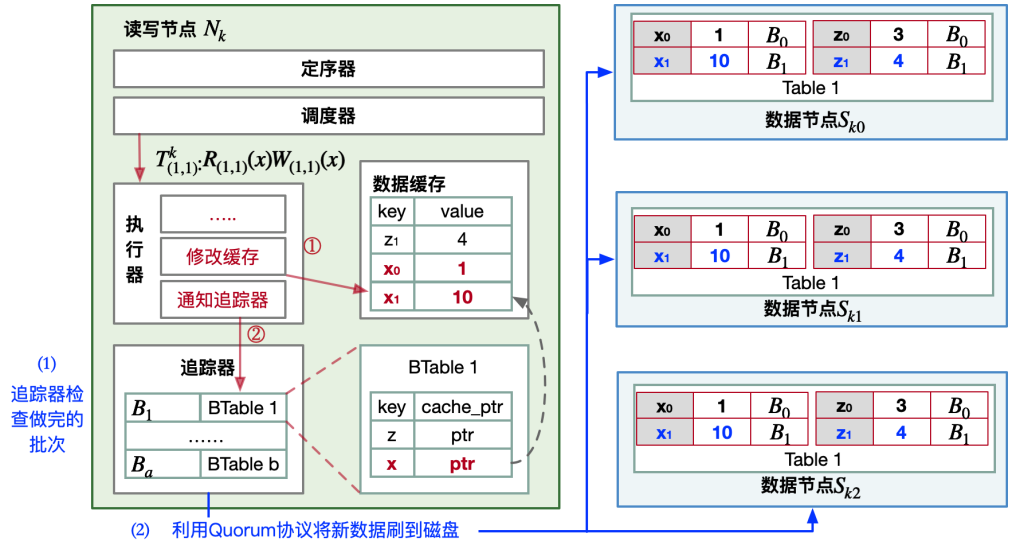


图 5 D3C 持久化机制

本文已经提到, 传统云原生数据库和确定性并发控制的数据持久化机制都不适用于 D3C. 而同步在存储

节点中写入事务的修改又会严重影响事务的性能. 为此, 本文提出了基于多版本的异步批量数据持久化机制, 具体思路是: 定期收集事务所产生的最新版本的数据, 并将这些最新的数据写入到存储层当中. 如图 5 所示, 本文在读写节点中引入一个追踪器来收集事务产生的最新版本的数据, 并定期将数据刷入磁盘. 追踪器中有一个单独的刷数据线程定期将数据刷入磁盘, 还维护有多个数据修改表(BTable), 每个 BTable 都对应一个当前尚未完成持久化的事务批, 比如 BTable 1 对应事务批 B_1. 每个 BTable 记录了对应批中的事务在所有数据项上最后的更改, 其中的每一项记录了被修改数据项的 key 以及其被修改值的缓存地址 cache_ptr.

接下来本文利用图 5 中的例子来介绍 D3C 是如何异步持久化数据的, 包含两个主要流程, 事务将数据写入缓存并通知追踪器(红色线以及红色字)以及追踪器异步将缓存数据刷入磁盘(蓝色线以及蓝色字). 本文沿用图 4 中的事务 $T^k_{(1,1)}$来介绍第一个流程. 在执行器执行完 $T^k_{(1,1)}$的加锁、读数据等操作之后, 执行器会为 $T^k_{(1,1)}$对数据项 x 的修改生成一个新版本 x_1(第①步); 之后则需要通知追踪器, 事务 $T^k_{(1,1)}$修改了数据项 x. 执行器会从追踪器中找到 B_1 对应的 BTable 1, 在 BTable 1 中记录数据项 x 最新修改值的缓存地址为版本 x_1 的地址(第②步). 之后执行器就完成了事务 $T^k_{(1,1)}$的执行. 注意, 此时数据项 x 的缓存会被锁定以防止缓存被换出.

接下来本文介绍追踪器异步将缓存刷入磁盘的流程, 追踪器会定期检查哪些批的事务已经做完. 在本文的例子中, 当追踪器检查到 B_1 已经执行完毕(第(1)步), 接下来追踪器的刷数据线程会根据 BTable 1 中的信息, 收集被 B_1 中事务修改的数据项, 包括数据项 x 和数据项 z. 接下来追踪器会利用 Quorum 协议, 将数据项 x 和 z 的修改同步地更新到数据节点 S_k 的三个副本(S_{k0},S_{k1} 和 S_{k2})当中(第(2)步). 当三个副本中的大多数更新数据成功, 追踪器完成对 B_1 和 B_2 修改数据的写入, 最后释放数据项 x 和 z 的缓存锁定.

4.2 基于数据一致性点的只读事务优化

通过使用本文 4.1 节中所描述的数据持久化方案, D3C 可以确保只读事务能够读取到最新且一致的数据, 同时保证数据库系统具备线性一致性. 但是在很多情况下, 只读事务并不需要读取到最新的数据, 只需要读取到一致性的数据即可.

因此, 本文希望提出一种只读优化, 通过牺牲只读事务的外部一致性, 避免只读事务执行并发控制来进一步提高系统的吞吐. 然而, 由于在 D3C 中, 事务的子事务之间不使用 2PC 协调提交, 而是在对应的读写节点上各自独立提交. 因而, 各个节点可能会出现事务执行进度不一样的情况, 导致不同数据分片的数据处于不一致的状态. 更进一步, 由于 D3C 采用基于 Quorum 的一致性协议来实现副本一致性, 当读写节点完成数据写入时, 只保证了数据节点中的大多数写入了数据, 因而, 同一数据分片不同副本的数据也会处于不一致的状态. 以图 6 为例, 由 B_1 修改的数据项 x 的版本 x_1 只在数据节点 S_{k0} 和 S_{k2} 上写入了多数节点, 正在用 Gossip 协议同步到 S_{k1}, 此时 S_{k1} 和其他两个副本之间的数据不一致. 而由 B_2 修改的数据项 r 的版本 r_2 已经完成了在 S_{n0}, S_{n1} 和 S_{n2} 所有节点上的同步, 而 B_2 修改的版本 x_2 还在尝试写入大多数,此时数据项 x 和数据项 r 之间也存在数据不一致问题.

如果不对只读事务进行并发控制, 让只读事务在随机一个副本上读取旧版本的数据, 可能会造成一个事务在不同数据项上读取到不一致的数据. 继续以图 6 为例, 事务 $T_{(4,1)}$需要访问数据项 x 和数据项 r. 如果 $T_{(4,1)}$选择 S_{k1} 和 S_{n1} 读取数据, 就会同时读到由 B_0 写入的 x_0 和由 B_2 写入的 r_2, 从而出现数据一致性问题. 因此, 在 D3C 中, 本文引入数据全局一致性点来保证只读事务在不进行并发控制的情况下能够读取到一致性的数据.

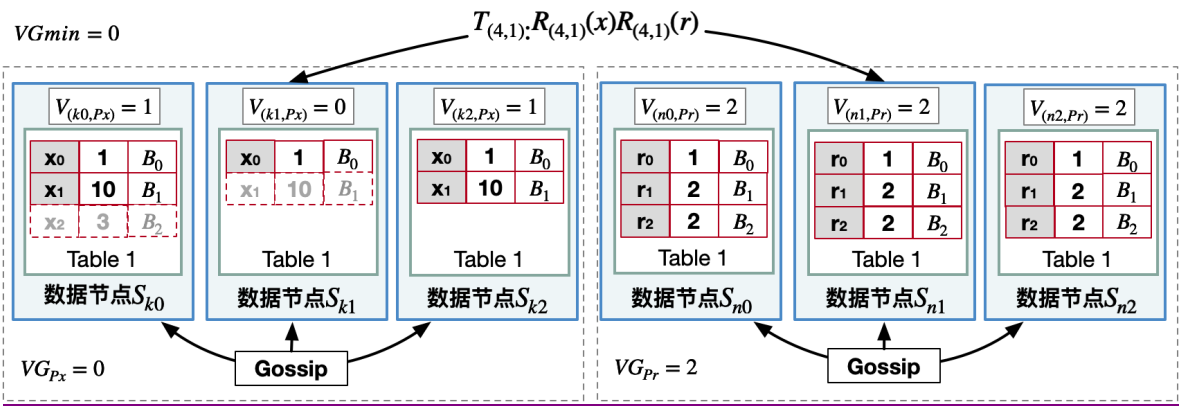


图 6 基于数据一致性点的只读事务优化

为了解决这个问题，本文首先在每个数据分片上引入数据一致性标识 $V_{(k,i)}$. 该变量表示数据节点 S_k 上存储的数据分片 i 已经完成持久化批的批 ID. 当读写节点将批 B_a 修改的数据发送给 S_k，并由 S_k 完成持久化后，$V_{(k,i)}$即被赋值为 a. 每个数据分片的副本所在的数据节点之间会同步该标识. 在这些副本节点上，本文引入分片全局一致性标识 VG_i，当数据分片的所有副本都已持久化批 B_a 时，数据分片将 VG_i 的值修改为 a. 每个数据节点将会获取所有数据分片对应的变量 VG，并将其最小值 VGmin 作为当前全局可获得的数据一致性的点，称为数据全局一致性点.

假设数据项 x 在数据分片 Px 上，数据项 r 在数据分片 Pr 上. 在上面的例子中，数据节点 S_{k0}, S_{k1} 和 S_{k2} 上 $V_{(k0,Px)}$, $V_{(k1,Px)}$和 $V_{(k2,Px)}$的值分别为 1, 0 和 1，代表 S_{k0}, S_{k1} 和 S_{k2} 上 Px 分别已经持久化了 B_1, B_0 和 B_1 的数据. 此时 Px 的分片一致性标识 VG_{Px} 为 0. 数据节点 S_{n0}, S_{n1} 和 S_{n2} 上 $V_{(n0,Pr)}$, $V_{(n1,Pr)}$和 $V_{(n2,Pr)}$识均为 2，代表 S_{n0}, S_{n1} 和 S_{n2} 上 Pr 都已经持久化了 B_2. 此时 Pr 的分片一致性标识 VG_{Pr} 为 2. 因此当前数据全局一致性点 VGmin 为 0. 所以 $T_{(4,1)}$只能读取 x_0 版本和 r_0 版本，来保证数据一致性.

注意，系统中可能存在一个批中的所有事务都不修改某个数据分片 i 上的数据的情况. 在这种情况下，即使读写节点已经将所有该批修改的数据持久化，由于数据分片 i 的 VG_i 没有增长，数据全局一致性点也同样无法增长. 为了避免这一情况的发生,当一个批不需要对数据分片 i 的数据做出任何更改时，与数据分片 i 对应的读写节点仍需通过追踪器向数据分片 i 对应的存储节点发送一个仅包含当前执行完的批 ID 的消息，从而保证数据分片 i 上的 VG_i 能够增长.

