软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.007279]

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

PG-RAC: 基于 PostgreSQL 的共享缓存多写事务处理数据库 *

印钰杰[1], 史浩洋[1], 范自豪[1], 周华辉[1], 刘晟驰[1], 胡卉芪[1], 魏星[2], 陈河堆[2], 屠要峰[2], 蔡鹏[1], 周 [1]

[1](华东师范大学 数据科学与工程学院, 上海 200062)

[2](中兴通讯, 江苏 南京 210012)

通讯作者: 胡卉芪, E-mail: hqhu@dase.ecnu.edu.cn

摘 要: 云原生数据库的主流设计采用一主多从架构,集群中从节点可以分担主节点的只读请求,写请求由主节点处理.在此基础上,为了进一步满足大规模交易扩展的需求,一些云数据库尝试实现多写事务扩展.多写扩展的一种实现路径是在计算节点间实现共享缓存,支持跨节点的数据访问.在基于共享缓存的数据库系统中,跨节点远程访问的开销远大于本地访问,因此缓存协议的设计是影响系统性能和可扩展性的关键因素.本文对缓存协议提出了两个创新性改进,并基于 PostgreSQL 实现了支持多写事务处理的共享缓存数据库 PG-RAC.一方面,PG-RAC 提出一种新型的分布式链式路由策略,将路由信息分散在各计算节点.相比单点目录管理的路由策略,事务平均延迟降低了约 20%.另一方面,本文还改进了副本页失效机制,将失效操作从事务路径分离,减小了事务处理关键路径的延迟.在此基础上,PG-RAC 利用多版本并发控制的特性,进一步提出推迟副本页失效时机,有效提高了缓存利用率.TPC-C 实验结果显示,在配备 4 台计算节点的集群中,吞吐率为 PostgreSQL 的近 2 倍,为分布式数据库 Citus 的 1.5 倍.

关键词: 云原生数据库;共享缓存数据库;缓存一致性协议;事务处理

中图法分类号: TP311

中文引用格式: 印钰杰, 史浩洋, 范自豪, 周华辉, 刘晟驰, 胡卉芪, 魏星, 陈河堆, 屠要峰, 蔡鹏, 周烜. PG-RAC: 基于 PostgreSQL 的共享缓存多写事务处理数据库. 软件学报. http://www. jos.org.cn/1000-9825/7279.htm

英文引用格式: Yin YJ, Shi HY, Fan ZH, Zhou HH, Liu SC, Hu HQ, Wei X, Chen HD, Tu YF, Cai P, Zhou X. PG-RAC: A Multi-Write Transaction Processing Database with Shared Buffer Based on PostgreSQL. Ruan Jian Xue Bao/Journal of Software. http://www.jos.org.cn/1000-9825/7279.htm

PG-RAC: A Multi-Write Transaction Processing Database with Shared Buffer Based on PostgreSQL

YIN Yu-Jie[1], SHI Hao-Yang[1], FAN Zi-Hao[1], ZHOU Hua-Hui[1], LIU Sheng-Chi[1], HU Hui-Qi[1], WEI Xing[2]

CHEN He-Dui[2], TU Yao-Feng[2], CAI-Peng[1] , ZHOU Xuan[1]

[1](College of Data Science and Engineering, East China Normal University, Shanghai 200062, China)

[2](Zhongxing Telecommunication Equipment Corporation, Nanjing 210012, China)

Abstract: Single-master is the mainstream architecture of cloud-native databases.For Single-master databases,reads can be spread across the cluster,while updates must be handled by one single master node. To meet the demands of large-scale transaction processing, some of the cloud-native databases attempt to further enable transaction execution across multiple nodes. One possible approach is to introduce

———————————————————————

*基金项目: 国家自然科学基金(No.92270202); 上海市自然科学基金(No.23ZR1418300); 中兴通讯研究基金(编号 HC-CN-20220721010)

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

shared cache.It enables cross-node data access and leads to a multi-master database based on shared caching. For shared-cache databases,the overhead of remote access is significantly higher than local access.Therefore,coherence protocol is the crucial factor that affects system performance and scalability. In this paper,we present PG-RAC,a multi-master shared-cache database based on PostgreSQL.This paper proposes two novel improvements to the coherence protocol.First,we propose chained routing strategy,which distributes the routing information acorss nodes in cluster. Compared to the routing strategy that utilizes centralized directory ,it reduces the average transaction latency by approximately 20%.Second,PG-RAC enhances the invalidation mechanism. PG-RAC separates invalidation operations from the transaction path, reducing the latency of the critical path in transaction processing. Additionally, based on the features of MVCC,PG-RAC proposes delaying the invalidation point,which effectively improves cache utilization.TPC-C benchmark results show that,for a cluster with 4 compute nodes, the throughput is 2× that of a single-node PostgreSQL and 1.5× that of the distributed database Citus.

Key words: cloud-native database; shared-cache database; cache coherence;transaction processing

云数据库是一种基于云计算技术的数据库服务模式,它将数据库系统部署在云端的数据中心中,用户可以通过互联网进行访问和管理.目前主流的云原生数据库遵循“计算-存储分离”的设计理念[1,2,3,4,5].存储层是由多个存储节点组成的共享存储,存放数据文件和日志文件.计算层由无状态的计算实例构成,负责 SQL 语句解析、物理计划执行、事务处理等功能.基于“计算-存储分离”特性,用户可以根据负载特点,弹性扩展与伸缩存储、计算的资源规格和容量,有效利用资源.

目前绝大多数的云数据库产品采用一主多从架构,由单个节点处理所有写请求,只读节点分担一部分读请求,如 Amazon Aurora[1].这种架构可以实现只读扩展,但在写密集场景下不具备横向扩展能力.为此,部分数据库厂商尝试对计算层进行改动以支持多写扩展.以 Aurora 多主集群为例[2],事务期间的修改以日志形式下推到存储层,在事务提交时判断是否存在数据的冲突访问,并进行提交或回滚操作.对于写冲突率较高的负载,这种乐观并发控制的方式会带来过高的冲突回滚率,因此性能并不理想[6].

另一种多写扩展的方式是通过网络连接各计算节点的缓冲区,实现一个统一的共享缓存.共享缓冲区和共享存储的存在,为所有计算节点提供一份统一且完整的数据视图.本地缓冲区内的数据访问未命中时,需确定数据所在位置,并进行远程访问,数据拉取到本地缓冲区后,由本地计算实例处理数据.

数据访问操作在整个事务路径中占据较大的比重,而其中对缓冲区的访问通常多于对底层存储的访问,因此缓存层的设计是影响多写事务处理性能的一个关键因素.我们通过图 1 来刻画缓存设计对事务处理的影响：集群内部的中心路由服务维护了全局缓存页的路由表,记录数据页的位置信息.若本地缓冲区内没有 p0 的有效数据,将发起一次远程访问.步骤(1)从路由服务中获取 p0 的实际位置, 随后步骤(2)从远端节点 2 的缓冲区拉取 p0 到本地缓冲区,进行读写操作.若事务修改了数据 p1,将通过步骤(3)通知集群中其余节点,避免后续读取数据不一致.

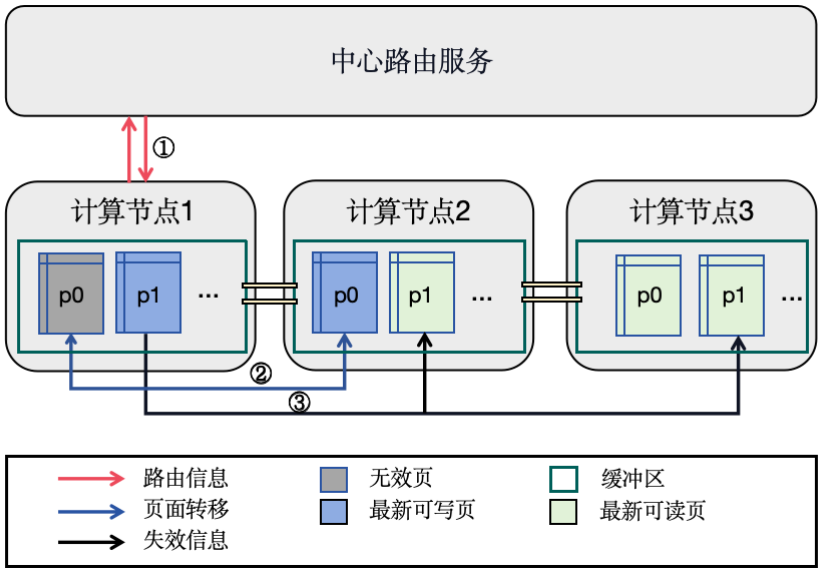

图 1 共享缓存数据库的访问基本流程(彩印,office ppt 绘制)

由此可见，缓存层设计影响事务多写处理的因素可概括为两方面：

(1)路由方式. 本地缓冲区内的数据访问未命中时,节点需要确定数据所在位置.如图 1 所示,在共享缓存的典型设计中,由中心路由服务记录数据的位置信息,并负责数据路由.这一设计参考了多核体系结构中基于目录的缓存一致性协议,易于实现.但是中心路由的设计存在两个问题.一方面,在由多台机器构成的集群中,中心化目录单点服务很有可能成为全局瓶颈.另一方面,一次远程读写需要访问目录并对数据页上锁,随后访问目标节点获取数据,最后再次访问目录解锁.总共涉及三次往返通信,这会引入过高的通信开销.

(2)数据一致性维护. 数据一致性维护包括两个部分,一是验证本地缓冲区内的副本是否失效,二是向集群同步本地发起的修改,图 1 的 2、3 两个步骤描述了这个过程：根据缓存一致性协议,如果本地页面被判定为失效,需发起页面转移请求,获取最新页面；如果本地修改了某页面,需要通知其他节点该页面已经失效.由于分布式场景下远程访问的代价远高于本地访问,因此在满足一致性的前提下,系统应该尽可能减小数据同步的开销.

