探索面向分布式共享缓存架构的高性能数据库

尚碧筠[1],魏 星[2*],周士俊[2],冀文冠[2],董诗琦[2],屠要峰[2],董振江[1]
(1. 南京邮电大学计算机学院,江苏南京 210023;2. 中兴通讯股份有限公司,广东深圳 518057)

摘 要: 随着物联网和智能终端的普及,边侧产生的数据量远远超出了边缘节点的计算与存储能力,亟需云边协同处理以满足大规模数据的实时分析需求. 共享缓存架构因其计算、内存与存储解耦的特点,成为满足边缘海量数据处理需求的关键方案. 然而,在共享缓存架构中仍然存在一些亟待解决的问题. 首先,在事务处理场景中,当热点缓存数据在节点间频繁迁移时,现有系统的日志落盘机制会产生大量日志写入操作,进而影响系统性能. 此外,现有的缓存写失效机制会导致部分热点缓存数据频繁被淘汰,致使一些执行较慢的事务无法及时从共享缓存中读取目标数据,触发大量缓存重载而引发系统性能下降. 针对这些问题,本文提出了一种基于依赖表的日志延迟写入机制,通过整合多条日志写盘操作,并推迟到日志缓冲区填满或事务提交时刻,降低了日志刷写频次和写盘开销;设计了一种异步缓存延迟失效机制,通过引入异步回放失效消息、页面可见性判断及优化的缓存替换策略,有效延长缓存数据服务时间,提升了缓存命中率和系统性能. 基于这些机制,本文实现了一套高性能共享缓存数据库系统 EBASE-T. 实验结果表明:与优化前相比,EBASE-T 的吞吐量提升了 19.5%,时延降低了 13.1%,在 TPC-C(专门针对联机交易处理系统的规范)测试中,EBASE-T 相较于大多数共享缓存数据库系统表现出了显著的性能优势.

关键词: 数据库;分布式共享缓存;事务处理;日志刷写;缓存失效
基金项目: 国家重点研发计划(No.2021YFB3101101);江苏省重点研发计划重点项目(No.BE2023025)
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112(2025)02-0314-15
电子学报 URL:http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20240868

Exploring the High-Performance Database Based on Distributed Shared Cache Architecture

SHANG Bi-yun[1], WEI Xing[2*], ZHOU Shi-jun[2], JI Wen-guan[2], DONG Shi-qi[2], TU Yao-feng[2], DONG Zhen-jiang[1]
(1. *Department of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, Jiangsu* 210023, *China*; 2. *ZTE Corporation, Shenzhen, Guangdong* 518057, *China*)

Abstract: With the widespread adoption of internet of things (IoT) and smart devices, the volume of data generated at the edge has far exceeded the computational and storage capabilities of edge nodes. This creates an urgent need for cloud-edge collaborative processing to meet the real-time analysis demands of large-scale data. With the decoupling of computation, memory, and storage, the shared-cahe architecture has become a critical solution for addressing the processing requirements of massive edge data. However, there are still several issuses remained in shared-cache architecture. First, in transactional processing scenarios, when hotspot cached data frequently migrates between nodes, the log persistence mechanisms of existing databases will generate a large number of log write operations, thereby impacting system performance. Secondly, the existing cache write-invalidation mechanism can lead to frequent eviction of some hotspot cached data, causing slower transactions to fail in retrieving target data from the shared cache in time. This could trigger a large number of cache reloads, resulting in system performance degradation. To address these issues, this paper proposes a dependency-table-based delayed log flushing mechanism. By consolidating multiple log write operations and deferring them until the log buffer is full or a transaction is committed, the mechanism reduces the frequency of log flushing and the overhead of disk writes. In addition, this paper also introduces a cache delayed invalidation mechanism that incorporates asynchronous replay of invalidation messages, page visibility determination, and an optimized cache replacement. This approach effectively extends the

收稿日期:2024-09-23;修回日期:2025-01-26;责任编辑:孙瑶
*通讯作者:魏星

service time of cached data, improving cache hit rates and overall system performance. Based on these mechanisms, this paper implements a high-performance shared-cache database system called EBASE-T. Experimental results show that, compared to its pre-optimized version, EBASE-T achieves a 19.5% increase in throughput and a 13.1% reduction in latency. In TPC-C (online transaction processing system benchmarks) tests, EBASE-T demonstrates significant performance advantages over most shared-cache database systems.