通过上述机制，系统中的读写事务和只读事务都可以获取到一致性的数据. 对于读写事务 $T_{(a,i)}$，如果 $T_{(a,i)}$需要获取数据项 x，且 x 在读写节点的缓存中未命中，读写节点仍然需要从数据节点获取 x 的最新版本. 这是因为读写事务需要读取最新的数据项从而保证可串行化. 通过追踪器锁定尚未持久化的数据缓存，数据库系统保证了最新的数据项在被持久化之前不会从缓存中被丢弃. 即，最新的数据项要么在读写节点的数据缓存当中，要么已经被持久化在数据节点当中. 对于只读事务 $T_{(a,j)}$, $T_{(a,j)}$会首先获取数据全局一致性点 VGmin，如果 $T_{(a,j)}$在获取数据项 x 时，数据缓存中未命中，读写节点可以选择从任意数据节点读取 x 在数据全局一致性点上的版本，从而提高事务执行效率.

在 D3C 中，只读事务有两种执行方法. 一种方法是只读事务仍被拆分为多个子事务. 各个子事务被负责不同分片的读写节点调度，由这些读写节点从各自的缓存中或从存储节点获取一致性的旧版本的数据. 另一种方法是只读事务只由单一读写节点进行调度，并由该读写节点从存储节点获取全部的旧版本的数据. 这两种方法各有优劣. 前者能够避免读写节点的缓存中换入不属于自己负责的分片的数据，然而当只读事务涉及到多个读写节点负责的分片时，只读事务需要一次额外的消息通信以便在单个读写节点上获取到所有需要返回给客户端的数据. 后者则避免了额外的消息通信，但在节点的缓存空间紧张时，容易导致性能问题. 在

D3C 的实际实现当中, 本文选择了第一种方式, 将只读事务拆分为多个子事务.

4.3 基于多调度器的读写事务优化

单个调度器按定序器预先给定的顺序对节点上的子事务进行加锁,能够确保子事务获取锁的顺序完全与定序器给事务预先确定的顺序一致. 而事务被要求在获取到所有的锁才能开始执行. 因而, 单个调度器能够保证事务执行的正确性和确定性. 然而, 在 7.2 节的测试中本文发现, 当云原生数据库的缓存容量足够大时, 单个调度器将成为事务执行的单点瓶颈, 这和其他工作[38,49]中所观察到的现象一致. 这是因为单个调度器需要为该节点上的众多执行器提供完成加锁的事务, 执行器的事务执行效率高于了调度器的事务加锁效率.

在 4.2 节中, 本文已经提出了基于数据一致性点的只读事务优化, 通过这一优化, 只读事务无须通过调度器进行并发控制即可执行. 因而, 对单个调度器的优化将针对读写事务进行.

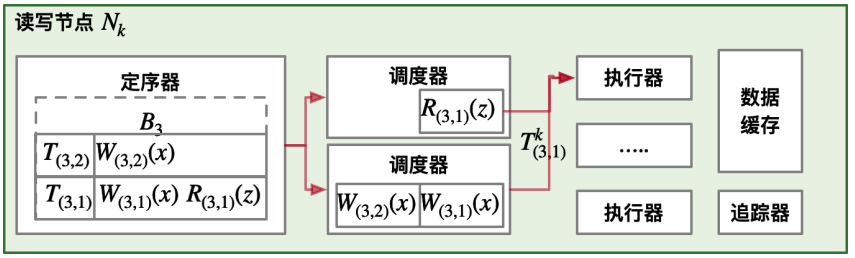


图 7 多线程加锁

为了解决这个单点瓶颈, 如图 7 所示, D3C 中首先将读写节点负责的数据分片进一步划分为多个子分片, 并为这些子分片配备相同数量的调度器. 一个调度器负责为一个子分片上的事务读写操作加锁. 如图 7 中所示, 当节点 N_k 的调度器接收到两个 B_3 的子事务 $T^k_{(3,1)}$和 $T^k_{(3,2)}$. 其中 $T^k_{(3,1)}$需要获取数据项 x 和 z 的锁, 而 $T^k_{(3,2)}$只需要获取数据项 x 的锁. 数据项 x 和 z 分别处于两个子分片当中. 此时, 负责数据项 x 子分片的调度器就会按照 $T^k_{(3,1)}$、$T^k_{(3,2)}$的顺序为这两个子事务在数据项 x 上加锁; 而在数据项 z 所在子分片当中, 对应的调度器就可以并发的帮助 $T^k_{(3,1)}$获取到数据项 z 的锁.

由于子分片完全独立, 使用单一的调度器为子事务在子分片上加锁仍然能保证子事务在子分片上获取锁的顺序与定序器给事务预先确定的顺序一致. 需要注意的是, 在读写节点 N_k 上, 事务 $T^k_{(3,1)}$并不会被交给多个执行器执行, 依旧是一个完整的子事务. 这样避免了多个更细粒度的子事务间发生更多的消息通信以及在多个执行器上对内存特定结构产生争用.

5 故障恢复机制

在云原生数据库中, 存储节点可以通过使用一致性协议达成高可用. 读写节点则利用只读节点作为它的备节点来保证高可用. 在读写节点故障时, 由一个只读节点切换为读写节点来继续提供服务. 在 D3C 中, 存储节点沿用了基于 Quorum 的一致性协议来保证高可用. 具体来说, 当一个日志节点发生故障时, 系统重启该日志节点或启动新的日志节点, 并通过 Gossip 协议将其他日志节点中已经持久化的日志数据同步到当前新启动的节点. 数据节点故障后的恢复流程也大致相同. 由于一致性协议的保证, 一个存储节点的故障不会影响整个存储服务, 读写节点仍然可以正常持久化日志和读取数据. 然而, 读写节点没有备节点可用于故障恢复, 并且确定性并发控制的原有的通过检查点进行故障恢复的机制恢复时间较久, 不能满足云原生数据库的高可用要求. 因此, 本节的重点在于如何实现读写节点的高可用.

如图 8 所示, 读写节点 N_0 的定序器正在将 B_3 的逻辑日志写入日志节点, 执行器正在将 B_3 对数据项 x 的修改写入数据节点, 如果此时读写节点 N_0 发生故障, 数据库中只有一个日志节点完成了 B_3 的日志的持久化, 剩余两个节点中都未完成日志持久化, 日志节点之间处于不一致状态, 此外, 如 4.2 节的描述, 数据节点之间也处于不一致的状态. 在这种不一致的状态下, D3C 需要在不阻塞其他读写节点执行的前提下, 启动新的读

写节点来取代 N_0 并通过故障恢复机制将数据库系统恢复到一致性状态. 因此, D3C 的故障恢复流程需要解决三个问题: (1)启动新的读写节点后, 数据库系统如何识别到不一致状态, 并决定从哪里开始进行故障恢复; (2)数据库需要恢复到什么状态新的读写节点才能够正常提供服务, 结束故障恢复流程; (3)如何进行故障恢复, 将数据库从不一致状态恢复到一致性状态. 针对上述三个问题, 本文提出了基于一致性点的故障恢复机制, 在启动新的读写节点后, 首先需要获取数据大多数一致性点和日志一致性点, 并从数据大多数一致性点开始重做相关逻辑日志直到完成日志一致性点的重做, 此时重启的读写节点能够正常提供读写服务, 完成了故障恢复流程.

5.1 基于Quorum协议的一致性点构建机制

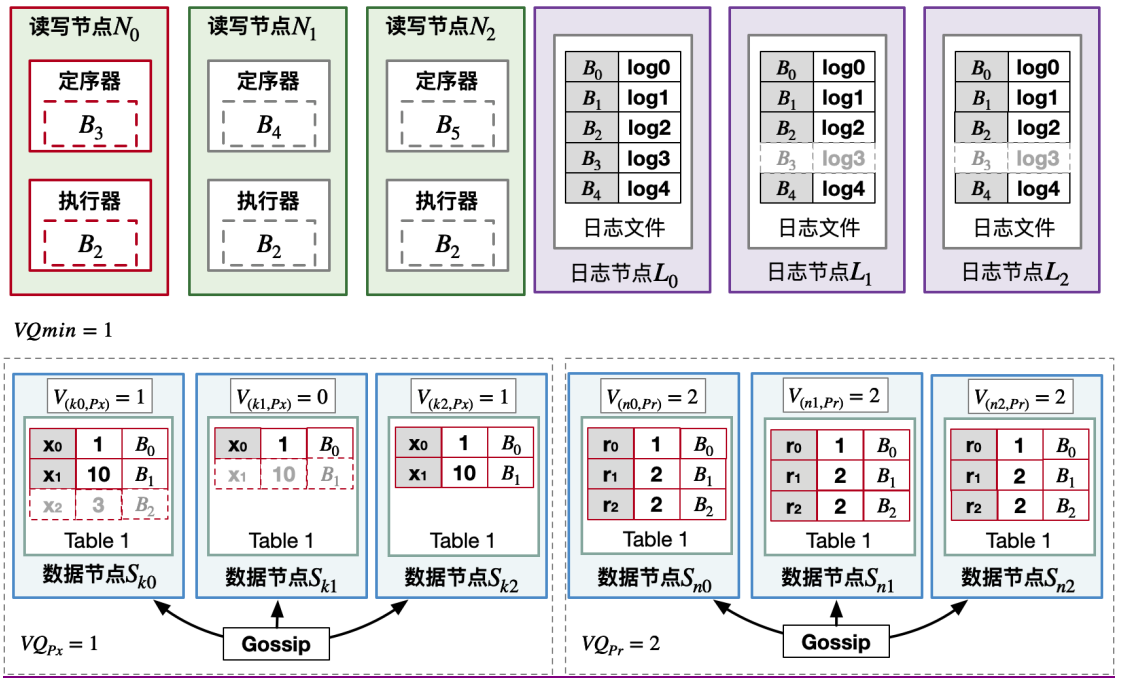


图 8 基于一致性点的故障恢复

数据大多数一致性点: 在 4.2 节中, 本文描述了如何获取数据全局一致性点以及如何使用数据全局一致性点来对只读事务进行优化. 与 4.2 节中的数据全局一致性点类似, D3C 使用数据大多数一致性点来保证数据节点的高可用. 对于数据分片 i, 本文引入分片大多数一致性标识 VQ_i, 当该分片的大多数副本都已经持久化了批 B_a 的修改数据后, 该数据分片将 VQ_i 更新为 a. 当需要进行故障恢复时, 读写节点会获取所有数据分片的 VQ, 并将其最小值 VQmin 作为当前全局可获得的数据大多数一致性点. 如图 8 所示, 对于数据项 x 所在的数据分片 Px, 大多数的数据副本持久化了 B_1 的修改, 因此该数据分片的 VQ_{Px} 为 1, 同理数据项 r 所在数据分片的 VQ_{Pr} 为 2, 因此当前状态下数据大多数一致性点 VQmin 为 1.

日志一致性点:在 D3C 中, 读写节点通过日志节点的连续日志最大批 ID 来获取日志一致性点. 对于每一个日志节点, 当存在一个批 ID 为 B_c 的批, 该批及该批之前的批日志都已经被持久化, 且该批的后一个批的日志尚未持久化时, B_c 被称作是该日志节点的连续日志最大批 ID. 读写节点需要读取大多数节点并获取这些节点中最小的 B_c 作为日志的一致性点, 例如图 8 中日志节点 L_0, L_1 和 L_2 的连续日志最大批 ID 分别为 B_4, B_2 和 B_2, 在新的读写节点恢复时, 读取了 L_0 和 L_1, 则此时日志的一致性点为 B_2. 日志一致性点被应用在读写节点故障恢复流程中来保证日志节点的一致性.