针对上述的两个问题,本文提出了 PG-RAC,一个以 PostgreSQL 作为基础的高性能共享缓存数据库.为了满足多写事务处理的高性能,本文基于单节点 PostgreSQL 数据库缓冲区和并发控制机制,将其扩展为多节点共享存储和共享缓冲区的架构,并对共享缓冲区的缓存机制和事务并发控制机制提出了多项优化.PG-RAC 满足事务的快照隔离级别.

PG-RAC 在缓存一致性协议的创新包括两部分.首先,PG-RAC 提出了去中心化的链式路由机制,链式路由根据分散在各计算节点的路由表,得到数据页的跳转路径,最终追溯到数据实际位置.这一机制将单点目录的访问压力分散到各计算节点,并减少了远程访问的平均通信次数,有效提升了系统的可扩展性.

其次,为了减少数据同步带来的开销,本文提出了优化的副本失效机制.PG-RAC 将失效操作从事务路径中分离,有效减小了事务持有资源的时间.在此基础上,本文还设计了与多版本并发控制结合的缓存一致性协议,推迟副本页失效时机,极大地提升了系统性能和扩展性.

我们将本文的贡献概括为以下几点：

(1)本文提出去中心化的链式路由,并且通过优化缩短了路由路径,避免单点瓶颈和过高的通信开销,从而显著提升了系统整体性能.

(2)本文设计并实现了副本页失效机制.将失效操作从事务路径分离,消除了事务提交时的同步失效等待操

作.结合多版本并发控制的机制,本文提出了副本页延迟失效.在保证一致性的前提下,尽可能降低远程访问开销.

(3)我们实现了一个以 PostgreSQL 为代码基础的高性能共享缓存数据库 PG-RAC,并对 PG-RAC 作了详尽的实验评估.实验结果显示,PG-RAC 具备良好的性能表现：对于配备 4 台计算节点的集群,TPC-C 测试的每分钟处理订单数量为 PostgreSQL 的近 2 倍,为分布式数据库 Citus 的 1.5 倍

本文的结构如下.第一章描述背景及相关工作.第二章介绍系统的整体架构.第三章和第四章将详细展开分布式链式路由机制和副本页失效机制.第五章进行实验评估.第六章总结本文.

1 背景与相关工作

1.1 基于共享缓存的分布式数据库

基于共享缓存的分布式数据库允许多个数据库实例访问共享存储,实现数据库的水平扩展与负载均衡.将数据库工作负载分布到多个节点或实例上,有效地利用集群中的资源,提高数据库的性能和响应速度.现有的共享缓存数据库有 Oracle Real Application Cluster（RAC）以及 Azure DB2 purescale 等[7,8,9].Oracle RAC 使用全局缓存服务和全局排序实现内存融合技术,前者管理缓存一致性以及数据节点间传递,后者负责多节点事务并发锁管理.通过内存融合,保证计算节点在获取最新版本数据块使用权时进行修改,获取数据分布图,获知数据版本,完成传递过程,在锁的保护下并发访问资源[10].DB2 purescale 中全局缓冲池（GBP）作为中心节点在作为数据路由目录的同时保存具体数据页缓存,在节点本地不命中时访问中心节点即有成功命中,减少从主节点传递数据的开销.

表 1 对比了 Oracle RAC,DB2 purescale 和 PG-RAC 在整体架构、路由机制和页面失效机制上的区别.

(1)整体架构. PG-RAC 与 Oracle RAC 在整体架构上相似,主页分布于整个集群,各计算节点的缓冲区共同组成全局缓冲区.而 DB2 purescale 由 GBP 服务缓存所有主页,GBP 相当于一个全局的中心内存池,提供内存扩展.

(2)路由机制. Oracle RAC 采用中心化目录路由,查询主页所属位置全部经过全局缓存服务.DB2 purescale 访问数据全部通过 GBP,无需路由服务.PG-RAC 采用分布式链式路由机制,消除单点瓶颈,将路由压力分散到各计算节点,第三章将详细展开.

(3)失效机制. Oracle RAC 和 DB2 purescale 在事务路径内执行缓存失效.PG-RAC 提出优化的副本页失效机制,将失效操作从事务路径中分离,并且延迟副本页失效时机,将在第四章详细展开.

共享缓存数据库	整体架构	路由机制	失效机制
Oracle RAC	分布式共享缓冲区	中心化目录服务	事务内失效
DB2 purescale	全局中心内存池	访问全局缓冲池,无需路由	事务内失效
PG-RAC	分布式共享缓冲区	分布式链式路由	事务路径外失效+延迟失效

表 1 基于共享缓存分布式数据库对比

1.2 无共享架构的分布式数据库

无共享架构是一种广泛应用于分布式数据库的扩展模型。在这种架构中，每个请求由集群内的一个单独节点独立处理，主要目的是消除节点之间的争用，充分发挥分布式集群的并行处理能力。无共享架构数据库通常将数据分区到多个节点，每个节点对其分区中的数据具有独占访问权限。当请求被有效划分到单一分区时，无共享架构可以提供强大的扩展能力。然而如果一个事务跨越多个分区，则需要分布式事务处理来维护事务一致性，分布式事务的开销极大抵消了并行处理带来的性能优势。此外，当集群需要扩展或收缩时，可能需要对数据进行重新分区，这个过程将引发数据迁移，对系统的运行造成影响。总的来说，尽管无共享架构为系统可扩展性提供了显著的优势，但这些优势也带来了一系列挑战和复杂性。

有诸多知名的分布式数据库采用无共享架构。例如，Citus是一个基于PostgreSQL的无共享架构数据库，由Citus Data于2016年开发并开源。它的设计目标是在保持PostgreSQL的兼容性、易用性和可扩展性的基础上，提供高性能、高可用性和分布式处理能力。Citus通过将数据分片存储在多个节点上，并行执行事务和查询，从而实现了对大规模数据的高效处理。Citus的核心技术包括数据分片、分布式事务管理和分布式查询处理等。

1.3 云原生数据库

传统数据库直接迁移上云存在着资源浪费,数据更新写放大的问题,无法充分利用云服务弹性扩展,成本控制灵活的特性.由于计算资源与存储资源绑定,在两者需求不平衡时,为避免出现瓶颈,将导致其他资源过度申请产生浪费.并且,传统数据库在数据更新时通过将脏页写回的方式,存在着写放大的问题.尤其在云服务中,页面写回经由网络,进一步加剧写放大带来的开销.针对这些问题,云原生数据库[1,11,12,13,14,15,16]应运而生.云原生数据库基于计算-存储分离架构,计算资源与存储资源独立管理、调度与扩展,提高资源利用率；提出“日志即数据”的思想,只下刷日志到磁盘,通过回放日志代替写回脏页,减少网络传输的数据量,解决写放大问题.典型的云原生数据库有 Amazon Aurora、华为 Taurus、Azure Socrates 及阿里巴巴 PolarDB-X 等.

Aurora 最早提出“日志即数据”这一革命性观点,将日志下沉,在存储层后台异步回放日志进行页面更新.日志重做取代页面传输,既避免逻辑随机写与物理按页写入带来的写放大问题,又避免重复日志与页面的双重持久化过程.并且异步回放日志,只需将日志持久化即可完成写入,数据页的更新及副本间复制并不会同步阻塞写入.Taurus 与 Socrates 在两层架构下进一步将存储层模块化,分离日志与数据页存储.在使用日志回放取代数据页写回的方式下,日志写入速度为影响读写性能的关键因素,而数据页只需要存储记录即可,对读写速度不敏感.考虑这一点,细分日志与数据存储层,更加充分利用灵活的云服务满足不同的写入要求[17].Taurus 基于华为分布式存储服务,组织多种 SSD 提供分布式存储服务,将上层计算节点日志操作以及数据页操作抽象到存储层分别处理,自动适应性地选择合适的物理存储,提供高效的存储层服务.而 Socrates 在计算层与存储层之间单独设计日志服务层,用于管理日志的持久化、传递以及回放.基于日志下沉存储层回放可以带来写入时的性能提升,但是读取时在计算节点缓存落后的情况下需要从远程持久化存储获取数据页,读取的性能较差.所以 PolarDB Serverless[18]将计算资源与内存资源解耦,通过高速 RDMA 网络连接多个计算节点,构建远程内存池,为存储层增加一层缓存.PolarDB SCC 同样利用 RDMA 低延迟传输的特性加速日志传输,支持更低延迟的强一致读[19].

云原生数据库也提出了实现多写的方案,进一步提出 Aurora Multi-Master[2]与 TaurusMM[16].Aurora Multi-Master 是在基础 Aurora 上进行水平扩展,与 Oracle RAC 和 DB2 Purescale 不同,并不是基于共享缓存与中心化事务管理器保证一致性,仍然以节点为粒度通过存储层分布式事务乐观并发控制实现.通过日志复制及回放在每个计算节点上保存数据多个副本.当出现冲突时,则拒绝更新,返回中止消息.当发起事务即产生日志记录的节点收到满足仲裁集合的成功回复,成功写入,否则回滚.所以,这种并发控制方式导致出现大量中止事务,并且还需要额外的仲裁以及回滚,影响性能.TaurusMM 延续使用中心化的全局锁管理器进行事务悲观并发控制,并引入矢量-标量混合时间戳标志事务偏序关系.矢量时间戳提供全局满足充分必要条件的排序,但锁机制保证单一页面上的写入一定是串行,只需要标量时间戳即可表示单一页面上更新的因果关系.所以只需要在进行全局对齐如获取全局快照时使用矢量时间戳,其他情况下只使用标量时间戳,尽可能控制网络消息大小，降低网络开销.