Key words: database; distributed shared cache; transaction processing; log flushing; cache invalidation

Foundation Item(s): National Key Research and Development Program of China (No.2021YFB3101101); Jiangsu Key Research and Development Plan (No.BE2023025)

1 引言

近年来,随着边缘计算需求的激增,大量物联网设备和终端产生了海量实时数据. 由于端侧算力和存储资源有限,这些数据无法在边缘节点进行长期存储和深入分析,因此,需要云边协同处理来管理这些大规模数据. 为了应对边缘设备持续增长的数据规模,各大数据库厂商选择使用分布式架构部署系统,以充分利用横向扩展的硬件资源,提供高性能的数据存储和处理服务[1~4]. 在数据库发展的早期阶段,系统选型主要集中在分布式无共享架构(例如 Spanner[1] 和 CosmosDB[5])和共享存储架构(例如 Aurora[6])之间. 随后,分布式共享存储架构由于其计算与存储解耦的特性,完美契合了云环境对资源灵活管理的需求,逐渐成为云数据库系统的主流选择(例如 Amazon Redshif[7] 和 Snowflake Elastic Data Warehouse[8]). 边缘设备能够借助多个计算节点进行低时延、强一致的缓存数据访问,从而有效避免了并发访问存储层时的I/O瓶颈. 值得注意的是,当前基于共享存储架构的数据库大都采用单点写入模式,即所有的写请求事务都由单个计算节点来完成,其产生的写前日志(Write-Ahead Log,WAL)会被刷入到共享存储中,以供其他计算节点读取后构建出最新的数据页,从而提供可扩展的读访问能力[3,9].

虽然基于共享存储架构的数据库系统能够支持资源弹性扩展和故障热切换等优秀特性,但单个事务处理节点的能力往往难以应对云边协同场景下大量边缘设备所产生的写密集型负载[9]. 而且,单纯增加事务处理节点的数量并不能有效解决这一问题. 倾斜负载下的多节点并行写入会引发大量跨节点的冲突竞争,同时多节点的并行数据刷写也容易造成共享存储层的访问瓶颈. 针对这一问题,当前的主流解决方案(例如 Oracle RAC[10] 和 DB2 pureScale[11])选择在计算层和共享存储层之间引入一个共享缓存层,形成共享缓存架构,从而缓解存储层的并发写入压力,并借助缓存一致性协议[12~15]来协调多个计算节点在内存缓存中的并发访问.

在共享缓存架构中,缓存层的引入为数据库中的各个计算节点提供了一个统一且完整的数据视图,大幅扩展了单个计算节点的内存缓存空间,从而有效提升了系统的内存利用率[16,17]. 然而,共享缓存层的出现也影响了传统的事务多写处理流程,带来了一些新的问题和挑战. 首先,共享缓存层中的数据是易失的. 当一个计算节点上事务所修改的缓存数据被另一个计算节点上的事务访问时,这些修改所产生的WAL需要被强制刷写到共享存储层中,以保证在异常情况下数据库能够从有序完整的日志中恢复[10,11]. 然而,当计算节点间存在大量写倾斜事务时,热点缓存数据会被多个节点频繁访问,从而触发大量的日志刷写操作,进而影响数据库系统的性能. 其次,为了保证共享缓存架构下多计算节点间的数据访问一致性,共享缓存层还额外引入了缓存一致性协议. 在这种协议下,任何节点上针对缓存数据的写操作都会导致其他节点中缓存副本数据的失效,使得后续访问请求必须从远端缓存或存储层重新加载数据. 然而,在分布式环境中,不同节点间的处理速度往往存在快慢差距,对于一些先到达节点的请求,其所需的缓存数据很可能被后到达节点的请求导致失效,从而引发频繁的数据重载.

针对上述问题,本文探讨了一种面向共享缓存的高性能数据库框架的设计与实现,重新设计了共享缓存架构下多个节点并发执行事务时的日志刷写机制,并优化了传统缓存一致性协议所导致的频繁数据重载,从而显著提升了数据库系统的并发处理性能. 主要工作和贡献如下:

(1)提出了一种基于依赖表的日志延迟写入机制. 通过在缓存数据转移过程中引入“依赖表”,将多条日志写盘操作整合并推迟到日志缓冲区填满或事务提交时刻,降低了日志刷写频次和写盘开销,同时也保证了事务的一致性和系统的可恢复性.