5.2 基于一致性点的故障恢复机制

在引入数据大多数一致性点和日志一致性点之后, 本文能够回答第 5 节开头提出的三个问题.

(1)**识别不一致状态, 并决定故障恢复起点.** 所有读写节点向存储节点请求当前的数据大多数一致性点 VQmin, 该一致性点之后的数据均为不一致状态. 数据大多数一致性点为故障恢复的起点.

(2)**识别故障恢复终点.** 新启动的节点获取日志节点的日志一致性点, 日志一致性点就是故障恢复终点.

(3)**执行故障恢复流程.** ①所有读写节点获取 B_{VQmin} 的下一个批的事务日志. ②新启动的读写节点根据事务日志重做事务, 其他节点寻找该批中涉及故障节点负责分片的事务, 根据依赖关系重做对应的数据项上的读写操作, 用于协助新启动的节点处理不属于该节点分片的读写依赖. ③获取下一批的事务日志, 并重复步骤②, 直到新启动的读写节点执行到日志一致性点, 并完成日志一致性点相关日志的重做.

当新启动的读写节点执行完成日志一致性点, 则数据库系统完全恢复服务. 系统在恢复服务的过程中不会完全失去可用性. 这是因为在完成恢复前, 只有故障节点负责的数据分片完全无法对外提供服务. 在故障节点恢复之前, 其他节点可以跳过该故障节点对应的批继续执行. 在图 8 的例子中, N_1 和 N_2 将跳过故障节点 N_0 对应的 B_3 执行 B_4 和 B_5. 并且, 如果一个事务没有子事务涉及故障节点, 则这个事务仍然可以执行和提交, 不需要被阻塞. 在图 8 的例子中, N_1 和 N_2 可以执行 B_2 中不涉及 N_0 的子事务, 而将涉及 N_0 的子事务阻塞. 在 B_2 中不涉及 N_0 的子事务执行完时, 继续执行 B_4 中不涉及 N_0 的子事务.

在获取日志一致性点后, 新启动的读写节点需要判断所有批 ID 超过日志一致性点的日志, 检查并维护日志的连续性. 其目的是, 防止出现系统再次故障时, 新启动的读写节点再次获取到上一次故障的日志一致性点(例如, 在图 8 中, 当 N_0 再次故障时, 由于 B_2 后仍然不连续, 日志一致性点再次获取到 B_2). 具体来说, 新启动节点需要遍历所有日志节点, 判断所有由故障节点负责且批 ID 超过日志一致性点的批事务的日志是否被写入大多数节点, 如果该批事务的日志未被写入大多数节点, 则使用空日志替换所有已经持久化的该批事务的日志, 并使用空日志填补未持久化该批事务日志的日志节点, 从而保证日志的编号连续. 使用空日志作为该批事务的日志, 是因为故障时,这一批事务尚未完成持久化, 并没有发送给其他计算节点执行, 其他计算节点也会跳过该批事务的执行, 因而应当将该批视作没有事务. 以图 8 为例, 新启动的读写节点在获取日志一致性点 B_2 后, 需要检查所有超过 B_2 且由当前节点负责的批事务的日志, 发现批 B_3 的日志为故障节点负责的事务且该日志只写入了 L_0 节点, 未被写入大多数节点, 则需要生成空日志替换 L_0 中批 B_3 的日志并将空日志填补到其他所有日志节点中.

6 系统实现

目前, 没有云原生数据库的开源的原型系统. Deneva[40]是一个被广泛用于并发控制性能测试的开源的内存分布式数据库的原型系统, 实现了包括 Calvin 在内的许多并发控制. 并且, 该系统还高度模块化, 易于扩展. 因此, 本文选择修改 Deneva 来构建云原生数据库的原型系统 Cloud-Deneva†, 实现了一写多读和 D3C 两套架构.

6.1 架构与数据结构

Cloud-Deneva 是一个面向 OLTP 的云原生数据库原型系统, 由客户端, 计算节点和存储节点组成.

客户端负责在系统初始化时生成事务负载, 并在系统运行时发送包含事务读写集的事务消息来调用计算节点的预编译的存储过程. 预编译的存储过程是使用 C++来编写的, 支持对数据的增删查改等读写操作. 这种预编译的存储过程在许多的原型系统中都有应用[14,25,32]. 在系统运行中, 客户端对存储过程的调用是闭环(closed-loop)的, 即客户端将始终确保计算节点运行指定数量的事务. 计算节点每完成一个事务, 客户端才将下一事务发送给计算节点. 客户端还承担了 D3C 中负载均衡器的功能. 对于一写多读, 客户端不仅能够保证

† https://github.com/dbiir/Cloud-Deneva.git

读写事务与只读事务分别路由到一写多读下的读写节点和只读节点上, 并能够在读写节点较为空闲时, 将部分的只读事务分配给读写节点.

计算节点负责事务的执行. 在一写多读中, 计算节点又分为读写节点和只读节点. 在 D3C 中,所有的计算节点都是读写节点, 包含定序器, 调度器, 执行器等事务执行组件以及缓存和追踪器. 为了避免定序器, 调度器和执行器互相抢占计算资源, 它们都绑定在CPU核心上. 缓存的具体结构和访问方式将在6.3节介绍. 值得注意的是, 定序器, 调度器和执行器都实现在同一个节点上. 从功能上来说, 调度器和执行器是紧密结合的, 如果将它们拆分到不同节点, 意味着读写事务执行都需要为加锁和解锁操作增加网络通信开销. 定序器则可以被拆分到不同的节点上单独扩展. 然而, 本文 7.2 节中的实验表明, 在云原生数据库中, 调度器和定序器将先于执行器到达瓶颈. 因此, 在系统中更应该考虑调度器和定序器成为瓶颈的问题. 更进一步, 由于执行器相比于调度器和定序器更为空闲, 相比于为调度器或定序器单独增加节点, 在节点内调配计算资源, 减少空闲的执行器, 增加调度器和定序器数量将能更充分的利用计算资源. 具体来说, 本文在 4.3 节中已经说明了调度器如何在节点内扩展, 可以通过减少执行器的线程数来增加调度器. 定序器由于功能相对独立, 同样可以直接牺牲执行器线程增加定序器线程. 如何协调各个组件的数量来达到最优的性能不是本文的重点, 本文将其作为未来的工作.

存储节点融合了 D3C 中的日志节点和数据节点. 因此, 存储节点具有日志持久化和日志回放两种线程. 注意, 原型系统中未区分日志节点和数据节点是为了方便系统的开发和部署. 系统中用于测试的表结构是预先定义的, 表中的数据在存储节点初始化时生成. 与一般的云原生数据库的存储节点采用数据页的形式组织数据不同, 在 Cloud-Deneva 中, 存储节点采用键值对的形式组织和存储数据. 每个表都具有一个主键, 表中的数据通过哈希索引或 B+树索引作为主索引进行组织. 一个表也可能具有次级索引, 次级索引也直接指向对应的数据项. 一个数据项可能有多个版本, 这些版本的位置都被记录在数据项上. 存储节点通过哈希运算将数据映射到各个数据分片中, 从而实现水平分片.

6.2 负载均衡

负载均衡模块实现在客户端当中. 每个客户端会分别统计自己有多少事务消息发送给了不同的计算节点. 并且会根据统计, 调整自己给不同计算节点发送的事务数量以保证负载均衡. 因此, 每个客户端都能实现它向计算节点发送的事务消息是负载均衡, 所有计算节点总体上也是负载均衡的.

具体来说, 对于一写多读, 客户端首先判断将要发送的事务是否为读写事务. 如果该事务为读写事务, 则该事务直接被发给读写节点. 而当事务为只读事务时, 客户端需要判断当前负载下发送给读写节点的事务消息是否少于只读节点, 如果是, 则将当前事务发送给读写节点补足消息数量, 否则, 客户端将只读事务发送给只读节点. 因而, 一写多读在读写事务比例较高的情况下, 客户端只能保证只读节点之间能够做到负载均衡, 读写节点的负载会较只读节点更重. 而读写事务比例较低的情况下, 所有计算节点之间都能做到负载均衡. 对于多写架构, 每个计算节点都只和相同数量的客户端连接. 在系统初始化时, 每个客户端都给单个计算节点发送数量相同的事务消息, 并通过前述闭环调用的方法, 在计算节点执行完一个事务时才将下一个事务发送给计算节点, 从而保证每个计算节点都同时执行有相同数量的事务.

6.3 数据缓存

计算节点的缓存是多个双向链表组成的数据结构, 其结构如图 9 所示. 不同数据项会根据其主键 key 的哈希值映射到数据缓存模块中的某一个双向链表, 如图 9 中, 数据项 x 和 z 都映射到了第一个双向链表当中. 在每个双向链表中, 其中的每一个元素由四个部分组成: 数据项的主键 key, 数据项的版本链 row, 指向链表中前一个元素的指针 prev_ptr 和指向后一个指针的 next_ptr. 其中版本链 row 是一个环形队列, 用于存放一个数据项的多个版本, 其大小可以被调整, 从而满足不同负载下需要缓存的版本数量的差异. 在 D3C 中, 计算节点数据项上的多版本环形队列中的元素记录了一个批最后对该数据项的修改, 即仅当下一批事务开始访问该数据项时, 环形队列的指针才会向前移动. 这样的设计是因为追踪器在实现时一个 BTable 也仅对应一个批,

通过一个批共用一个数据项版本，能减少追踪器追踪版本的开销.

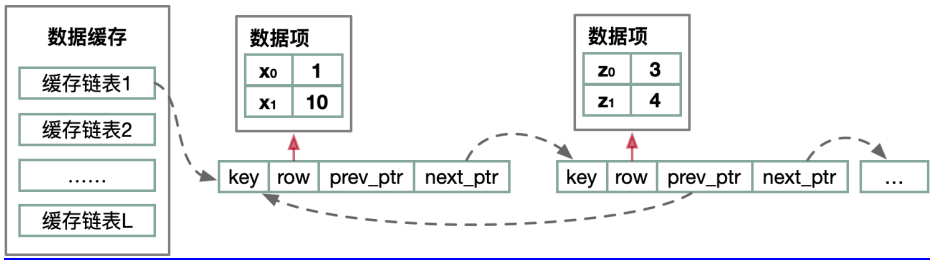


图 9 缓存数据结构

接下来本文分别介绍缓存模块的换进换出机制、事务获取数据项缓存的机制、一写多读和 D3C 中获取缓存的差异.