云原生多写数据库将一致性以及并发控制下压至日志层进行处理，通过控制日志记录的回放或回滚保证数据一致性.日志下沉避免数据页传输以及数据页写放大问题，写性能提高。但是读操作需要从存储层获取最新版本,读取性能受限.PG-RAC 基于共享缓存实现多写，虽需要解决缓存一致性问题，但是一定程度上平衡读写能力,提供较好的读性能.

1.4 分布式缓存一致性协议

目前缓存一致性协议主要有两种思路,基于嗅探的一致性协议以及基于目录的一致性协议.基于嗅探的一致性协议通过为每一个缓存单位记录共享状态,由网络上命令来更新此状态.存在两种更新方式,一种写入更新,将新的写入数据广播至所有缓存,更新旧版本,从而在下一次使用时仍缓存命中,本地仍是最新,如 Dragon 协议[20].另一种为写入失效,当发生新的写入时,广播一条失效消息,将所有其他缓存数据设为失效,下一次使用时缓存不命中需要主动获取最新数据,如多处理器缓存一致性的 MESI 协议[21,22,23,24].分布式缓存一致性协议 Hermes 协议[25,26]结合两种数据更新方式,在写入时同步广播失效消息,确保多节点数据强一致性；并同时在失效消息中携带更新值,更新旧数据后经确认再次提供读写.

基于目录的一致性协议构建目录在确定的位置对缓存单位的共享状态进行跟踪,通过访问目录获取状态,如多处理器中 DASH 协议[27],分布式系统中 Oracle RAC 的 Cache Fusion[4].RAC 为每个缓存数据在集群中选定一个主节点,在主节点的全局资源目录（GRD）中记录数据在所有节点上的使用情况.在进行并发读写时,需要访问主节点获取最新状态,到当前数据所在节点上获取最新数据后进行读写操作,最后更新主节点 GRD.

进一步扩展到节点副本一致性问题上,可以将缓存一致性问题视作需同步更新全局节点的副本一致性问题.在这一视角下,可通过广播、点对点复制以及链式复制的方式实现全局更新.广播协议通过主节点将最新数据广播复制到集群中所有节点,如 Hermes 协议.点对点复制由单个节点周期性随机选择若干邻居节点传播信息,通过层层传播,最终所有节点获得相同的信息,如 Gossip 协议[28].链式复制将集群节点组织为一条链,按顺序从头到尾依次进行复制,如 CRAQ 协议[29,30].

在 PG-RAC 的共享缓存中,使用基于嗅探的失效机制保证缓存一致性,又使用链式的缓存页面路由方式寻找主页.既避免目录式一致性协议的单节点瓶颈,又在吸纳链式组织节点,数据获取路径确定的优点的同时规避链式复制时延较大的缺陷.

2 PG-RAC 架构

2.1 整体架构

图 2 展示了 PG-RAC 的整体架构,依照功能可以将各组件分为事务组件、缓存组件、存储组件、网络组件四个部分.

事务组件. 事务组件分为全局事务管理器(GTM)和本地事务管理器(LTM).GTM 位于中心节点,维护全局事务信息,并负责统一分配事务号和快照.LTM 位于计算节点,负责同步必要的全局事务信息.LTM 开启事务前将必要的全局信息从 GTM 拉取到本地,在提交或回滚事务时将本地事务信息推送到 GTM.LTM 内部的日志管理器提供全局日志服务,其核心功能之一是确定全局日志顺序,后文将详细描述.

缓存组件. 缓存组件主要分为本地缓存服务（LCS）和全局缓存服务（GCS）.LCS 负责管理本地缓存.缓冲区内的页面分为主页和副本页,其中主页拥有页面的主权,读写操作均可在主页上进行.副本页是主页的拷贝,只提供读服务.LCS 通过内部的转移线程和失效线程,实现页面转移和缓存失效.GCS 位于中心节点,在 Oracle RAC 中 GCS 的主要作用是作为全局目录提供路由服务,PGRAC 的设计弱化了 GCS 的功能,实现了去中心化的路由机制,具体细节将在第四章展开.

存储组件. 在多台存储节点上搭建了一个分布式文件系统作为底层共享存储,数据文件和日志文件可被所有计算节点访问.文件有多份冗余，实现了存储上的容错。

网络组件. 网络组件实现了一个基于 RDMA 双边通讯的 rpc 通信框架,以降低节点间的通信延迟.

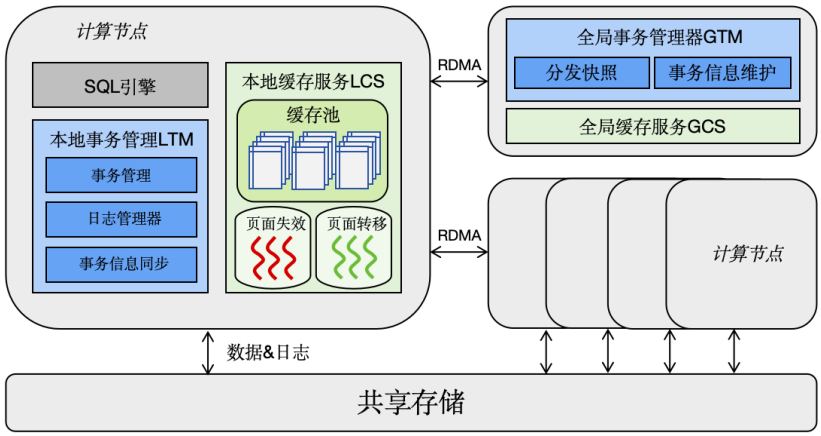

图 2 PG-RAC 整体架构(彩印,office ppt 绘制)

2.2 事务执行

PostgreSQL 通过多版本并发控制实现快照隔离,PG-RAC 遵循 PostgreSQL 的并发控制方法[31].本节介绍事务执行的具体细节.

事务信息. PG-RAC 维护了一些必要的事务信息,包括事务号、提交顺序号(CSN)、CSN 日志及事务快照.其中事务号用于唯一标识事务.提交顺序号 CSN 确定事务之间的提交顺序.CSN 日志记录事务号和提交顺序号之间的对应关系.快照表示事务时刻,用于可见性判断.上述事务信息由 GTM 统一分配或管理.虽然单点的事务管理器可能影响系统整体可扩展性,但根据我们的测试结果观察,在共享缓存架构节点数有限的前提下,中心化的事务管理器并不会成为全局瓶颈.

事务执行流程. 线程处理事务分为准备、执行、提交三个步骤.

(1)准备阶段.事务在开始后,从 GTM 获取快照,GTM 将两个全局变量(最老活跃事务号和最新已提交事务号,用于加速可见性判断)和当前 CSN 组合成快照,发送给执行事务的进程.收到响应后进程检验拉取线程是否已经拉取到必要的 CSN 日志,如果没有则进入等待.除此以外,进程还需要等待失效消息被应用,这将在第五章详细讨论.完成以上两个等待步骤后,进入指令执行阶段.

(2)执行阶段.指令执行过程中对页面的访问需要遵循一致性协议.读、写页面的规则不同.读操作可以在副本页或主页上进行.主页可以直接提供读,而副本页则需要判断是否失效.如果副本页已经失效,线程需要通过路由服务找到该页所处节点,从远端拉取页面并替换旧副本.写操作只能在主页上进行.如果当前不是主页,那么仍然从主节点获取最新页面,并修改路由信息.每个事务维护一个页面集合,记录事务内修改的所有页面,在事务提交时通知其他节点进行失效.如果页面不在共享缓冲区内,此时需要从底层存储拉取.

(3)提交阶段.执行完所有指令后,事务进入提交阶段.进程发起提交请求,GTM 根据提交信息更新全局事务信息,为事务分配 CSN 号.随后事务进程将页面集合通过 RDMA 广播给集群内的所有节点,完成这一步骤后提交完毕.

冲突处理. 传统数据库中存在多种冲突可能,主要包括页面级冲突、行级冲突、库表级冲突.本文充分考虑了可能的冲突情况.对于页面级冲突,PG-RAC 采用悲观的方式控制页面的并发访问.协议规定写操作只能在主页进行,而主页同一时刻只位于一个节点,这避免了不同节点对于页面的并发修改.根据快照隔离的要求,并发事务对同一行的修改是不被允许的.因此,针对行级冲突,PG-RAC 的实现方法是:在数据行上记录更新事务号,表示修改这一行的事务号.对于并发修改的情况,后修改这一行的事务会检查更新事务号对应的事务状态.如果更新事务号状态为"已提交",表明先进行更新的另一个事务已提交,需要终止并回滚本事务.如果更新事务号状态为"已终止",表明另一事务提交失败,可以继续修改这一行.如果更新事务号状态为"正在运行中",可以选择直接回滚,不影响正确性.或者等待直到远端事务结束,根据最终状态做判断.针对库表级冲突,PG-RAC 实现

了全局锁服务,并将 PostgreSQL 中原有的加库表级锁操作从本地修改为全局.库表级锁的使用频率较低,因此全局锁服务不会成为性能瓶颈.

死锁处理. 本文考虑了 PostgreSQL 中可能的死锁情况,并在 PG-RAC 中实现了相应的死锁处理方法.例如,事务 t1 和事务 t2 并发执行, t1 持有行 r1 的行锁,并申请 r2 的行锁. t2 持有行 r2 的行锁,并申请 r1 的行锁.这种情况将产生死锁.在本节的冲突处理部分已经提到,PG-RAC 是采用主动退出的方式处理行级冲突.在避免冲突的同时,也可以杜绝上述死锁情况的出现.