(2)设计了一种异步缓存延迟失效机制. 通过异步回放失效消息、页面可见性判断及优化的缓存替换策略,避免节点缓存因写操作立即失效,从而尽可能长时间地延续缓存数据服务时间,提升缓存命中率和系统性能.

(3)基于上述两种机制,实现了一套高性能的共享缓存数据库系统 EBASE-T. 通过实验证明了基于依赖表的日志延迟写入机制和异步缓存延迟失效机制的有

效性. 在高并发的读写负载下,EBASE-T实现了吞吐量提升19.5%和时延减少13.1%. 与大多数共享缓存数据库系统相比,EBASE-T在TPC-C(专门针对联机交易处理系统的规范)基准测试中展现了显著的性能优势.

2 背景与相关工作

为了管理边缘设备所产生的海量数据,云边协同系统中数据库需要具备多节点并发访问能力,以实现水平扩展的数据管理性能. 目前,支持多节点并发访问的数据库(例如,Oracle RAC、DB2 pureScale、PolarDB-MP[18]、GaussDB-MP[19]和本文提出的EBASE-T等)普遍采用共享缓存架构,其利用各个节点中的一部分内存空间来构建一个支持并发访问的共享缓存层. 通过将共享存储中的热点数据缓存在共享缓存层中,多个节点上的事务可以直接在共享缓存中进行数据访问,并借助缓存一致性协议来保证数据的一致性,从而大幅减少数据访问开销,显著提升整体性能. 与此同时,以Aurora-MM[20]和Taurus-MM[21]为代表的数据库系统采用日志记录的回放或回滚保证数据一致性,进而协调多个节点的并发访问. 相比之下,Aurora-MM和Taurus-MM的这种日志下层设计虽然可以避免数据页在不同节点间传输以及数据页写放大等问题,但其读操作始终需要从存储层获取最新版本,因而只适用于写密集型场景,难以满足复杂的云边协同业务需求.

接下来,本节将重点分析主流基于共享缓存架构的数据库系统,并探讨其日志落盘机制和缓存失效机制对数据库性能的影响,表1为主流共享缓存数据库间对比情况.

表1 主流共享缓存数据库间对比

项目	Oracle RAC	DB2 Scale	PolarDB-MP	GuassDB-MP	EBASE-T
日志刷盘时机	事务提交+页面落盘+后台刷写+页面转移	事务提交+页面落盘+后台刷写+页面转移	事务提交+页面落盘+后台刷写+页面转移	事务提交+页面落盘+后台刷写+页面转移	事务提交+页面落盘+后台刷写
缓存管理策略	基于目录的写失效	基于目录的写失效	基于目录的写失效	基于目录的写失效	基于目录的写延迟失效
崩溃异常恢复	I/O冻结+崩溃实例日志恢复	崩溃实例日志恢复	崩溃实例日志恢复	崩溃实例日志恢复	崩溃实例日志恢复+逐出“非法”页

2.1 当前共享缓存数据库

本节将详细探讨现有共享缓存数据库系统在日志刷盘时机、缓存失效策略以及崩溃异常恢复三个方面的差异.

(1)日志刷盘时机. 事务提交和页面落盘时刷写日志是WAL的核心原则,而后台刷写WAL则是一种优化数据库性能的常用技巧. Oracle RAC、DB2 pureScale、PolarDB-MP和GaussDB-MP为避免两阶段提交的性能开销,要求数据页可被所有节点访问,允许缓存页面在节点间转移,即缓存页面从一个节点转移到另一个节点,或从一个节点回传到共享缓存. 为了确保节点崩溃的情况下仍保持数据一致性,各个计算节点需要在缓存页面转移前刷写该页面的修改WAL日志.

(2)缓存管理策略. 当前的缓存一致性通常采用基于目录的一致性协议,例如非统一内存访问(Non-Uniform Memory Access,NUMA)架构中基于HTTP的动态自适应流媒体(Dynamic Adaptive Streaming over HTTP, DASH)协议[12]和Oracle RAC中的Cache Fusion[10]. 在该协议下,计算节点中会维护着一份缓存数据的状态目录,其记录着缓存数据的分布与使用情况,以及各个节点对应的角色(即拥有者或副本持有者). 当发生并发读写时,计算节点需要根据目录中的信息快速获取可访问缓存数据的位置. 当并发读写操作产生修改时,计算节点可根据目录信息,通过广播、点对点复制或链式复制的方式来实现全局失效或者更新. 与Oracle RAC不同,DB2 pureScale、PolarDB-MP和GaussDB-MP配备有独立的共享缓存组件,用于保存缓存数据的状态目录.