换进换出机制. 计算节点缓存的换进换出机制采用了 LRU 算法. 当某个数据项要换入缓存时，系统会首先获取数据项的主键，通过哈希运算将数据项映射到某个双向链表上，并在链表尾端为数据项创建链表节点. 如果该双向链表无法接受更多的数据项缓存时，缓存将从最链表的头部开始将数据项换出. 当某个在缓存中的数据项再次被使用时，系统会将该数据项的链表节点移到链表的尾端.

事务获取数据项缓存的机制. 当数据项 x 或数据项 x 的某个版本在缓存中未命中时，一次只会有一个事务从存储节点获取缓存. 其他需要该数据项缓存的并发事务则会记录自己对数据项 x 的需求并陷入等待. 当一个事务获取到 x 的缓存后，会通知那些需求数据项 x 缓存的事务开始执行. 当事务成功命中缓存后，事务会将该缓存暂时锁定，直到该事务完成执行，从而避免数据项被过早换出，事务后续无法正常读写数据项. 这种一次只有一个事务从存储节点获取缓存的设计能够避免存储节点忙于计算节点获取数据项的请求，从而有节省出更多的计算资源用于日志回放等其他工作.

一写多读和 D3C 获取缓存的差异. 在一写多读和 D3C 两种不同的架构中，其主要差异在于事务从存储节点获取缓存的方式. 一写多读架构下，事务会逐个访问数据项，如果访问某个数据项时缓存未命中，则事务将尝试从存储层获取数据项并等待直到缓存获取成功，并在每次缓存未命中时，都需要单独的从存储层获取数据项. 因而，当缓存较小，命中率较低时，一写多读需要事务花费较多的时间在获取缓存上，导致缓存获取成为事务执行的瓶颈. 由于 D3C 在事务执行前已经提前确定了事务的读写集，D3C 允许事务一次性获取所有未在数据缓存中的数据项. 更进一步，由于 D3C 在事务执行前先执行事务调度. 事务将在调度时检查其读写集中涉及的数据项是否都已经在缓存中，并一次性获取未在缓存中的所有数据项. 通过这种方式，D3C 下的事务执行将更少受到缓存命中率的影响，从而在计算节点缓存容量较小时能取得较好的性能.

6.4 日志机制

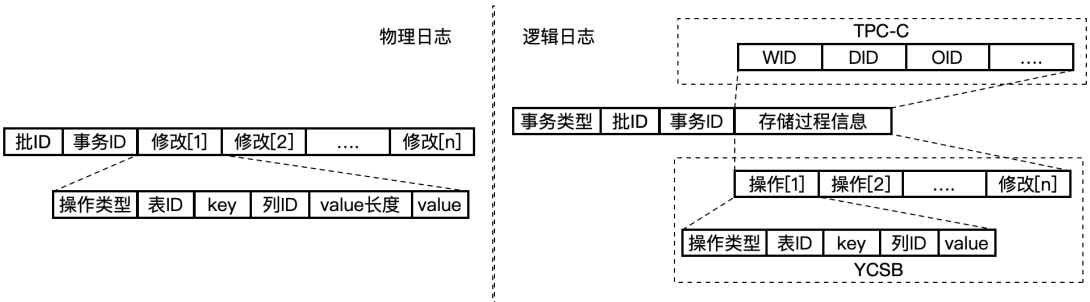


图 10 日志结构

图 10 分别展示了一写多读和 D3C 中的日志结构. 左图展示了用于一写多读的物理日志，右图展示了用于 D3C 的逻辑日志.

一写多读通常使用 WAL(write ahead log)日志[49]来持久化事务的执行结果. 常见的 WAL 日志通过记录物理页面和页面上的偏移量来记录事务修改的位置, 并使用 LSN(log sequence number)来追踪数据页的版本. 在 Cloud-Deneva 中, 一写多读也使用了 WAL 日志, 但是因为 Cloud-Deneva 使用的是键值对存储, 本文用表的标识和数据项主键取代取代了原本 WAL 日志中的数据页的标识; 同时由于 D3C 以批为单位来追踪版本和确保数据的一致性, 为了一写多读实现简便, 本文利用批 ID 取代了 LSN, 并以此来回放和读取对应的数据项. 因此, 物理日志中包括如下内容: 事务所在的批 ID, 事务 ID, 以及事务的每个修改. 每个修改里记录了操作的类型, 修改的表、数据项、列, 以及修改的值和值的长度. 其中的批 ID 会在事务提交时, 通过原子操作顺序来顺序地获取. 为了解决一写多读下可能会出现的数据一致性问题, 本文使用了与 4.2 节中一致的数据一致性方案, 具体来说, 存储节点每完成一个事务的日志回放, 就会更新对应数据分片的 $V_{(k,i)}$.

逻辑日志则首先记录了事务的类型, 用于确定日志中包含的值的具体语义, 比如标记当前事务是 TPC-C[50]负载中的 NewOrder 事务. 日志随后记录了调用预编译存储过程所需要的全部信息. 举例来说, 对于 YCSB(Yahoo! cloud serving benchmark)[51]负载, 逻辑日志将记录事务每个要访问的数据项的 key, 事务对该数据项的操作类型(读操作/写操作), 以及对于事务的写操作需要记录该操作将要修改的值. 而对于 TPC-C 负载, 由于预编译的存储过程中已经记录了每种事务对于每个表中数据项的操作类型, 因此, 逻辑日志将只记录事务要访问的数据项, 以及事务执行逻辑中的其他参数,比如访问的仓库 ID(WID), 区 ID(DID)和订单 ID(OID).

7 实验分析

在这一节, 本文在 Cloud-Deneva 中对比了传统的一写多读和 D3C 中事务的吞吐情况. 其中一写多读使用了 Silo[25]作为并发控制, 该并发控制算法在单机上具有良好的性能.

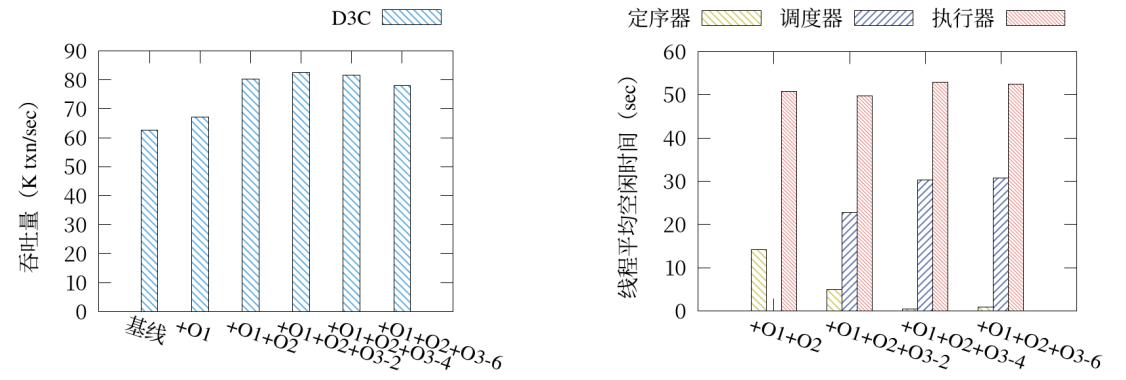
7.1 实验设置

测试平台. 本文除 7.4 节之外的实验, 均部署在 6 个虚拟节点中, 其中 2 个节点包含 12 核/24 线程并且拥有 32GB 内存用于部署计算节点, 其余 4 个节点包含 4 核/8 线程并且拥有 32GB 内存用于部署客户端和存储节点. 在 7.4 节可扩展性的测试中, 本文在阿里云上部署了 6 个 ecs.c6.8xlarge 实例, 每个实例配备 32 线程 64GB 内存. 每个实例当中各部署一个计算节点, 一个客户端和一个存储节点. 在以上两种配置当中, 计算节点中包括 16 个线程作为执行器和调度器、1 个线程作为定序器以及若干负责节点间通信和事务管理的线程. 客户端中包括 4 个线程用于生成事务, 其余线程负责节点间通信. 存储节点中包括 1 个线程负责管理日志, 3 个线程用于数据回放, 另有若干线程负责节点间通信.

工作负载. 性能测试使用 YCSB 和 TPC-C 两种负载. YCSB 是雅虎公司开源的数据库服务器端压力测试工具. 它提供了可调试参数如写操作比例, 冲突率等, 以进行全面的评估. 事务访问服从 Zipf 分布, 即少量数据获得大量访问的长尾分布. 倾斜率在 0 到 1 之间, 越接近 1 数据访问冲突越大, 用于测试系统在不同冲突率下的性能表现. 默认情况下, 本文设置了与存储节点数量相同的数据分片, 每个数据分片包含 2^23, 约 800 万条记录. 每个事务执行 10 次读/写操作. 默认的倾斜率为 0.3, 写读比为 0.3(即 30%读写事务和 70%只读事务). TPC-C 则是一种流行的 OLTP 基准, 模拟了一个仓库订单处理应用程序. 其中包含 9 张表,每个仓库包含 100MB 的数据大小. 默认情况下, 本文为每个数据分片设置 32 个仓库. 事务生成比例与 TPC-C 标准一致, 因此只读事务占比为 8%.

配置.此外, 本文额外引入了三个指标:分布式事务率、缓存行数量、D3C 中调度器数量. 分布式事务率量化系统中访问跨多个节点数据的事务比例, 默认情况下, 本文将 YCSB 工作负载的分布式事务率配置为 20%. 而在 TPC-C 负载中, 其中 10%的 NewOrder 事务和 15%的 Payment 事务为分布式事务, 其他事务均为单机事务. 缓存行数量用于指示计算节点可用于存放记录的缓存大小, 本文默认将缓存行数量设置为 200 万. 调度器数量则表示每个计算节点的 16 个用于执行器和调度器的线程中有多少个线程作为调度器, 本文默认将加锁线程数量设置为 2. 本文每个实验重复进行了 5 次, 去掉最低值和最高值, 报告剩余三个结果的平均值. 本文每次实验开始前都有一个 60 秒的预热阶段, 接着是 60 秒的数据收集阶段.

7.2 优化效果测试



(a)D3C 的事务吞吐量随优化变化情况　　(b)D3C 的线程平均空闲时间随优化变化情况

图 11 D3C 优化效果测试

这一节在 YCSB 负载下通过消融实验验证本文提出的事务优化方法. 在该实验下, 缓存行数量设置为 300 万, 较大的缓存行数量可以提高性能, 从而充分暴露系统的性能瓶颈.

在图 11 中,基线表示不使用任何本文提出的事务优化方法, 即立即持久化事务执行结果, 不进行只读优化, 仅单调度器的性能情况. "+O1"表示使用基于多版本的异步批量数据持久化机制, "+O1+O2"表示在"+O1"的基础上增加基于数据一致性点的只读事务优化, "+O1+O2+O3-x"表示在"+O1+O2"的基础上增加基于多调度器的读写事务优化, 其中"x"所示的数字表示增加的调度器个数.

从图 11.a 中可以看出, D3C 的性能随着本文提出的事务优化方法的增多而提升. 性能在所有事务优化方法均启用, 且调度器设置为 2 时, 达到最优. 值得注意的是, 进一步提高调度器数量并没有进一步提升 D3C 的事务吞吐量, 反而导致事务吞吐量有所下降.