2.3 日志模块

CSN 日志. 提交顺序号 CSN 标识一个事务的全局提交顺序,而 CSN 日志负责记录事务号和 CSN 的对应关系.执行线程可以结合事务快照和 CSN 日志,判断数据的某一版本是否可见.本地新增的 CSN 日志在提交时更新到 GTM,LTM 无法感知其他计算节点的提交情况,因此还需要一个全局的同步管理器.我们避免了各节点 CSN 日志的强一致同步,而是在事务开始前拉取缺失的 CSN 日志信息.对每个节点记录一个偏移量表示目前已同步的位置,这确保了拉取过程是增量的.

CSN 拉取线程. 事务开始执行指令前,需要保证本地已经同步必要的 CSN 日志.然而拉取 CSN 日志是一个相对繁重的操作,将其放置在事务关键路径上会导致事务整体延迟增加,从而进一步加剧资源争用问题.PG-RAC 设计采用一个单独的 CSN 拉取线程持续拉取 CSN 日志,这使得获取 CSN 日志的操作从关键事务路径中分离出来.事务获取快照时,响应消息中会包含该事务需要推进到的CSN日志位置,如果拉取线程进度落后,则事务同步等待.这一设计还有几个优点：1.仅有一个线程来持续获取 CSN 日志,保证了顺序写入.2.单线程拉取操作的速度远大于 CSN 日志产生的速度,事务等待时间可以忽略不计.3.避免了多线程并发修改本地 CSN 日志,减少冲突.

去中心化 WAL 日志. 获取 WAL 顺序号是一个非常高频的操作,仅一条单行更新的 SQL 语句就可能包含若干条日志记录,因此由单个服务分配日志号的做法是不切实际的.PG-RAC 延续 Lamport 逻辑时钟[32]的思想保证全局的日志顺序一致.具体来说,日志号分为本地日志号和全局日志号,本地日志号指定本地日志在文件中的位置,每个节点顺序写入私有的日志文件.全局日志号确定不同节点间日志的顺序关系,每条日志记录中写有对应的全局日志号.全局日志号的分配有两个规则：（1）同一节点产生的全局日志号递增（2）页转移后分配的全局日志号大于转移前.这两点保证了同一个页面对应的全局日志号顺序一致.

2.4 故障恢复

去中心化日志使用日志号元组〈全局日志号,本地日志号〉标志日志顺序.本地日志号为实际本地日志写入位置,全局日志号标志全局事务顺序.当写入新的日志时,同时推进本地日志号与全局日志号,将全局日志号与本地日志号之间进行映射.当页面转移时,携带全局日志号信息进行同步.请求页面的节点将全局日志号与最新的日志对齐,并与此节点本地日志后进行映射.于是同步之后的新写入的日志项其全局日志项将必然大于发生在数据页之前的所有更新的日志号元组.从而可以保证,如果两个操作之间存在冲突,那么全局日志号更小的日志相对应的操作一定发生地更早.

故障恢复时,根据全局日志号对各节点的本地日志进行合并排序,然后从检查点开始恢复.恢复操作是以页面为单位的,保证了相同页面的日志顺序也就确保了最终恢复的正确性[33,34].

3 分布式路由策略

当页面的主节点发生变更时,需要以某种方式通知其余节点,以便后续可以从正确的位置获取页面.为实现这一目的,主要有基于广播和基于目录两种方式.

基于广播是指,在主节点变更时通过广播的方式告知所有计算节点,任何节点内部都记录了数据实际位置.基于目录是指,集群设置目录记录数据的位置信息.主节点变更时修改目录.需要远程访问时,访问目录以获取数据所属位置.

单机多核场景下,基于嗅探的缓存一致性协议通过总线实现广播,能提供更快的速度和更好的稳定性.而对于共享缓存数据库,通过网络实现广播的方式存在两个问题.首先,广播产生的数据传输量较大,随着集群的扩

展,网络带宽将有可能成为瓶颈.其次,一次主节点变更需要收到所有节点的回复,集群中任何节点出现网络波动都会阻塞变更的提交,进而影响整体性能.而基于目录的方式受到网络影响更小,具有更好的可扩展性.

出于以上原因,本文没有采用广播的方式维护缓存一致性，而是考虑基于目录的缓存一致性管理.本文实现了中心化缓存管理,并在此基础上进行了一定探索,创新性地提出了链式路由策略.

3.1 中心化缓存管理策略

在开始一次远程访问前,计算节点通过路由获取页面的具体位置.现有共享缓存数据库对路由机制的设计基本上参考了基于目录的缓存一致性协议,由目录管理缓存一致性相关的信息.通过访问该目录,CPU 得到缓存行的位置信息,从而获取最新数据,或令其余核心的缓存行失效.在分布式共享缓存数据库的典型设计中,可以实现类似的目录缓存管理服务,目录服务的主要功能是提供路由服务,并控制页面的并发访问.我们在 PG-RAC 上实现了这种基础的路由策略,并称之为中心化缓存管理.

根据中心化缓存管理,一次常规的远程访问包括三个步骤.

(1) 对 GCS 的页面项加共享锁或排他锁,成功加锁后,将主页位置作为响应返回给计算节点.

(2) 计算节点向主节点发送请求,主节点回复最新的页面内容.

(3) 计算节点收到页面,再次向 GCS 发送解锁请求.

显然,中心化缓存管理的设计存在两个限制.首先,中心化的缓存一致性管理,极大限制了系统的可扩展性.其次,一次远程访问包含了固定的三次往返通信,开销过大.为了缓解中心化瓶颈,部分系统对 GCS 进行分片处理,每个分片管理全局缓存的一个子集,这一定程度上缓解了第一个问题.然而通信开销过高的问题仍然无法得到解决.为此,PG-RAC 提出了一种新型的分布式路由机制,同时解决了中心化缓存管理的两个限制.

3.2 链式路由策略

链式访问规则. 如图 3 所示,在链式路由中,每个计算节点内存中都有一份完整的路由表,路由表存储页面与所属主节点的映射关系.路由表中的主节点起到指针的作用,指向下一个写入该页面的节点.路由表仅在主页转移时被修改.一次主页转移由源节点发起,目标节点（持有主页的节点）收到主页转移请求后,修改本地路由表,将页面对应的主节点改成源节点,随后进行页面内容的转移.源节点收到返回的页面后,修改本地路由表,标记该页面为主页.

计算节点访问页面前,首先查询本地路由表.如果对应的主节点为本节点,表明当前页面为主页,可直接进行读写.反之则需要一次远程访问,源节点向路由表指向的主节点发送页转移请求.初始状态下,本地路由表查不到页面的路由信息,则向 GCS 发送请求(全局对页面的第一次访问会被注册到 GCS).如果 GCS 也没有路由信息,表明主页不在任何一个节点上,需从共享存储读取.

依据上述的修改规则,本地的路由信息可能是过时的,实际的主页可能已经被转移到其他节点,此时目标节点将请求转发给下一个节点,直到找到主页实际位置.以上的整个访问过程可以被总结为“链式访问”.如图 3 所示,页面 P0 所属主节点依次为节点 1,2,3,因此形成了图 3 中的路由表.节点 1 的路由表已经过时,访问节点 2 后得知页面已被转移到节点 3,转发到节点 3 后最终由节点 3 返回最新页面.

避免饥饿问题. 假设一个节点进行链式访问,每次写请求到达主节点前,都有另一个节点的写请求先到达.这意味着每次处理请求时主页已经被转移了,该节点的事务始终无法继续推进.这种饥饿问题在写密集场景下是有可能出现的.PG-RAC 通过良好的设计杜绝了这一问题：到达主节点的远程访问请求会形成一个等待列表.主节点按顺序处理等待列表中的请求.主页转移时,将主页和等待列表一起发到远端节点,后续由新主节点接管等待列表.主页转移必定伴随一次对页面的写入操作,因此主页转移的进度相比链式路由更慢.这就意味着任何时刻发起的远程访问请求最终会落入等待列表,被某个计算节点处理.

并发处理. PG-RAC 实现了路由项锁,以控制对路由表的并发访问.每个页面对应一个路由项锁.路由项锁有共享和排他两种模式.远程读请求仅查询路由表,因此加共享锁.远程写请求则需要对路由项加排他锁.为了避免上文提及的饥饿问题,对路由项锁实现了等待队列,主页转移时将页面和队列一起发送.路由项锁仅短暂持

有,持锁期间不会占有其他资源,不具备死锁产生条件,因此无需进行死锁检测.

在确定页面所属位置后,访问请求被转发到对应节点,该节点会将页面内容和等待队列打包成消息发送给请求节点.整个过程间需要对页面加共享锁,避免其他进程并发修改.