(3)崩溃异常恢复. 在发生计算节点异常崩溃时,Oracle RAC、DB2 pureScale、PolarDB-MP和GaussDB-MP通常只需要回放故障节点的WAL日志. 在Oracle RAC中,每个节点只负责管理部分页面的锁请求,该节点叫作这些页面的主节点. 如果一个节点发生故障,该节点所管理的页面将变为无主状态. 在全局资源目录(Global Resource Directory,GRD)重新配置这些页面的主节点之前,整个集群将无法执行I/O操作或处理新的锁请求. 而DB2 pureScale、PolarDB-MP和GaussDB-MP借助共享缓存组件避免了全局冻结,在计算节点故障时可直接识别需要恢复的页面. 在EBASE-T中,只需回放故障节点的WAL日志. 得益于全局缓存服务节点(Global Cache Service,GCS)的存在,EBASE-T避免了全局I/O操作冻结和锁冻结的开销. 此外,EBASE-T引入日志延迟写入机制(详见下文)后需逐出“非法页”,但这一过程支持高并发操作,因此产生的额外开销较小.

2.2 缓存数据流转引发频繁的日志落盘

在支持多写的共享缓存数据库中,一个节点可以读取其他节点缓存的热点数据来响应自身的访问请求. 如图1所示,节点1将数据页X拉取到了本地共享

缓存中,并将其修为X'. 随后,节点N可以直接从节点1的缓存中读取修改后的页面X',并进一步将其修改为X''. 直观来看,这样的缓存间互相访问可以避免节点1修改完的数据页X'先落盘,然后再被节点N从硬盘加载,从而显著提升访问效率. 但考虑到共享缓存中页面的易失性,当一个节点中的缓存数据被读取到其他节点前,现有支持多写的共享缓存数据库系统会先将该节点上被读取数据的修改日志进行落盘,从而为系统的异常恢复提供保证. 例如,在图1中节点N读取节点1中修改后的数据页X'前,节点1先将数据页的修改日志($X\rightarrow X'$)写入硬盘. 值得关注的是,这样的日志持久化操作会频繁地发生在多节点并发修改热点数据的场景下,从而产生大量的I/O开销,进而导致系统性能受限. 因此,当前亟需设计一种针对节点间缓存数据流转的高效日志落盘机制,从而降低系统的I/O开销. 为了降低缓存页面转移带来的日志刷写开销,Oracle RAC还引入了Smart Fusion Block Transfer机制[22]. 通过将日志传输到Exadata(Exadata Database Machine)中,计算节点不需要等到I/O结束便可转移缓存页到远端节点. 然而,当远端节点要修改该页面时,其仍然需要等待该页面涉及到的I/O操作完成,因而还是会影响到整个系统的并发读写性能.

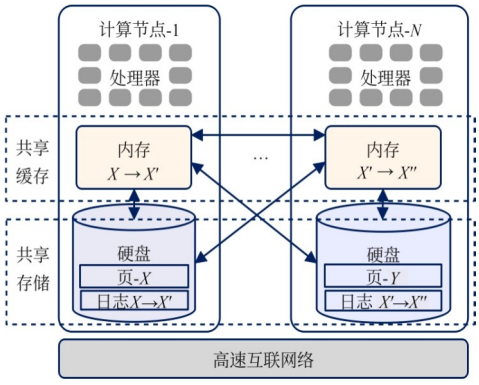


图1　页面刷盘机制

2.3　缓存写失效带来不必要的数据重载

在基于目录的缓存一致性协议中,当某个节点要对缓存页进行修改时,该节点须通知主/目录节点去查询缓存此页的所有节点,并对这些节点发送缓存失效信息. 当收到所有相关节点的失效确认信息后,该节点才可进行缓存页面的修改. 以图2为例,当节点1收到修改页面X的事务请求后,其首先请求主/目录节点查询缓存页面X的相关节点(即节点2),并将页面X的失效信息转发给相关节点. 节点2收到失效消息后便开始失效缓存页面X(随后请求无法访问该缓存页面),然后再通知节点1失效完成. 当节点1收到节点2的失效确认后,该节点便修改缓存页面X为X'. 最后,节点1完成事务提交. 但值得注意的是,图2中节点1的事务提交序列号(Commit Sequence Number,CSN)为300,而节点2上事务发起的快照为200. 由此可见,在读已提交的隔离级别下,节点2上的事务仍然可以使用本地缓存的页面X,而不需要重新加载. 因此,当前的写失效机制中还需配合一些精细化的失效和可见性判断方案,从而避免并发场景下缓存页面的频繁重加载.