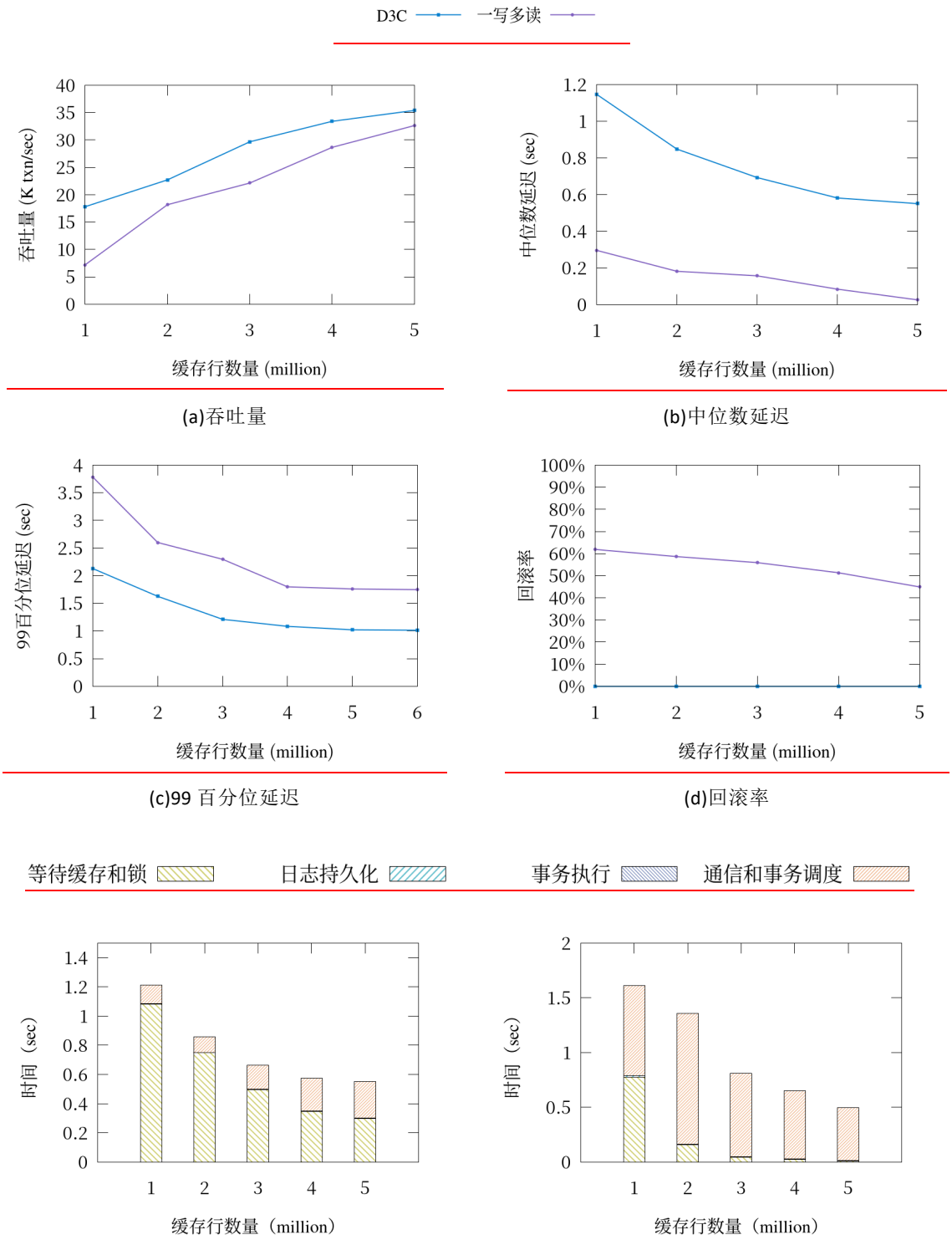
图 11.b 展示 D3C 的各个组件的线程的空闲时间, 随着优化增加而发生的变化. 从图 11.b 中可以看出, 在没有应用基于多调度器的读写事务优化时, 调度器的空闲时间几乎为 0. 此时, 调度器是整个系统的性能瓶颈. 系统的事务吞吐量受限于调度器数量. 在增加调度器的数量之后, 调度器立即获得了大量的空闲时间,此时, 定序器的空闲时间大量下降, 并成为系统内新的性能瓶颈. 执行器的线程的平均空闲时间则并未明显受到调度器数量增加的影响.

7.3 TPC-C测试

这一节在 TPC-C 工作负载下比较 D3C 和一写多读在不同的缓存行数量和仓库数量下的事务吞吐情况.

TPC-C 缓存行数量测试. 从图 12.a 中可以看出, D3C 的事务吞吐量始终高于一写多读, 事务吞吐量的差异在缓存行数量为 100 万时最大, 此时 D3C 的性能是一写多读的 2.47 倍. 这是因为一写多读的单个计算节点需要访问所有的数据, 而 D3C 的单个计算节点只需访问自己负责的数据分片的数据.因此两种架构对缓存的需求存在差异. 当缓存行数量较小时, 一写多读更容易出现缓存未命中的情况, 因而需要更频繁的访问存储节点, 并进行缓存的换入换出操作.

随着缓存行数量增加, D3C 和一写多读均能有明显的性能提升, 但提升幅度逐渐减小, 一写多读的性能提升幅度高于 D3C. 这是因为当缓存容量提升时, 一写多读的缓存命中率增长更多, 因此性能也有更大幅度的提高. 缓存命中率的增长会随着缓存容量的提升而减缓, 性能的提升幅度也因此减小. 图 12.b 和图 12.c 的中位数延迟和 99 百分位延迟上可以观察到类似趋势, 随缓存行数量增加, 延迟逐渐下降且下降幅度逐渐减小.



(a)吞吐量 (b)中位数延迟

(c)99 百分位延迟 (d)回滚率

(e)D3C 事务延迟分析 (f)一写多读事务延迟分析

图 12 TPC-C 缓存行数量测试

值得注意的是, 图 12.d 中一写多读的回滚率呈现出了一直下降的趋势. 这是因为当一个事务某个数据项未命中缓存时, 需要从存储节点获取数据. 此时其他事务访问该数据项, 也会同时等待该缓存行. 当该缓存行就绪时, 可能会产生多个事务同时读写同一数据项的调度, 导致事务回滚. 因而, 缓存行数量会影响事务的回滚率, 从而进一步影响事务吞吐.

此外, 因为 D3C 采用了基于确定性并发控制的云原生数据库事务处理机制, 事务不会因为并发控制而回滚, 所以在回滚率图中, D3C 的事务回滚率始终为 0.

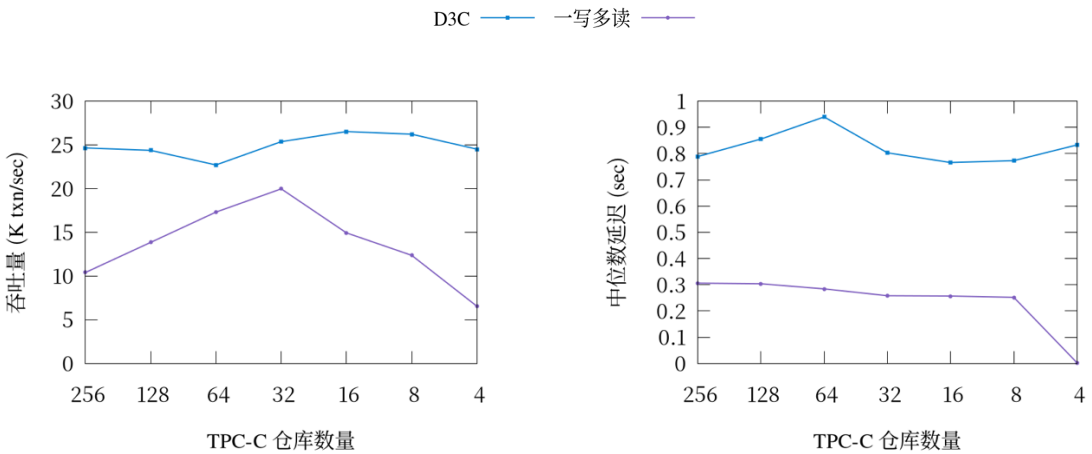
图 12.e 和图 12.f 分别展示了 D3C 和一写多读的平均事务延迟, 以及延迟的组成. 由于对于 D3C, 等待锁和等待数据缓存的时间是重合的, 因此, 这两个时间合并在一个类别中. 一写多读只会在这个类别中只会等待数据缓存. 值得注意的是, D3C 的平均事务延迟和中位数延迟基本一致. 而一写多读由于采用了 Silo 作为并发控制, 回滚的事务会显著提高系统内的尾延迟, 因此, 平均延迟较中位数延迟更高. 从这两个图中可以看出, D3C 和一写多读的延迟都随着缓存行数量的增加而减少. 减少的时间均来源于等待缓存和锁的时间减少.

TPC-C 仓库数测试. 在 TPC-C 负载当中, 事务之间的冲突率会随着 TPC-C 的仓库数减少而增大. 因此在本测试中, 本文探究了 TPC-C 仓库数量减少, 带来的事务冲突率变化对一写多读与 D3C 吞吐量、延迟以及回滚率的影响. 实验数据以系统总仓库数量从 256 逐渐减小到 4 的形式呈现.