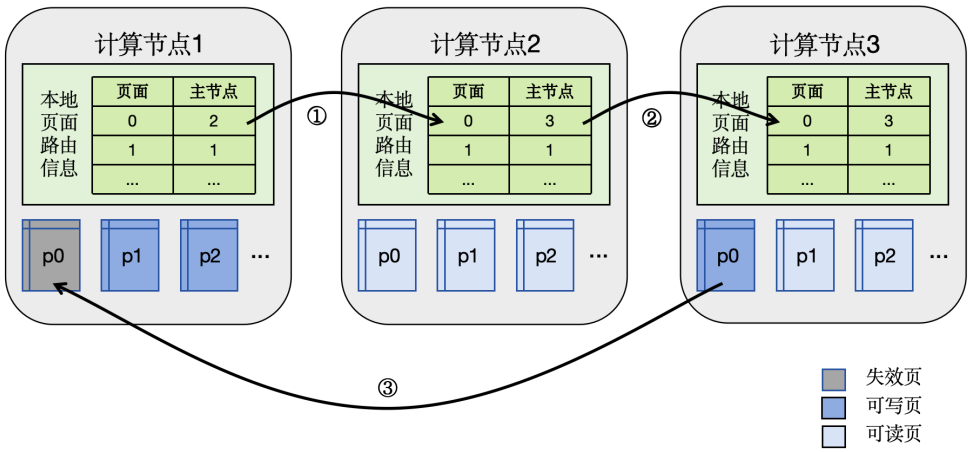

图 3　链式路由机制(彩印,office ppt 绘制)

3.3 路由路径优化

在集群计算节点数量较少的情况下,链式路由机制具有优越的性能.在仅有两个计算节点的特殊情况下,每次远程访问都只涉及一次往返通信.四节点集群运行 sysbench 读写混合负载,可以保证 1 到 2 跳内的远程访问占比达到 85%以上.然而,集群继续横向扩展会使情况发生改变.多个节点对相同页面的连续修改会增加访问链的长度,可能使远程访问的平均开销超过中心化路由机制下的三次.虽然在共享缓存数据库集群配置过多计算节点是不被建议的,这个问题仍然不能被忽视.因此,本文实现了缩短访问链路径的几个优化,以减少远程访问的往返次数.

a.源节点发起写转移请求后,链式访问过程中可能会途径非主节点.跳转过程中可以将中间节点的路由表值修改为源节点.这样,下次由中间节点发起的写请求就不需要后续多余的跳转.优化的依据是：源节点发起了写请求,最终必然会成为主节点.在源节点拿到主页之前,中间节点的远程访问会被阻塞.

b.提出旧页面主动更新机制.在本地缓存管理的失效线程内维护一个队列,标记页面为失效后将页面号加入队列.后台的更新线程周期地从队列取出旧页面进行更新,这既保证了读副本的新鲜度,也可以将本地路由表更新至最新状态.更新线程不在事务关键路径中,因此不会增加事务延迟.

c.对于某一个页面,每经过 x 次主页权的转移,就以同步的方式更新它在 GCS 的主页信息.当一次链式访问跨越的节点数超过一个限定 y 时,通过访问 GCS 可以实现快速跳转,以减少访问链的长度.x 的值应当控制在一个合理范围,过小的 x 值会导致页转移的延迟增加,过大的 x 值意味着 GCS 信息过于滞后.同时,为了防止反向跳转,还要确保 x<y.这一机制主要针对跨节点数过大的极端情况.

算法 1 给出了应用优化后,一次链式路由的算法流程.其中 *next_owner* 表示链式路由过程中下一个应该访问的节点 id,*local_id* 为自身节点 *id,num_traverse* 表示当前访问跳转次数.

在给出算法前,首先对伪代码中使用的函数作如下解释.函数 **Hash** 的功能是输入页面描述符，查询本地的哈希表，返回值是页面物理位置 *addr.*函数 **RemoteAccess** 的功能是输入页面描述符 *buf_desc* 和目标节点 id，发起远程访问,返回值 *response* 为目标节点的回复消息.函数 **AccessGCS** 的功能是输入页面描述符 *buf_desc,*访问 GCS 并返回页面所在位置.

发起一次链式路由首先读取本地路由表（第 2 行），获取页面路由信息。如果判断页面位于本地缓冲区，则通过哈希函数获取页面实际位置(3-5 行)，并返回结果。如果判断页面位于远程缓冲区，则发起一次远程访问(7-15 行）。根据优化 b，链式访问过程中如果超过规定跳转次数，将访问 GCS 进行快速跳转(12-13 行)。通过链式访问获取到数据页后，需修改本地路由表(16-23 行)。最后，根据优化 c，更新多次转移的主页 GCS

信息(20-23 行)。

算法 1: 链式路由算法

输入: 页面描述符 *buf_desc*,访问模式 *mode*,页面每转移 *x* 次更新一次 gcs,单次访问最大跳转次数 *y*

输出: 页面地址 *addr*

```
1:  begin
2:  next_owner ← buf_desc.owner_id    // 读取本地路由表，获取路由信息
3:  if  next_owner == local_id  then  // 页面位于本地缓冲区
4:      addr ← Hash(buf_desc)
5:      return addr
6:  else                              // 页面位于远程缓冲区,发起链式访问
7:      num_traverse ← 0
8:      response ← RemoteAccess(buf_desc,next_owner)  // 发起远程访问请求
9:      while next_owner ≠ response.next_owner        // 链式跳转,直到找到准确位置
10:         if  ++num_traverse <= y  then
11:             next_owner ← response.next_owner
12:         else
13:             next_owner ← AccessGCS(buf_desc)      // 优化 b:通过 gcs 进行快速跳转
14:         response ← RemoteAccess(buf_desc,next_owner)
15: copy response.page → BufferPool  // response 内部的页面内容拷贝到本地缓冲区
16: if mode == Read then
17:     buf_desc.owner ← next_owner
18: else if mode == Write
19:     buf_desc.owner ← local_id
20:     buf_desc.num_traverse++
21:     if  buf_desc.num_traverse == x  then
22:         UpdateGCS(buf_desc)           // 优化 c:更新 GCS 路由表
23:         buf_desc.num_traverse ← 0
24: return addr
25: end
```

通讯次数分析. 采用分布式链式路由策略显著降低了通讯开销.根据我们的测试结果,应用优化的链式路由策略后,配备 6 台计算节点的 PG-RAC 集群运行 sysbench 读写混合负载,可以保证 1 到 2 次往返通讯的远程访问占比达 92%以上,平均通讯往返次数为 1.45,显著低于中心化路由策略的 3 次通讯往返.同时,路由路径优化有效降低了远程访问的尾延迟,3 到 5 次往返通讯的访问占比仅 7%,不存在通讯次数大于 5 的远程访问.

3.4 路由故障恢复

在去中心化页面路由机制中,即使只有单点故障,集群中其他正常运行节点因为故障节点在页面路由的关键路径上也不可指示页面位置,只有等到故障节点完全恢复后才可重新提供服务.在现有的方法下,恢复故障节点通过将所有计算节点日志合并后按照全局日志号顺序回放,从而恢复出所有页面的最新版本以及页面转移

信息.然而实际上,故障节点只需要恢复当前持有主页的具体数据,和其他非主页的路由信息即可,直接全局回放日志将导致多余的日志扫描与重做.

于是提出了页面转移日志,记录页面的转移情况.在进行页转移时,目标节点写入页面转移日志.通过回放页面转移日志,即可确定故障节点上主页,即必须进行日志重做的缓存页,以及非主页的页面转移路由信息.于是，后续只需等待回放故障节点主页的日志记录，非主页上访问可直接转发至其当前主页所在节点.从而缩短故障节点日志重做过程,并缩短系统不可用时间.

页面转移日志记录转移源节点与目标节点,同样适用去中心化的全局日志号与本地日志号的二元组标志全局顺序.当发生页面转移时,目标节点在收到缓存页后写入更新前,写入转移日志.页面转移日志在事务提交时进行持久化.所以,保证页面的转入日志一定在页面的修改之前持久化.如果转入日志仍未持久化,说明页面修改的事务也仍未提交,所以认为主页仍在之前的节点上,在之前节点上回放也是最新版本,并不影响正确性.

4 副本页失效机制

多核体系结构的缓存一致性协议规定,在数据修改操作完成提交之前,其他 CPU 上的缓存必须被同步地标记为失效,以此保证数据的一致性.在单机多核 CPU 场景下,同步失效的代价是可以接受的,因为连接 CPU 的通信总线能提供极低的延迟.然而,对于由网络连接的共享缓存数据库,实行同步失效会严重影响系统性能.针对这一问题,PG-RAC 结合多版本并发控制的特性,对副本页失效机制进行了优化.

在第二章中已经提到,快照隔离级别下的事务在开始前会获取一个快照.根据这个快照,一个事务的读操作可能作用于数据的旧版本上.从页面层面来看,即使缓存页不是最新的,其中也可能包含了数据行的旧版本.对于一些较早开启的事务,这部分旧的数据行是可见的,无需更新缓存页.基于以上事实,我们可以进一步改进副本页失效机制.首先，本文提出了事务路径外失效,将失效操作从事务路径中分离出来,有效减小事务持有资源的时间。另外，本文还提出结合多版本并发控制方法的特性，推迟页面失效时机.这一机制使得一些过时的页面仍能进行读操作,降低了副本同步的代价.

4.1 事务路径外失效

处理缓存一致性问题的最朴素的想法是,当主页发生修改时就立即通知读副本进行失效.这种做法没有利用多版本的特性,会造成过大的同步开销.根据事务隔离性的要求,未提交事务的修改对于其他并发事务而言,本身就是不可见的内容.因此可以在事务提交的时刻统一进行失效.对于每个单独的事务,PG-RAC 将事务过程中修改的页面整合为一批,作为消息发送.消息附带事务的提交顺序号,这样远端节点可以确定失效消息所对应的事务.批量失效有助于减小副本同步带来的额外延迟,在缩短事务执行时间的同时缓解了并发事务之间的资源争用,可有效提升系统整体性能.

事务在运行过程中通常会持有特定资源,例如行锁,这会影响并发事务的运行.因此,减少事务的持续时间能有效减少并发冲突的可能.上述的批量失效针对这一点做了很好的优化,但仍然没有完全消除同步等待的延迟.如果等待其余节点的回复时间较长,将严重阻塞该事务的提交.因此,本文提出异步失效,如图 4 所示,发起失效的节点无需同步地等待远端节点响应,只要确保失效消息发出,可以立即提交事务.同时,每个计算节点开启一个回放线程,异步回放收到的失效消息.在每个事务开始前,需要保证已提交事务的失效消息应用到本地失效哈希表.由回放线程维护一个变量 current_csn,这个变量表示失效消息在本地的推进进度.事务开始前比较快照提交时间戳和当前提交时间戳,决定事务能否开启.