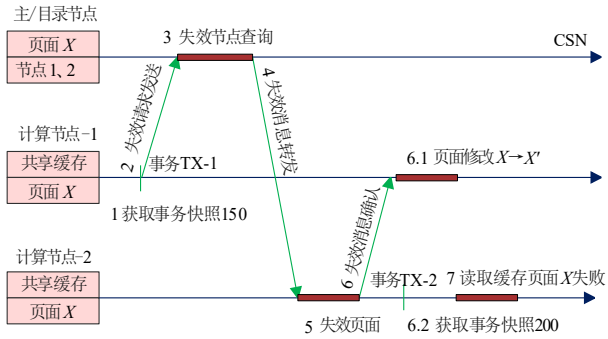


图2　写缓存失效机制

针对上述日志频繁刷盘和缓存非必要重载的问题,本文分别提出了日志延迟写入机制和异步缓存延迟失效机制,并将其集成到共享缓存数据库EBASE-T中,有效提升了倾斜负载下的并发读写性能.

3　系统架构

本节首先介绍了共享缓存数据库系统EBASE-T的系统架构,然后详细阐述了EBASE-T处理读写事务请求的流程.

EBASE-T的系统架构如图3所示,分为如下5个单元.

(1)计算节点(Computing Node). 计算节点是系统中执行事务处理的主要工作单元,包含SQL引擎,本地事务引擎和本地共享缓存管理器. SQL引擎负责语句处理. 本地事务引擎负责将全局事务信息同步到本地、更新本地CSN日志,并维护WAL缓冲区. 本地共享缓存管理器通过一致性协议,将多个计算节点的缓存空间整合成一个大的共享缓存区域,并对外提供支持一致性的并发访问接口.

(2)高性能网络(High-performance Network). 高性能网络负责提供高效的消息及数据传输服务. 在高性能网络中,各个节点间构建了基于远程直接数据存取(Remote Direct Memory Access,RDMA)的RC连接[18],并借助任务队列管理和轮询方式降低网络传输任务处理时延. 同时,该套网络服务还提供了完善的错误处理和资源管理机制,以提升整体通信效率和安全性.

(3)全局事务管理节点(Global Transaction Management,GTM). GTM是系统的核心事务控制节点,负责生

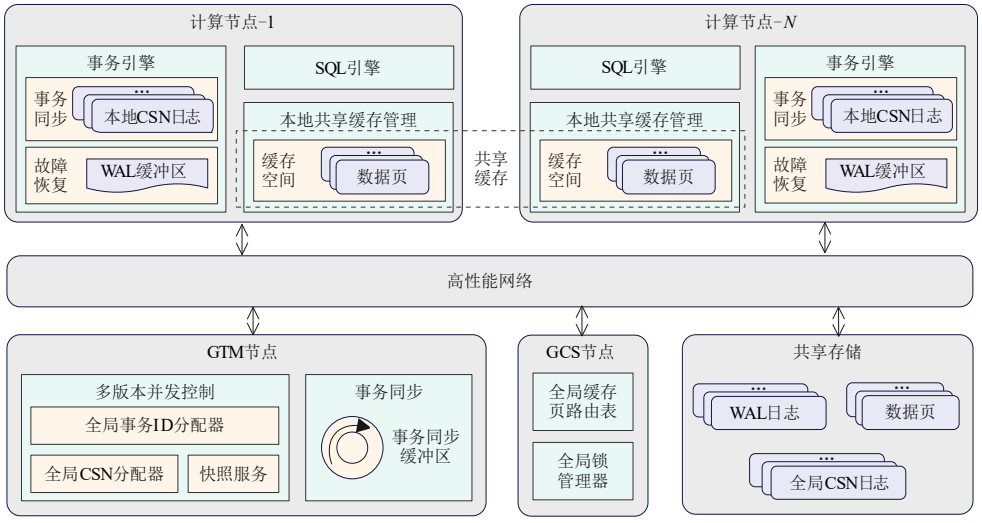


图3 EBASE-T系统架构图

成全局唯一的事务标识符(Transaction ID,XID)和全局提交顺序号(Commit Sequence Number,CSN). GTM还提供快照服务以确保分布式事务的一致性. 同时,GTM也管理全局事务信息的同步,协调各计算节点间的事务处理,确保全局数据的一致性.