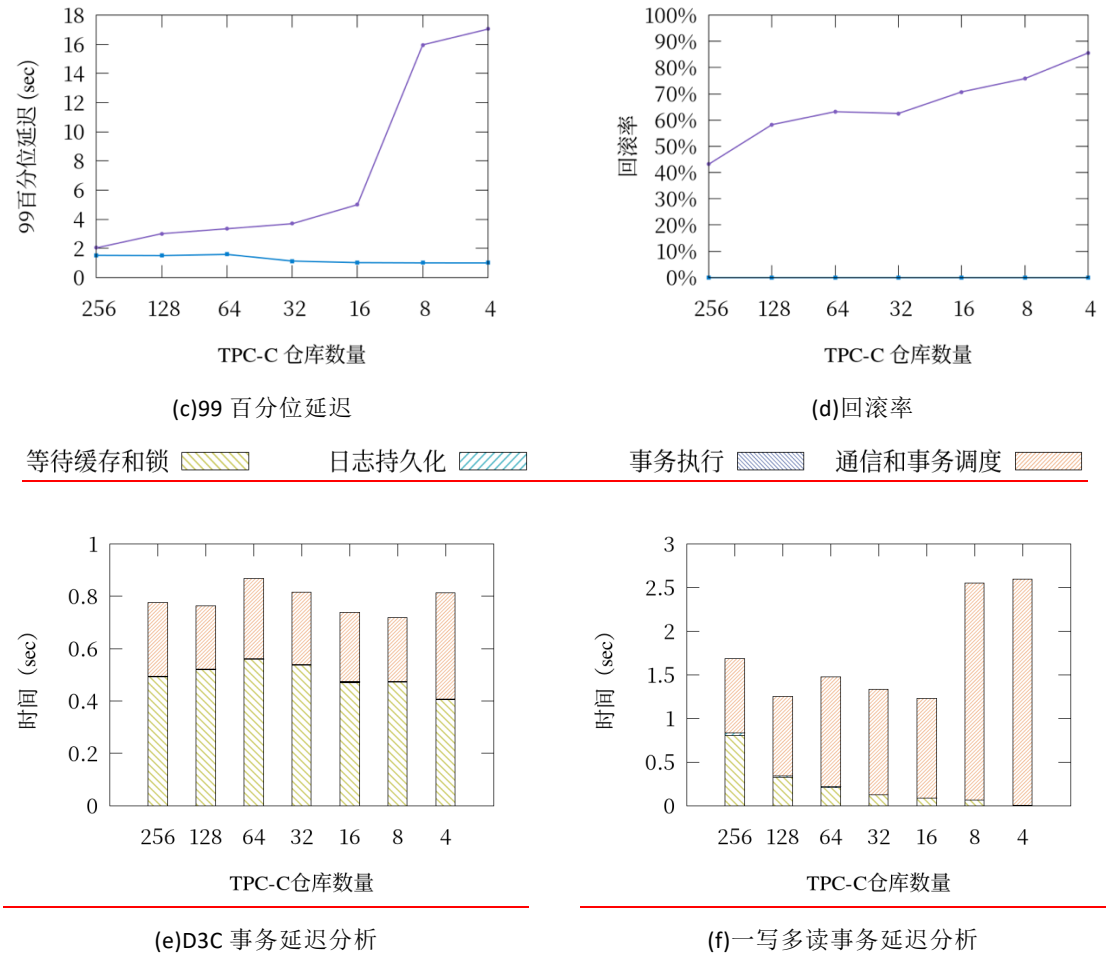
从图 13.a 看出, 随着 TPC-C 仓库数量从 256 减少至 32, 一写多读的吞吐量逐渐提高, 这是因为随着仓库数量减少, 一写多读中命中缓存的概率提升; 当仓库数量进一步减少时, 一写多读的吞吐量开始下降. 如图 13.d 中所示, 随着仓库数的不断减少, 一写多读的回滚率不断提升. 当仓库数少于 16 时, 此时一写多读中缓存命中率带来的性能提升并不能抵消大量事务回滚带来的性能下降.

而 D3C 的吞吐量始终趋于稳定, 因为 D3C 不会因为事务冲突而回滚, 同时受到缓存命中率影响低, 因此性能维持稳定. 图 13.b 和图 13.c 中的 D3C 事务延迟也证明了这一点.

在图 13.b 一写多读的中位数延迟在仓库数从 8 到 4 时存在一个明显的下降, 从图 13.f 中可以看出, 一写多读等待缓存的时间在随着仓库数量的上升而下降, 而通信和事务调度时间随着仓库数量的上升而上升, 此时事务调度时间主要为事务回滚带来的开销. 在仓库数为 4 时, 缓存已经可以在单个节点上缓存全部的数据, 事务不再需要访问存储节点来获得数据项.



(a)吞吐量 (b)中位数延迟



(c)99 百分位延迟 (d)回滚率

(e)D3C 事务延迟分析 (f)一写多读事务延迟分析

图 13 TPC-C 仓库数量测试

7.4 YCSB测试

这一节在 YCSB 工作负载下比较 D3C 和一写多读在不同的倾斜率与写读比场景下的事务吞吐情况.

YCSB 倾斜率测试. 在该测试中, 本文通过调整 YCSB 负载的倾斜率参数从 0.1 到 0.99 来观察一写多读和 D3C 的吞吐量变化.

从图 14.a 中, 当倾斜率小于等于 0.4 时, 一写多读吞吐量保持基本稳定; 而当倾斜率高于 0.4 小于 0.7 时, 一写多读性能开始缓慢提升, 这是因为随着倾斜率的提升, 一写多读中的缓存命中率会变高, 从图 14.f 中也体现出一写多读因为缓存命中率提升, 等待缓存的时间减少, 从而事务执行的延迟下降; 而当倾斜率进一步提升到 0.7 以上时, 一写多读吞吐量急剧下降, 这是因为倾斜率过高导致大量事务回滚, 从图 14.d 中可以看出当倾斜率大于 0.7 时, 回滚率明显上升; 且图 14.f 也可以看出事务执行几乎全部的时间都被用于事务调度, 这是事务回滚从而产生的调度开销. 值得一提的是, 14.f 的平均延迟和 14.b 的中位数延迟呈现出的趋势不完全相同. 在中位数延迟中, YCSB 倾斜率在 0.8 时仍然处于低位, 直到 0.9 时才大量增长. 这个趋势不一致的原因可以通过图 14.d 来解释. 注意, 图 14.d 是该实验下提交事务未曾回滚比率, 即已提交事务中未曾发生过回滚的事务的数量除以提交事务的总数. 从图中可以看到, 这一比例在 YCSB 倾斜率为 0.8 时仍然很高, 0.9 时则快速降低至 50%. 这意味着, 在 YCSB 倾斜率为 0.8 时, 中位数延迟统计到的是未曾回滚过的提交事务的延迟, 该

事务的延迟因缓存命中率足够高而非常低, YCSB 倾斜率为 0.9 时, 中位数延迟统计到的则是在提交前回滚过的事务, 该事务存在较长的事务调度时间. 与之前 TPCC 实验时的原因相同, D3C 的吞吐并不会随着倾斜率的变化而变化. 而从图 14.e 中也可以看出, D3C 的事务的平均时延以及各部分占比并未发生太大变化.

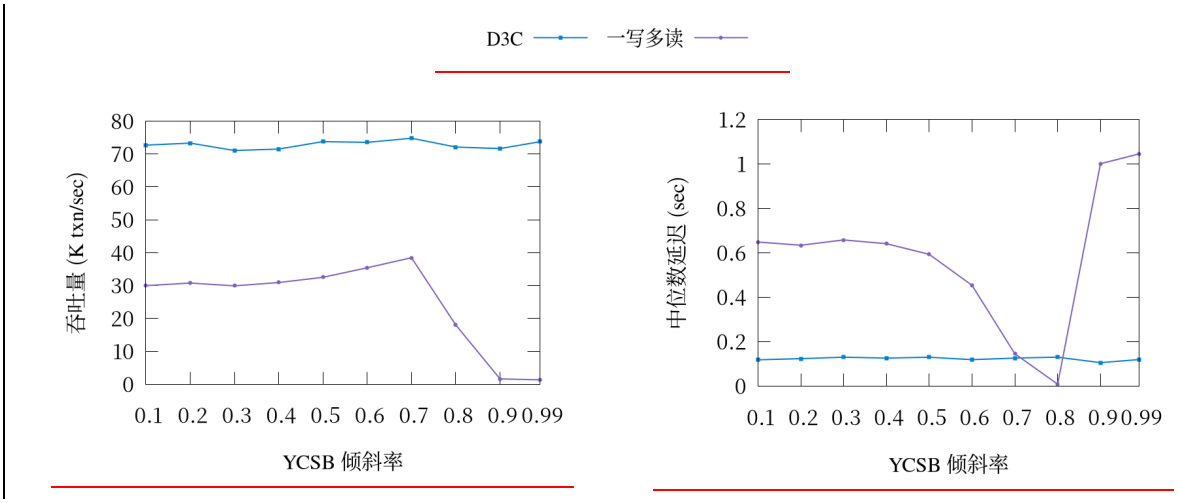
注意, 图 14.a 中一写多读吞吐量在 0.9 后剧烈的下降, 是因为本文使用了闭环测试的方法. 在该测试方法下, 计算节点每完成一个事务, 客户端才将下一事务发送给计算节点. 在 YCSB 倾斜率大于 0.9 时, 读写节点大量回滚读写事务, 将大量的读写事务累积在了读写节点中. 此时客户端必须等读写节点提交一个事务后才会发送下一个读写事务或只读事务. 因此, 在本实验中, 当 YCSB 倾斜率大于 0.9 后, 读写节点回滚率高, 事务吞吐量低, 而只读节点又必须等待读写节点提交事务才能继续执行只读事务, 是系统吞吐量降低的主要原因.

YCSB 写读比测试. 在该测试中, 本文通过调整 YCSB 负载的写读比参数从 0 到 1 来观察一写多读和 D3C 的吞吐量变化.

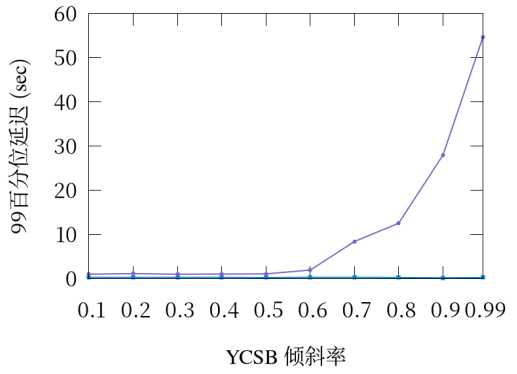
在图 15.a 中, 随着写读比的增加, 一写多读的吞吐量逐渐下降, 这个是因为一写多读方案中只有一个读写节点可以执行读写事务, 随着读写事务比例的增加, 只读事务逐渐变少, 一写多读中的只读节点无法执行读写事务, 并发执行的事务数逐渐减少, 从而导致性能下降. 而 D3C 通过设计基于确定性并发控制的事务处理机制以支持多写, 因此性能维持稳定. 同时由于文中提出的基于数据一致性点的只读事务优化, D3C 的吞吐量在只读场景, 即写读比为 0 时, 也能达到一写多读吞吐量的 2.3 倍.

在图 15.b 中, 一写多读的中位数延迟在写读比从 0.4 增长到 0.6 时, 发生了一次显著的提升, 这是因为当写读比为 0.4 时, 系统内的只读事务数量仍然高于读写事务数量, 因此此时系统内只读节点和读写节点之间能够做到负载均衡. 而当写读比提升到 0.6 时, 系统不再能在只读节点和读写节点之间做到负载均衡, 读写节点的负载压力提高了事务延迟. 从图 15.f 中也可以看出, 随着读写节点同时执行的事务数增多,延迟仍然主要增加在了等待缓存的时间上.

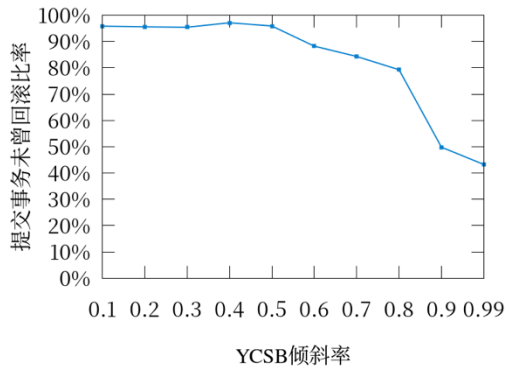
在图 15.c 中, 99 百分位延迟在读写比从 0 增长到 0.2 时也发生了显著的提升, 这是因为在读写比为 0 时, 系统内只有只读事务, 而读写比为 0.2 时, 系统内出现读写事务, 只读事务的延迟较读写会更低, 因而读写事务提高了整体的延迟.