通过事务提交时批量发送失效,节点失效线程异步回放的方式,PG-RAC 完全将失效操作从事务路径中分离.事务结束前的等待时间被转移到事务开始前,但是事务开始前线程没有读写任何数据,并不会持有资源,因此不会阻塞并发事务的运行.事务路径外失效机制有效减小事务持有资源的时间,减少资源争用,提升了系统可扩展性.

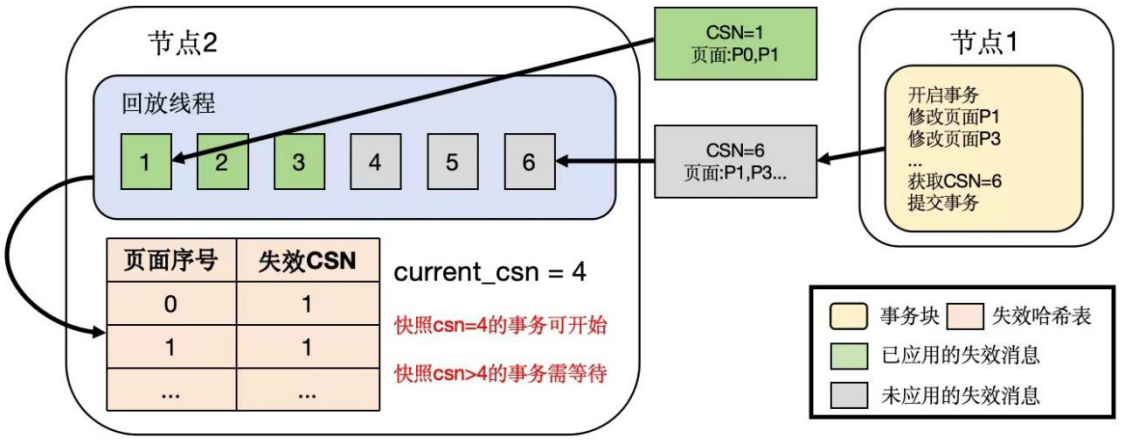

图 4 事务路径外失效 - 示意图(彩印,office ppt 绘制)

4.2 推迟页面失效时机

快照隔离保证事务中进行的所有读操作都将看到数据库的一致快照.因此,对于满足快照隔离的事务,其可见的数据范围在事务开始时就已经被确定.事务运行过程中即使有并发事务的修改提交,对于该事务也是不可见的.根据 4.1,失效消息的发送时机已经被推迟到事务提交时刻.但是,对于更早开启的事务,一部分已被标记为失效的页仍能提供读服务,我们通过下面的例子阐释.

以图 5 为例,副本所在节点(后续称为“副节点”)和主节点同时开启事务.主节点修改了主键为 1 的元组并提交,提交前获得事务的 CSN 为 5,然后发送失效消息给共享节点.副节点获取快照 CSN 为 5,然后查询主键为 1 的行.根据 PostgreSQL 的并发控制原理,由于新增版本的 CSN 大于等于快照 CSN,这一版本对于副节点事务是不可见的.实际可见的版本(属性列=a 的行)在旧页和新页中都存在.即使旧页在缓存粒度上是过时的,它仍然能满足事务层面的一致性,因此暂时不需要将其失效.

具体实现上,PG-RAC 在各计算节点维护一个哈希表记录缓存状态.可以通过哈希表查询某页面是否是主页.哈希表还额外记录了每个页面的失效信息,用一个 64 位整形表示,称为失效 CSN.失效线程按事务提交顺序,依次处理失效消息.哈希表某条目为空时,失效消息中的 CSN 会被赋值给哈希表中的失效 CSN,而后续的失效消息不会覆盖先前的写入.

事务读取缓冲区内的页面前,首先查询失效哈希表.如果事务的快照 CSN 小于等于页面的失效 CSN,说明对于本事务而言,其余节点产生的修改是不可见的,可以继续读本地的旧页.反之说明本地旧页对该事务已经失效,需要拉取最新的页面. 每次更新页面时,都会将哈希表中的失效 CSN 重置为空. 失效 CSN 为空表示此时副本页和主页是同步的.

根据以上规则,失效 CSN 实际上可以确定副本页在整个时间线上处于哪一段.通过比较快照 CSN 和失效 CSN 的大小，就可以确定页面在事务层面上是否有效.

5 实验分析

页面推迟失效机制利用了多版本并发控制的特性,将系统对一致性的要求从缓存层面放宽到事务层面,极大地降低了同步副本的开销.应用这一机制的系统仍能满足事务一致性,并不违反快照隔离的定义.

这一章节展示 PG-RAC 的实验结果.我们采用两个基准测试(Sysbench,TPC-C)评估 PG-RAC 的整体性能和可扩展性.对比了 PG -RAC 与(1)单机 PostgreSQL(2)基于 PostgreSQL 的无共享架构数据库 citus.设置实验测试了本文两个创新性改进对性能的影响.

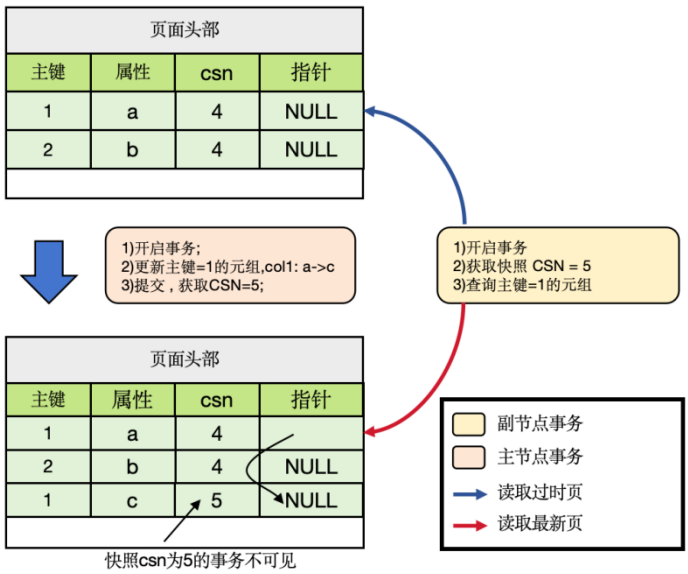

图 5　副本失效推迟 - 示意图(彩印,office ppt 绘制)

5.1 实验设置

实验运行在一个由计算节点和存储节点构成的集群上,每个计算节点配备有 Intel Xeon Gold 6240M 2.60GHz CPU(72 个虚拟核)和 376GB 的内存,并运行在 Centos7 操作系统之上.存储节点拥有和计算节点相同的配置,可根据实际需求对共享存储层进行扩缩容.集群由 15Gbps 的网络连接,每台机器配备有 Infiniband 网卡.将每个计算节点的缓冲区大小设置为 32GB,同时限制每个计算节点只能使用 32 个虚拟核.生成负载的进程运行在独立的服务器上,避免占用计算节点的硬件资源.以下所有测试的数据库系统都采用可重复读隔离级别,这一隔离级别能防止绝大多数异常并且提高事务执行的并发度,在工业级场景被广泛应用.

我们使用两个 OLTP 基准测试(Sysbench 和 TPC-C)对 PGRAC 进行了评估.为了测量数据访问的冲突率对系统性能的影响,对 sysbench 作了一定调整,将数据按照主键值均匀地进行划分,总共分成节点数份,每个计算节点分配有专属的热点区域,剩余数据属于非热点区域.各计算节点有 X%的概率访问非热点区域(即所有节点共享的区域),1-X%的概率访问本地专属的热点区域,下文统一称这个 X 值为共享率.共享率可以直观体现数据访问的冲突概率,共享率越高,访问的冲突概率越大.Sysbench 的实验设置共有 100 个表,每个表 50 万行数据(共 10GB 数据).另外,我们使用 BenchmarkSQL 作为测试工具,测试 PGRAC 在工业级基准测试 TPC-C 上的表现.实验设置了 100 个仓库,将数据根据仓库序号进行分区.同时,负载进程根据事务第一条 sql 语句的谓词,将事务请求发送到对应的计算节点.在 PG-RAC 的共享缓存架构下,数据会随着请求转移到相应节点,因此 TPC-C 负载在运行稳定后,自然形成了数据的分区.

5.2 多写性能

图 6(a)和图 6(b)分别展示了 PG-RAC 在 sysbench 读写混合负载、sysbench 只写负载上的性能表现.其中 K-QPS 指"千查询每秒",表示系统每秒能够处理的查询数量,1K-QPS 表示系统每秒可处理 1000 个查询.将计算节点数从 1 逐渐扩展到 6,每个节点启动 32 个连接,以测试系统的整体可扩展性.

从图中可以看出,对于两种负载,六节点以下的集群都可以做到接近线性扩展.可以注意到,随着共享率的增加,扩展性不可避免地会有所下降.然而即使数据共享率达 40%,增加节点数仍然能提升整体吞吐.对于读写混合和只写负载,6 节点集群运行并控制共享率在 40%,吞吐率仍能达到单节点集群的 2.80 倍和 2.28 倍.图 6 中的右半部分给出了两种负载下,单个事务的平均延迟.可以看到(1)当数据访问完全没有冲突时,随着节点数的增加,平均延迟的增长非常细微.(2)共享率增高时,增加节点数会导致延迟增加,因为节点数的增加导致了更多的冲突访问(3)节点数相同时,增加访问冲突率,也会显著增加延迟.由以上三点可以知道,仅增加节点数对性能的

影响极小,并发访问数据才是影响共享缓存数据库性能的关键因素.由于我们实现了快照获取上的优化,只读事务的效率得到了极大提升.sysbench 只读负载下可以做到完全线性扩展,在本文中我们不做展示.