(4)全局缓存服务节点(Global Cache Service,GCS). GCS主要负责全局缓存的一致性管理和页面的高效调度. GCS包含多个核心组件:页面元数据管理器负责管理包括页面所有者位置等信息;页请求路由器负责识别计算节点的页面请求,查找相应的元数据,并将请求转发至正确的目标节点;页面失效管理器处理来自计算节点的修改页面集合,通过过滤后将失效消息组包并分发到相关计算节点;全局锁管理器处理除页面锁以外的其他并发锁请求,确保全局资源的有效调度;集群节点管理器负责管理集群中的节点,包括节点增减、宕机检测和恢复节点选择等操作.

(5)共享存储(Shared Storage). 共享存储负责存储所有计算节点需要访问和持久化的数据. 为了确保数据的完整性和一致性,数据页和WAL日志必须存储在共享存储中,以便在节点崩溃恢复时,恢复节点可以读取崩溃节点的日志. 此外,在计算节点执行检查点时,全局提交顺序号的记录文件(CSN Log)和事务信息的记录文件(Commit Log,CLog)也会被持久化到共享存储,以支持崩溃恢复期间的数据访问.

3.1 写事务流程

在EBASE-T中,当计算节点执行写事务时,其会与多个节点通信(包括GTM和GCS),并修改相应的元数据. 如图4所示,当计算节点1的写事务开始时,其首先向GTM请求事务号(XID)和快照信息. 随后,节点1会尝试在本地共享缓存中查找对应页面. 如果本地缓存中的页面已经是全局最新版本的持有者(即页面的owner),计算节点1即可直接在本地获取该页面的排他锁(eXclusive lock,X锁),并进行写操作. 否则,本地缓存的页面只是副本页,节点1还需要向GCS节点请求获取页面的X锁和最新版本的页面. GCS在接收到页面X锁请求后,会在全局路由表中查找该页面当前的所有者. 此时可能出现以下两种情况:(1)若页面owner不在集群共享缓存中,GCS则在路由表中将计算节点1标记为该页面的owner. 随后,GCS会通知计算节点1从共享存储中读取该页面. (2)若页面owner在集群共享缓存中,GCS发现页面的当前所有者为计算节点2. 此时,GCS将路由表中的页面所有者更新为计算节点1,以便后续的读请求指向计算节点1. 同时,GCS向计算节点2发送页面所有权转移请求. 计算节点2在收到该请求后,会将本地页面标记为副本页,并通过RDMA将页面发送至计算节点1,由节点1进行相应的写操作. 此时计算节点2仍然持有该页面的副本,并对某些快照仍然有效. 这使得在读取该页面时,计算节点2不必请求最新的副本,从而减少了网络传输. 然而,当计算节点1获取提交的CSN号后,计算节点2中新获取的快照就需要看到该写事务对页面的修改. 因此,计算节点1在事务提交时需要向GCS发送页面失效消息. GCS根据路由表,将页面失效通知发送到所有持有未失效副本的节点. 有关页面失效机制的详细信息,请参见后文.

3.2 读事务流程

读事务的流程类似于写事务,但由于读事务不涉及页面修改,因此无需获取事务号或提交CSN号,也无需发送页面失效消息. 如图5所示,在收到读事务后,计算节点会检查本地缓存中是否存在所需页面. 如果