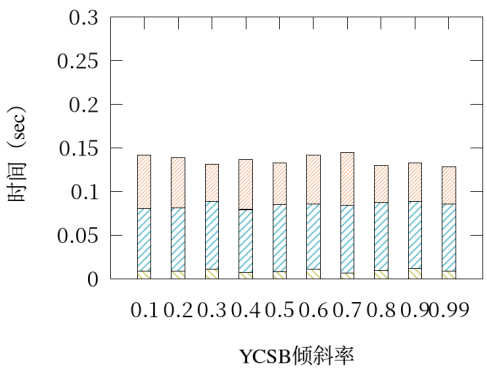
(a)吞吐量 (b)中位数延迟



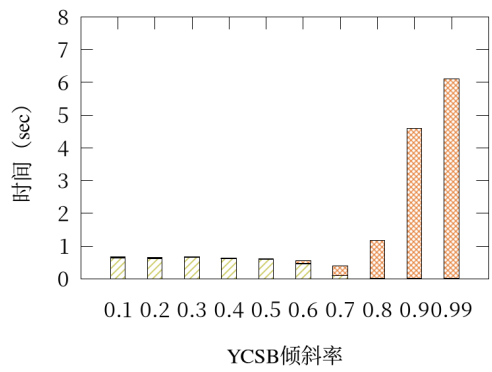
(c)99 百分位延迟



(d)提交事务未曾回滚比率

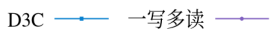


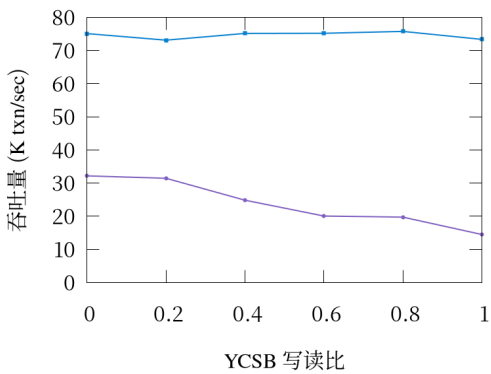
(e)D3C 事务延迟分析



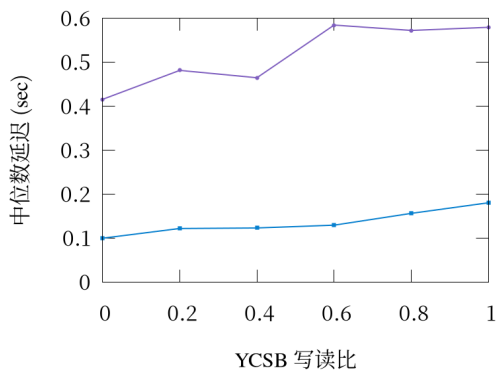
(f)一写多读事务延迟分析

图 14 YCSB 倾斜率测试

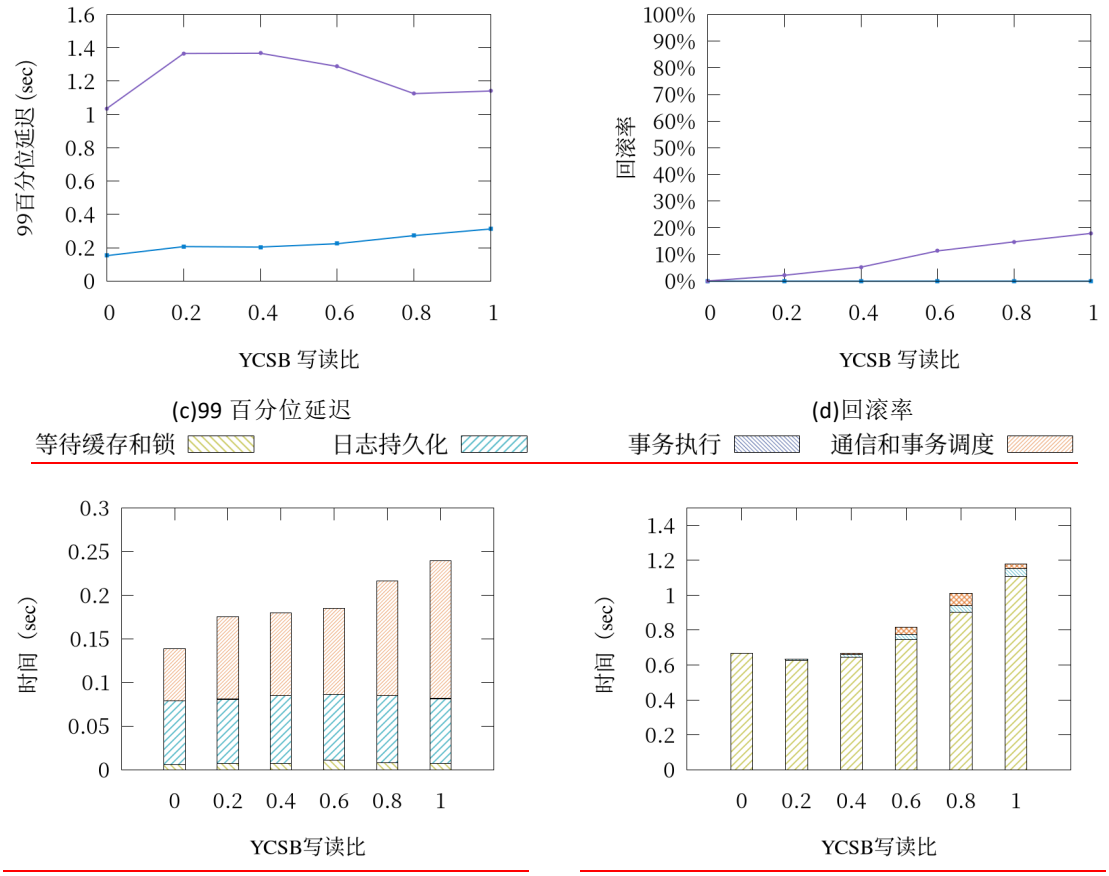




(a)吞吐量



(b)中位数延迟



(c)99 百分位延迟 (d)回滚率

(e)D3C 事务延迟分析 (f)一写多读事务延迟分析

图 15 YCSB 写读比测试

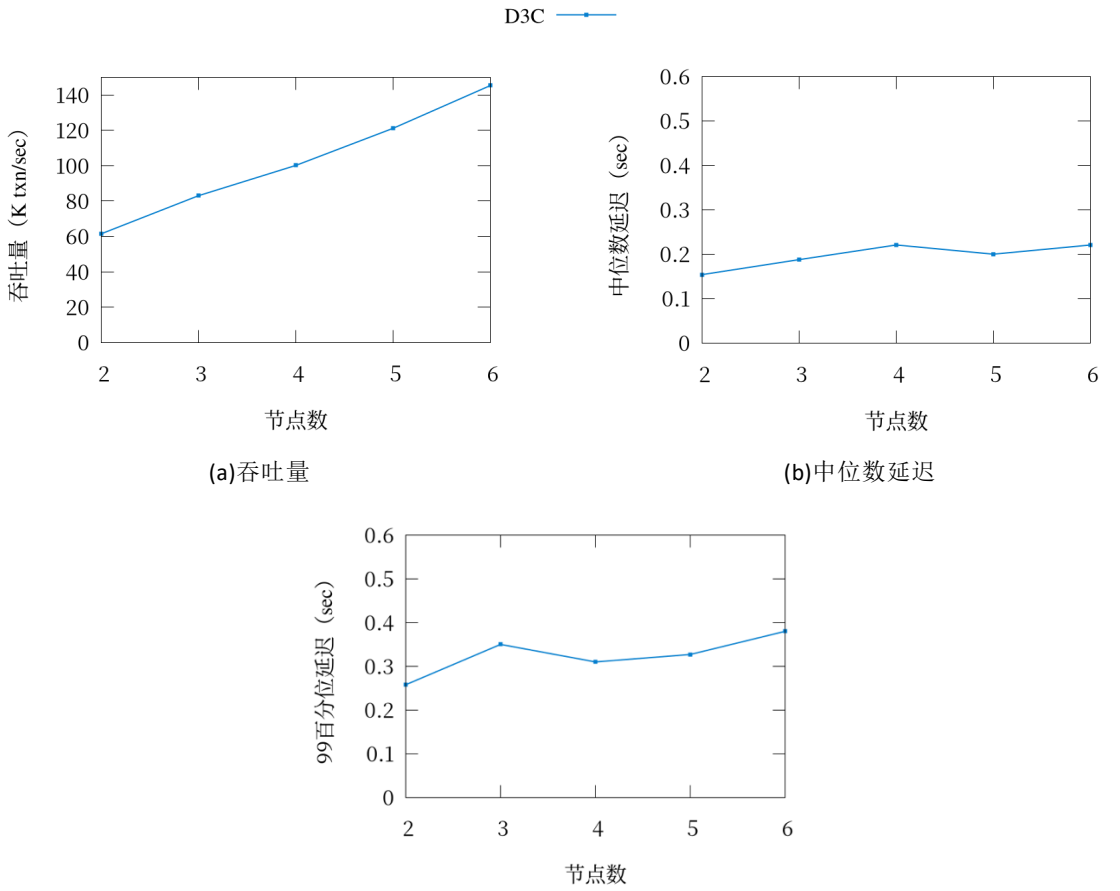
7.5 可扩展性

在该测试中, 本文通过调整节点数量在 YCSB 负载下测试了 D3C 的可扩展性. 从图 16 上可以看到, D3C 的事务随着节点数的增多线性可扩展, 且事务延迟基本保持不变. 当节点数从 2 增长到 6 时, 系统性能增长约 2.4 倍。因为 D3C 采用了基于确定性并发控制的云原生数据库事务处理机制, 事务不会因为并发控制而回滚, 因而本文在该实验中省略了回滚率图.

8 总结和展望

本文针对云原生数据库中一写多读的限制, 提出了 D3C 云原生数据库架构, 并参考确定性的思路设计了基于确定性并发控制的云原生数据库事务处理机制. 通过将事务拆分为子事务, 并根据预先确定的全局顺序在各节点独立执行, 以保证多个读写节点上读写事务同时执行, 实现多写机制. 此外, D3C 设计了一种基于多版本的异步批量数据持久化机制来实现高效的数据持久化, 以保证事务执行的效率. 同时, D3C 还额外提出了基于多调度器的读写事务优化和基于数据一致性点的只读事务优化两种优化策略, 进一步提升 D3C 中事务处理的性能. 最后, D3C 还提出了基于一致性点的故障恢复机制确保了在各种故障环境下数据的一致性和事务的正确性. 实验表明, D3C 在满足云原生数据库关键性能指标的同时, 在写密集场景下能达到一写多读 5.1 倍的吞吐.