资源消耗评估. 成本是评估云原生数据库的重要指标之一,因此本文设计了一组实验以测试 PG-RAC 在资源利用率方面的表现.实验选用 sysbench 基准测试,设置 100 个表,每个表含 160 万行数据,共计约 32GB 数据.. 表 2 给出了 PG-RAC 在 1、2、4 节点集群下的若干资源相关指标,包括平均 CPU 利用率、共享存储每秒 IO 次数(IOPS)及内存使用情况.实验结果显示,随着集群的横向扩展,CPU 资源的损耗并不显著.在平均 CPU 利用率上,4 节点集群相比单节点集群仅降低 6.29%.部分云原生数据库对于存储节点的计费主要依据网络 IO 次数,因此对于同等性能水平的云数据库,所耗费的 IO 次数应尽可能少.随着集群从 1 扩展到 2、4 节点,IOPS 分别提升 1.96 倍和 2.59 倍.这个提升幅度与吞吐提升幅度基本成正比关系,不存在显著的额外 IO 开销.在内存使用方面,PG-RAC 展现出多写内存扩展的优势: 4 个节点各自使用 8GB 的本地缓冲区,组成 32GB 的全局缓冲区.扩展后的缓冲区可以容纳完整的数据集,这极大地减小了脏页落盘对性能的影响.此外,4 节点集群可以使用多条小容量内存,相比于单条大容量内存(单节点 32GB),成本更低且灵活性更好.

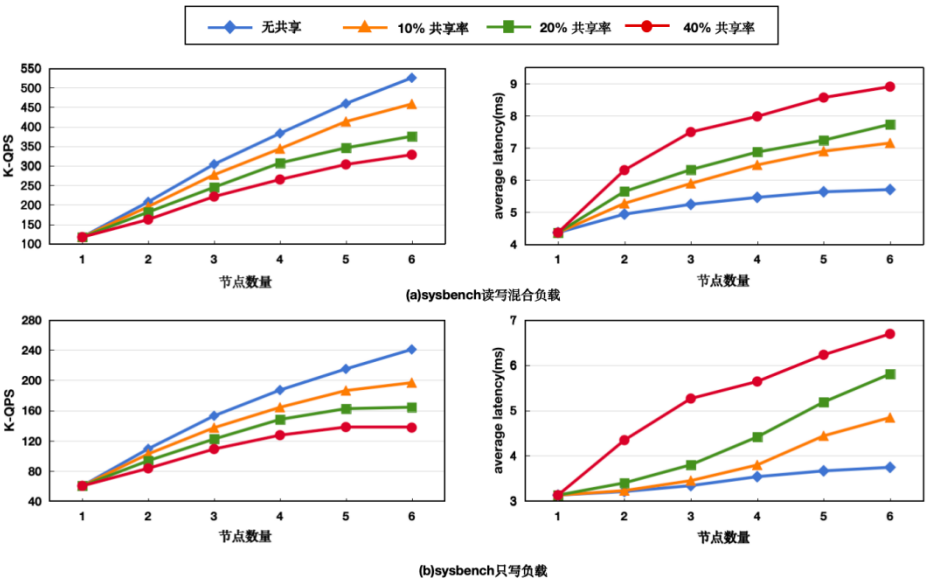

图 6 PG-RAC 总体性能测试(彩印,office excel 绘制)

	1 节点	2 节点	4 节点
平均 CPU 利用率(%)	41.50	39.63	35.21
每秒 IO 次数(IOPS)	17459.23	34274.67	45293.92
缓冲区大小	1*32GB	2*16GB	4*8GB

表 2 PG-RAC 资源消耗评估

5.3 链式路由影响

设置对照试验,以评估链式路由机制的影响.共享率被设定为 60%,运行 sysbench read-write 负载.图 7 展示了不同节点数集群下,链式路由机制远程访问所需的跳转次数.对于配置了 2 个计算节点的 PG-RAC 集群,主页

只可能处于本地或另一个远端节点,因此每次远程访问都仅需要 1 次往返通信(Roundtrip,RT).而如同预期的那样,随着节点数增加,RT 较高的访问占比也逐步增加.4 节点集群下,2RT 和 3RT 的远程访问占据了 29%.6 节点集群下开始出现极少量的 4RT 和 5RT 的访问,并且 1RT 比例进一步下降.由此可以得知,链式路由机制在集群规模较小的场景表现最佳.对于 4、6 节点集群,1RT 的访问仍占比超过 50%,平均 RT 分别为 1.31 和 1.45,仍低于中心化路由固定的 3RT.

表 3 展示了两种路由机制在 6 节点场景下,运行 sysbench 负载的性能表现.链式路由机制的吞吐显著高于中心化路由,并且事务平均延迟和 95%尾延迟都要更低.这表明即使运行在节点数较高的集群,链式路由仍然具有一定优势.我们发现节点数大于 6 时,频繁的冲突访问成为了限制性能的主要瓶颈,而对中心化服务(GTM 和 GCS)的访问延迟在访问链路中占比较小,此时选择何种路由机制对整体性能影响不大.

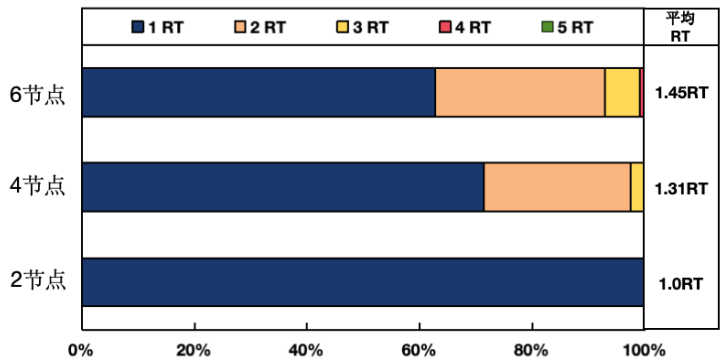

图 7 链式路由机制跳转次数(彩印,office excel 绘制)

路由策略	吞吐率(K-QPS)	平均延迟(ms)	95%尾延迟(ms)
链式路由	261.158	11.90	42.15
中心化路由	227.428	14.39	46.06

表 3 链式路由与中心化路由性能对比（6 节点）

5.4 副本页失效机制的影响

本文设计两组实验,以验证缓存处理策略优化的影响.为了更直观体现冲突率对于失效机制的影响,相比 5.2 的测试减小了数据总量,运行 sysbench 读写混合负载,设置共 20 个表,每个表 50 万行数据.两组实验设定共享率分别为 20%和 60%

每组实验测试不同设计组合的性能指标.路由策略包括分布式路由和中心化路由；副本失效机制可选择本文中的副本页失效方法,或事务内部失效的方法.两两组合,可以得到四种不同的缓存层设计决策.图 8 的左图和右图分别展示共享率为 20%和 60%时,四种设计的吞吐率随节点数的变化.实验结果显示,如果在事务内进行失效,此时同步副本的操作就成为系统的最主要瓶颈.此时无论选择链式路由或中心化路由,QPS 都处于较低水平,可扩展性也严重受限.引入优化后的页面失效机制,系统性能得到大幅提升.

共享率为 20%时,QPS 随着集群规模的扩展而持续上升,4 节点下相同路由策略的设计分别提升 1.94 和 1.97 倍,6 节点下相同路由策略的设计分别提升 2.18 和 2.26 倍.

共享率为 60%时,4 节点下相同路由策略的设计分别提升 1.80 和 1.85 倍,6 节点下相同路由策略的设计分别提升 2.06 和 2.09 倍.随着节点数扩展到 6 个,受冲突访问影响,四种设计的 QPS 普遍有所下降,但应用了优化缓存策略的设计所受影响更小.

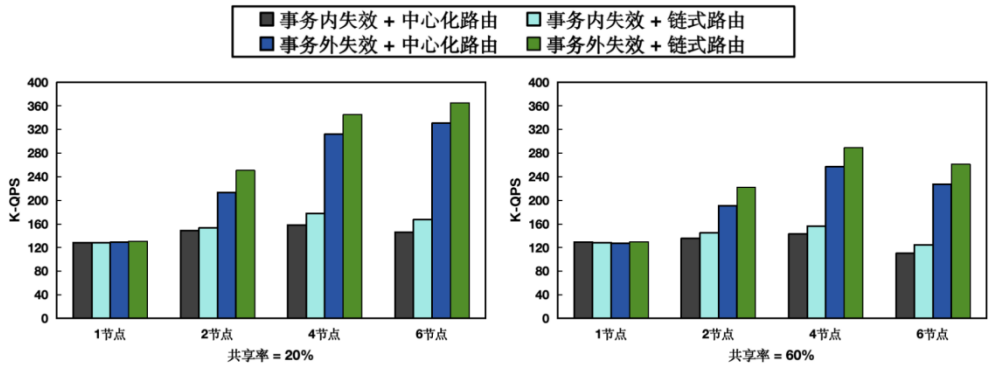

图 8 缓存协议不同组合的性能对比(彩印,office excel 绘制)

5.5 与其他系统对比

图 9 展示了在 sysbench 和 TPC-C 两个负载下,PG-RAC 与其他数据库的性能对比.对比的数据库选取了 PostgreSQL 以及开源分布式数据库 citus.PostgreSQL Server 有较长的历史,被广泛认为是先进并可靠的.citus 是无共享架构的分布式数据库,基于 PostgreSQL 开发.对于 sysbench,我们取消了前文提到热点访问,而是以均匀概率访问完整数据集.为了更精确的反映性能差异,通过实验选出了三者达到最高吞吐的连接数.PG-RAC 每个计算节点开启 64 个连接,citus 则是在一节点和四节点场景下,分别采用 128 连接和 256 连接,单机 PostgreSQL 采用 64 连接.实验结果显示,采用无共享架构的 citus,在 sysbench 这类不具备访问局部性的负载下表现不佳,4 个执行者+1 个协调者的 citus 集群的性能表现甚至弱于单机 PostgreSQL.而 PG-RAC 在这种场景下有更好的可扩展性,4 计算节点集群的吞吐达到单节点的 1.5 倍,达到单机 PostgreSQL Server 的 1.3 倍.对于 TPC-C,考虑通过数据分区以减少分布式数据库的额外开销.citus 采用无共享架构的通用做法,在运行测试前对数据分区,降低分布式事务的开销.对于 PG-RAC,执行负载进程根据 sql 的谓词(仓库序号 warehouse id),将事务请求发送到对应的计算节点,减少数据移动的开销.图 9 右图的纵坐标单位 K-tpmC 表示每分钟完成的 New Order 事务数量(千个),可以直观展示系统的吞吐性能.实验结果显示,对于 TPC-C 这类有较好分区性质的基准测试,两种架构的分布式数据库都展现出良好的可扩展性.四计算节点集群的 PG-RAC 性能表现更优.吞吐量达到 citus 的 1.5 倍,单机 pg 的 1.9 倍.