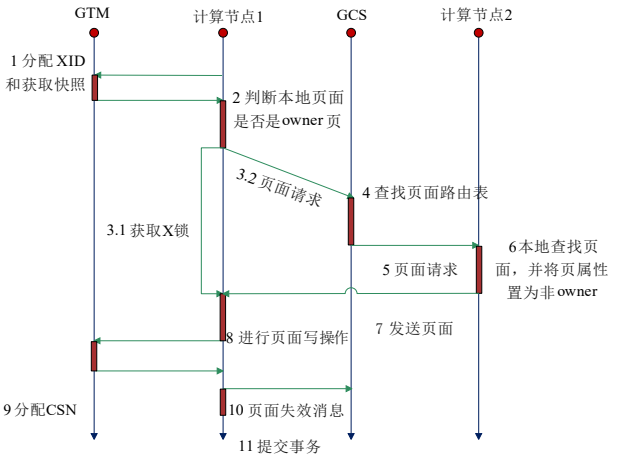


图4　EBASE-T写流程

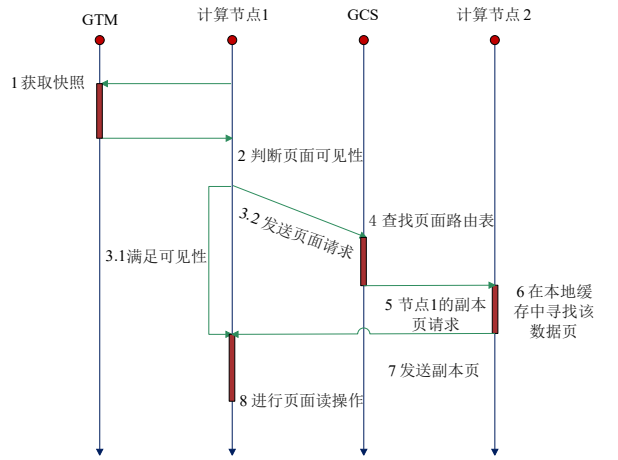


图5　EBASE-T读流程

页面不存在,计算节点会直接向GCS发送副本页请求;如果页面存在,计算节点还需要判断该页面是否对当前事务的快照可见(参见后文缓存延迟失效机制中的页面可见性判断机制). 若页面满足可见性要求,则计算节点可以直接使用本地页面;否则,计算节点向GCS发送页面请求. 当GCS收到副本页请求消息时,它会在路由表中查找该页面的owner. 如果发现集群中无该页面的owner,则将计算节点1指定为该页面的owner,随后GCS会通知节点1从共享存储中读取页面. 若路由表中已存在该页面的信息,GCS则将副本页请求转发给页面的当前owner节点. 例如,图5中计算节点2收到请求后便从本地缓冲区中找到该页面,并将副本发送到节点1. 节点1在收到副本后进行后续的读取操作. 由于节点1在读取页面前先获取了快照SNAPSHOT-A,并从节点2拷贝了最新的页面,因此,该页面必然满足SNAPSHOT-A的所有访问需求.

4　关键技术

本节将重点介绍EBASE-T的日志延迟写入机制和缓存延迟失效机制. 通过这些关键技术,EBASE-T有效避免了多节点并发访问共享缓存所带来的日志频繁刷写和不必要的缓存数据重载.

4.1　日志延迟写入机制

当计算节点间发生缓存页面转移时,EBASE-T数据库系统并不会触发WAL日志落盘,而是将页面修改相关的日志依赖信息一并转发到目标节点上,从而缓解日志频繁刷写这一问题.

4.1.1　依赖关系构建

在共享缓存架构下,任意节点上的页面修改都会产生WAL日志. 但各节点所产生的日志序列号(Log Sequence Number,LSN)无法标识节点间的日志依赖次序. 针对这一问题,本文借鉴文献[23]提出了全局序列号(Global Sequence Number,GSN)来规约多节点的日志次序,并将其记录在每条日志和每个数据页中. 为了方便表述,本文约定数据页面P中的GSN记为$p_{\mathrm{GSN}}(P)$,日志记录R中的GSN记为$r_{\mathrm{GSN}}(R)$,当前节点N中的最大GSN记为$n_{\mathrm{GSN}}(N)$. 在系统初始阶段,所有GSN记录都为0. 随着系统的运行,GSN的变化规则如下:

规则1　在节点A对远端读取的数据页P进行修改前,令$n_{\mathrm{GSN}}(A)=\mathrm{Max}\{n_{\mathrm{GSN}}(A),p_{\mathrm{GSN}}(P)\}$.

规则2　当节点A完成页面修改后,令$r_{\mathrm{GSN}}(R)=p_{\mathrm{GSN}}(P_1)=p_{\mathrm{GSN}}(P_2)=\cdots=p_{\mathrm{GSN}}(P_n)=++n_{\mathrm{GSN}}(N)$.

基于上述规则,节点间缓存数据页的转移和缓存数据页本身的修改都会依次推高日志记录的GSN,使得对于一个数据页的所有关联日志记录都可以确立次序关系.