本文存在两个未来的工作. 一方面, 消除事务的处理需要提前得知事务的读写集的前提假设是非常重要的. 尽管目前已经存在一些技术来缓解这个问题, 然而如果事务在实际执行时, 其读写集发生了变化, 那么事务必须重新执行侦查查询. 如何消除必须已知事务读写集的前提假设将是本文一个重要的未来工作. 另一方面, 尽管本文指出调度器是系统性能的瓶颈, 并提出了基于多调度器的读写事务优化来优化系统性能. 然而在解决调度器的瓶颈之后, 定序器将成为新的系统瓶颈. 如何使得节点能够最大化的提升执行效率, 协调节点内不同组件的线程数量, 将是本文的另一项未来工作.



(a)吞吐量 (b)中位数延迟

(c)99 百分位延迟

图 16 可扩展性测试

References:

[1] Global database management system (DBMS) market outlook. 2023. https://www.expertmarketresearch.com/reports/database-management-system-market

[2] Database-as-a-service (DBaaS): Global strategic business report. 2023. https://www.researchandmarkets.com/reports/4804281/databaseas-a-service-dbaas-global-strategic

[3] Database Development Research Report. 2023. (in chinese) https://13115299.s21i.faiusr.com/61/1/ABUIABA9GAAgrrmOpQYojvvn7AQ.pdf

[4] Verbitski A, Gupta A, Saha D, Brahmadesam M, Gupta K, Mittal R, Krishnamurthy S, Maurice S, Kharatishvili T, Bao X. Amazon aurora: Design considerations for high throughput cloud-native relational databases. Proceedings of the 2017 ACM International Conference on Management of Data. 2017: 1041-1052.

[5] Cao W, Zhang Y, Yang X, eLi F, Wang S, Hu Q, Cheng X, Chen Z, Liu Z, Fang J, Wang B, Wang Y, Sun H, Yang Z, Cheng Z, Chen S, Wu J, Hu W, Zhao J, Gao Y, Cai S, Zhang Y, Tong J. Polardb serverless: A cloud native database for disaggregated data centers[. roceedings of the 2021 International Conference on Management of Data. 2021: 2477-2489.

[6] Antonopoulos P, Budovski A, Diaconu C, Hernandez Saenz A, Hu J, Kodavalla H, Kossmann D, Lingam S, Farooq Minhas U, Prakash N, Purohit V, Qu H, Sreenivas Ravella C, Reisteter K, Shrotri S, Tang D, Wakade V. Socrates: The new sql server in the cloud. Proceedings of the 2019 International Conference on Management of Data. 2019: 1743-1756.

[7] Depoutovitch A, Chen C, Chen J, eLarson P, Lin S, Ng J, Cui W, Liu Q, Huang W, Xiao Y, He Y. Taurus database: How to be fast, available, and frugal in the cloud. Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020: 1463-1478.

[8] Cao W, Liu Z, Wang P, Chen S, Zhu C, Zheng S, Wang Y, Ma G. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. Proceedings of the VLDB Endowment, 2018, 11(12): 1849-1862.

[9] Bernstein P A, Hadzilacos V, Goodman N. Concurrency control and recovery in database systems. Boston: Addison-wesley, 1987.

[10] Zhang M, Hua Y, Zuo P, Liu L. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. 20th USENIX Conference on File and Storage Technologies (FAST 22). 2022: 51-68.

[11] Wei X, Dong Z, Chen R, Chen H. Deconstructing RDMA-enabled distributed transactions: Hybrid is better!. 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018: 233-251.

[12] Binnig C, Crotty A, Galakatos A, Kraska T, Zamanian E. The end of slow networks: It's time for a redesign. Proc. VLDB Endow. 9, 7 (March 2016), 528–539.

[13] Wei X, Shi J, Chen Y, Chen R, Chen H. Fast in-memory transaction processing using RDMA and HTM. Proceedings of the 25th Symposium on Operating Systems Principles. 2015: 87-104.

[14] Chen Y, Wei X, Shi J, Chen R, Chen H. Fast and general distributed transactions using RDMA and HTM. Proceedings of the Eleventh European Conference on Computer Systems. 2016: 1-17.

[15] Barthels C, Müller I, Taranov K, Alonso G, Hoefler T. Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. Proceedings of the VLDB Endowment, 2019, 12(13): 2325-2338.

[16] Yoon D Y, Chowdhury M, Mozafari B. Distributed lock management with RDMA: decentralization without starvation. Proceedings of the 2018 International Conference on Management of Data. 2018: 1571-1586.

[17] Gray J, Reuter A. Transaction Processing: Concepts and Techniques. Burlington: Morgan Kaufmann, 1992, 9(3):466-473.

[18] Wang S, Sa S. Introduction to Database System[M]. 5th edition. Beijing: Higher Education Press, 2014

[19] Ziegler T, Binnig C, Leis V. ScaleStore: A fast and cost-efficient storage engine using DRAM, NVMe, and RDMA. Proceedings of the 2022 International Conference on Management of Data. 2022: 685-699.

[20] Yang X, Zhang Y, Chen H, Li F, Wang B, Fang J, Sun C, Wang Y. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. Companion of the 2024 International Conference on Management of Data. 2024: 295-308.

[21] Depoutovitch A, Chen C, Larson P A, Ng J, Lin S, Xiong G, Lee P, Boctor E, Ren S, Wu L, Zhang Y, and Sun C. 2023. Taurus MM: Bringing Multi-Master to the Cloud. Proceedings of the VLDB Endowment, 2023, 16(12): 3488-3500.

[22] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. 1988. Granularity of locks and degrees of consistency in a shared data base. Readings in database systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 94–121.

[23] Bernstein P A, Goodman N. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. VLDB. 1980: 285-300.

[24] Yu X, Xia Y, Pavlo A, Sanchez D, Rudolph L, Devadas S. Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system. Proceedings of the VLDB Endowment, 2018, 11(10): 1289-1302.

[25] Tu S, Zheng W, Kohler E, Liskov B, Madden S. Speedy transactions in multicore in-memory databases. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. 2013: 18-32.

[26] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2014. MaaT: effective and scalable coordination of distributed transactions in the cloud. Proc. VLDB Endow. 7, 5 (Janary 2014), 329–340.

[27] Lim H, Kaminsky M, Andersen D G. Cicada: Dependably fast multi-core in-memory transactions. Proceedings of the 2017 ACM International Conference on Management of Data. 2017: 21-35.

[28] Berenson H. Bernstein P, Gray J, Melton J, O'Neil E, O'Neil P. A critique of ANSI SQL isolation levels. ACM SIGMOD Record, 1995, 24(2): 1-10.

[29] 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). http://www.jos.org.cn/1000-9825/6454.htm

[30] Fekete A, Liarokapis D, O'Neil E, O'Neil P, Shasha D. Making snapshot isolation serializable. ACM Transactions on Database Systems (TODS), 2005, 30(2): 492-528.

[31] Cahill M J, Röhm U, Fekete A D. Serializable isolation for snapshot databases. ACM Transactions on Database Systems (TODS), 2009, 34(4): 1-42.

[32] Thomson A, Diamond T, Weng S, Ren K, Shao P, Abadi D. J. Calvin: fast distributed transactions for partitioned database systems. Proceedings of the 2012 ACM SIGMOD international conference on management of data. 2012: 1-12.

[33] Faleiro J M, Abadi D J. 2015. Rethinking serializable multiversion concurrency control. Proc. VLDB Endow. 8, 11 (July 2015), 1190–1201.

[34] Qin D, Brown A D, Goel A. Caracal: Contention management with deterministic concurrency control. Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021: 180-194.

[35] Faleiro J M, Abadi D J, Hellerstein J M. High performance transactions via early write visibility. Proceedings of the VLDB Endowment, 2017, 10(5).

[36] Nathan S, Govindarajan C, Saraf A, Sethi M, Jayachandran P. Blockchain meets database: Design and implementation of a blockchain relational database. Proc. VLDB Endow. 12, 11 (July 2019), 1539 – 1552.

[37] Dong Z, Tang C, Wang J, Wang Z, Chen H, Zang B. Optimistic transaction processing in deterministic database. Journal of Computer Science and Technology, 2020, 35: 382-394.

[38] Lu Y, Yu XY, Cao L, and Madden S. 2020. Aria: a fast and practical deterministic OLTP database. Proc. VLDB Endow. 13, 12 (August 2020), 2047–2060.

[39] Lai Z, Liu C, Lo E. When private blockchain meets deterministic database. Proceedings of the ACM on Management of Data, 2023, 1(1): 1-28.

[40] Harding R, Van Aken D, Pavlo A, Stonebraker M. An evaluation of distributed concurrency control. Proceedings of the VLDB Endowment, 2017, 10(5): 553-564.

[41] Chen Z, Zhuo H, Xu Q, Qi X, Zhu C, Zhang Z, Jin C, Zhou A, Yan Y, Zhang H. SChain: a scalable consortium blockchain exploiting intra-and inter-block concurrency. Proceedings of the VLDB Endowment, 2021, 14(12): 2799-2802.

[42] Qi X, Chen Z, Zhuo H, Xu Q, Zhu C, Zhang Z, Jin C, Zhou A, Yan Y, Zhang H. SChain: Scalable Concurrency over Flexible Permissioned Blockchain. 2023 IEEE 39th International Conference on Data Engineering (ICDE). IEEE, 2023: 1901-1913.

[43] FaunaDB. 2024. https://fauna.com/

[44] Costa C H, Maia P H M, Carlos F. Sharding by Hash Partitioning - A Database Scalability Pattern to Achieve Evenly Sharded Database Clusters. Proceedings of the 17th International Conference on Enterprise Information Systems. 2015, 1: 313-320.

[45] Venkateswaran N, Changder S. Simplified data partitioning in a consistent hashing based sharding implementation. TENCON 2017-2017 IEEE Region 10 Conference. IEEE, 2017: 895-900.

[46] Cao T, Vaz Salles M, Sowell B, Yue Y, Demers A, Gehrke J, White W. Fast checkpoint recovery algorithms for frequently consistent applications. Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. 2011: 265-276.

[47] Liu M L, Agrawal D, Abbadi A E. An efficient implementation of the quorum consensus protocol. 1994.

[48] Diks K, Pelc A. Almost safe gossiping in bounded degree networks. SIAM Journal on Discrete Mathematics, 1992, 5(3): 338-344.

[49] Mohan C, Haderle D, Lindsay B, Pirahesh H, Schwarz P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems (TODS), 1992, 17(1): 94-162.

[50] TPC-C. [n.d.]. http://www.tpc.org/tpcc/.

[51] Cooper B F, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. Proceedings of the 1st ACM symposium on Cloud computing. 2010: 143-154.

[52] Thomson A, Abadi D J. The case for determinism in database systems. Proceedings of the VLDB Endowment, 2010, 3(1-2): 70-80.

附中文参考文献:

[3] 数据库发展研究报告. 2023. https://13115299.s21i.faiusr.com/61/1/ABUIABA9GAAgrrmOpQYojvvn7AQ.pdf

[18] 王珊, 萨师煊. 数据库系统概论[M]. 第 5 版. 北京: 高等教育出版, 2014.

[29] 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