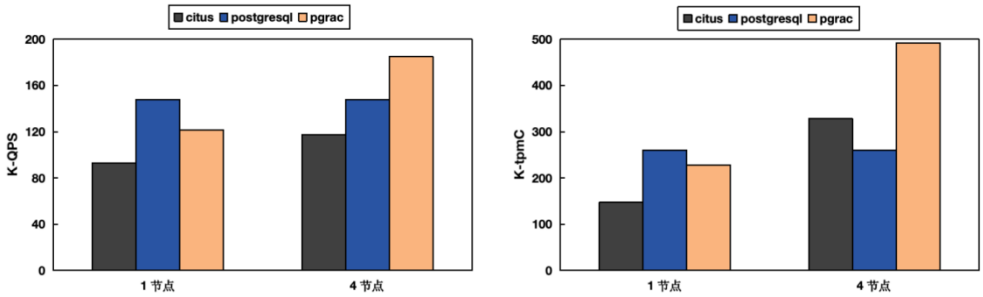

图 9 PG-RAC 与其他数据库对比测试(彩印,office excel 绘制)

6 总结

基于 PostgreSQL,本文设计并实现了一个高性能的共享缓存数据库系统 PG-RAC.PG-RAC 的整体架构由缓存层、事务层、存储层、网络层四个模块构成.本文提出了对系统四个模块的具体实现和优化措施.特别的,PG-RAC 对数据系统的缓冲区模块和事务并发控制模块提供了多种优化技术.共享缓存架构是云原生数据

库的一个发展方向,本文对共享缓存架构的缓存一致性协议方面做了一系列探索.PG-RAC 对缓存一致性协议提出了两个创新性改进（1）设计并实现分布式链式路由策略,代替基本的中心化目录式路由,中心目录的访问压力被分散到各计算节点.（2）优化了副本失效机制,将失效操作从事务路径中分离,在此基础上,本文还设计了与事务多版本特性结合的缓存一致性协议,推迟副本页失效时机,极大地提升了系统性能和扩展性.

References:

[1] A. Verbitski et al. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In Proceedings of the 2017 ACM International Conference on Management of Data (Chicago, Illinois, USA) (SIGMOD '17). ACM, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918. 3056101

[2] E. Boutin and S. Abraham. 2019. Amazon Aurora Multi-Master: Scaling out database write performance. Amazon ReInvent.

[3] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In SIGMOD. 215–226.

[4] Armenatzoglou N, Basu S, Bhanoori N, et al. Amazon Redshift re-invented[C]//Proceedings of the 2022 International Conference on Management of Data. 2022: 2205-2217.

[5] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In SIGMOD. 789–796.

[6] Zhang Z, Hu H, Zhou X, et al. Starry: multi-master transaction processing on semi-leader architecture[J]. Proceedings of the VLDB Endowment, 2022, 16(1): 77-89.

[7] Lahiri T, Srihari V, Chan W, et al. Cache fusion: Extending shared-disk clusters with shared caches[C]//VLDB. 2001, 1: 683-686.

[8] S. Chandrasekaran and R. Bamford. 2003. Shared cache-the future of parallel databases. In Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405). IEEE Computer Society, NY USA, 840–840.

[9] IBM white paper. 2009. Transparent application scaling with IBM DB2 pureScale.

[10] Yan L. Analysis and Construction of the Database Architecture based on Oracle RAC[J]. Computer Systems & Applications, 2013 (11): 200-203 (in Chinese with English abstract).

[11] Depoutovitch A, Chen C, Chen J, et al. Taurus database: How to be fast, available, and frugal in the cloud[C]//Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020: 1463-1478.

[12] Antonopoulos P, Budovski A, Diaconu C, et al. Socrates: The new sql server in the cloud[C]//Proceedings of the 2019 International Conference on Management of Data. 2019: 1743-1756.

[13] Cao W, Li F, Huang G, et al. Polardb-x: An elastic distributed relational database for cloud-native applications[C]//2022 IEEE 38th International Conference on Data Engineering (ICDE). IEEE, 2022: 2859-2872.

[14] Cao W, Liu Z, Wang P, et al. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database[J]. Proceedings of the VLDB Endowment, 2018, 11(12): 1849-1862.

[15] Verbitski A, Gupta A, Saha D, et al. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes[C]//Proceedings of the 2018 International Conference on Management of Data. 2018: 789-796.

[16] Depoutovitch A, Chen C, Larson P A, et al. Taurus MM: Bringing Multi-Master to the Cloud[J]. Proceedings of the VLDB Endowment, 2023, 16(12): 3488-3500.

[17] Dong HW, Zhang C, Li GL, Feng JH. Survey on Cloud-native Databases. Ruan Jian Xue Bao/Journal of Software, , 2024, 35(2): 899-926. (in Chinese with English abstract) http://www.jos.org.cn/1000-9825/6952.htm

[18] Cao W, Zhang Y, Yang X, et al. Polardb serverless: A cloud native database for disaggregated data centers[C]//Proceedings of the 2021 International Conference on Management of Data. 2021: 2477-2489.

[19] Yang X, Zhang Y, Chen H, et al. PolarDB-SCC: A Cloud-Native Database Ensuring Low Latency for Strongly Consistent Reads[J]. Proceedings of the VLDB Endowment, 2023, 16(12): 3754-3767.

[20] Atkinson R R, McCreight E M. The dragon processor[J]. ACM SIGOPS Operating Systems Review, 1987, 21(4): 65-69.

[21] Papamarcos M S, Patel J H. A low-overhead coherence solution for multiprocessors with private cache memories[C]//Proceedings of the 11th annual international symposium on Computer architecture. 1984: 348-354.

[22] Shukur H, Zeebaree S, Zebari R, et al. Cache coherence protocols in distributed systems[J]. Journal of Applied Science and Technology Trends, 2020, 1(2): 92-97.

[23] Chen JC, Li YH, Zhao YQ, Wang ED, Shi HZ, Tang SB. A Shared-Forwarding State Based Multiple-Tier Cache Coherency Protocol[J]. Journal of Computer Research and Development, 2017, 54(4): 764-774.(in Chinese with English abstract)

[24] Liu DF,Chen TS,Guo Q.DLS:a directoryless coherence protocol for shared cache[J].Gao Ji Shu Tong Xun/High Technology letters,2015,25(05):445-452.(in Chinese with English abstract)

[25] Katsarakis A, Gavrielatos V, Katebzadeh M R S, et al. Hermes: A fast, fault-tolerant and linearizable replication protocol[C]//Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020: 201-217.

[26] Dey S, Nair M S. Design and implementation of a simple cache simulator in Java to investigate MESI and MOESI coherency protocols[J]. International Journal of Computer Applications, 2014, 87(11).

[27] Lenoski D, Laudon J, Gharachorloo K, et al. The stanford dash multiprocessor[J]. Computer, 1992, 25(3): 63-79.

[28] Demers A, Greene D, Hauser C, et al. Epidemic algorithms for replicated database maintenance[C]//Proceedings of the sixth annual ACM Symposium on Principles of distributed computing. 1987: 1-12.

[29] Terrace J, Freedman M J. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads[C]//USENIX Annual Technical Conference. 2009.

[30] Van Renesse R, Schneider F B. Chain Replication for Supporting High Throughput and Availability[C]//OSDI. 2004, 4(91–104).

[31] Wu Y, Arulraj J, Lin J, et al. An empirical evaluation of in-memory multi-version concurrency control[J]. Proceedings of the VLDB Endowment, 2017, 10(7): 781-792.

[32] Lamport L. Time, clocks, and the ordering of events in a distributed system[M]//Concurrency: the Works of Leslie Lamport. 2019: 179-196.

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

[34] Zhou WS,Ye XJ.Implementation of ARIES Algorithms in PostgreSQL[J].Computer Engineering,2006(01):71-73+253.(in Chinese with English abstract)

附中文参考文献:

[10] 闫黎. 基于 Oracle RAC 的数据库架构分析与企业应用[J]. 计算机系统应用, 2013 (11): 200-203.

[17] 董昊文，张超，李国良，冯建华. 云原生数据库综述. 软件学报，2024, 35(2): 899-926. http://www.jos.org.cn/1000-9825/6952.htm

[23] 陈继承,李一韩,赵雅倩,王恩东,史宏志,唐士斌.一种基于共享转发态的多级缓存一致性协议[J].计算机研究与展,2017,54(04):764-774.

[24] 刘道福,陈天石,郭琦.一种无目录的共享高速缓存一致性协议[J].高技术通讯,2015,25(05):445-452.

[34] 周文胜,叶晓俊.ARIES 算法在 PostgreSQL 中的实现[J].计算机工程,2006(01):71-73+253.