4.1.2　依赖信息管理

为了管理不同事务和页面修改操作所依赖的日志记录,每个计算节点中都存放着针对特定事务的提交依赖表(Commit Dependency Table,CDT)和特定页面的页面修改落盘依赖表(Modify Dependency Table,MDT),其分别记录着事务提交和页面修改落盘操作所需依赖的节点日志. 例如,图6中节点1的事务TX-2将页面P'修改为P'',则其对应的提交依赖表CDT(TX-2)中记录着该事务提交前,需要节点2上GSN为30的日志落盘. 与此同时,节点1中页面落盘依赖表MDT(P'')记录着页面落盘前,不仅需要节点1上GSN为GSN(P'')的日志落盘,还需要节点2上GSN号为30的日志落盘. 最后,每个节点的内存中还管理着一个日志刷写表(Log Flushing Table,LFT),其记录着各个节点上的日志刷写最新进展(即最新落盘日志的GSN值). 当共享缓存中的页面在不同节点间流转时,该页面对应的MDT表也会随之转移到目标节点,并随着页面的修改,MDT表中

的GSN值会叠加到对应事务的CDT表中. 例如,图6中节点1的页面P'被事务TX-2修改后,页面的MDT表便更新了CDT(TX-2)内的GSN值. 在实际使用过程中,MDT表大小主要取决系统中的节点个数. 一个节点占一条记录,共计8字节. 因此,相较于页面转移,MDT表所带来的额外传输带宽可以忽略不计.

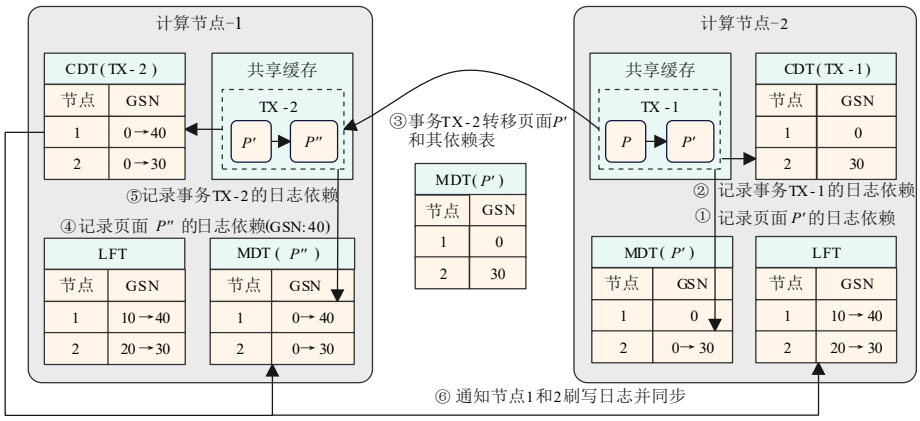


图6 日志延迟写入机制

4.1.3 延迟写入策略

基于上述依赖信息管理机制,本文进一步提出并实现了一种日志延迟写入策略. 该策略通过在页面修改阶段更新日志依赖,在页面转移阶段传递日志依赖,以及在事务提交阶段刷写依赖日志,成功地将原有页面转移过程中的日志落盘动作推后到事务提交或日志缓冲区写满的阶段,使得一次落盘开销被积累下的多条日志记录所均摊. 具体实现如下:

(1)页面修改阶段. 在事务修改数据页时,其所产生的日志首先会按次序刷入到本地WAL缓冲区,然后再将该日志的GSN号记录到对应的CDT和MDT中,同时基于朴素的思想——事务提交依赖于该事务修改的页面可落盘,因此,将MDT(P)叠加到CDT(TX-2)上,即对于每个节点下标i,MDT[i]=Max(MDT[i],CDT[i]). 例如,图6中节点2上事务TX-1将页面P修改成P',其对应日志的GSN(为30)被分别记录到CDT(TX-1)和MDT(P)中的节点2位置,因页面P非脏页,故MDT(P)为全0,因此无需叠加到CDT上.

(2)页面转移阶段. 页面在节点间转移时,会携带其对应的MDT. 例如,图6中节点1上的事务TX-2要将页面P'修改成P'',其首先要将页面P'和对应的MDT(P)转移到节点1上. 随后,该事务修改页面P',其修改日志的GSN(为40)也会被记录到MDT(P)和CDT(TX-2)中,且将MDT(P)叠加到CDT(TX-2)上.

(3)事务提交阶段. 在事务提交时,会对比本地的CDT和LFT. 如果LFT显示各节点已经将CDT中记录的依赖日志刷盘,那么该事务可以直接提交;否则,该事务还需通知相关节点完成其所依赖日志及其前序日志的刷写. 例如,当图6中事务TX-2进入提交阶段后,发现LFT中记录的各节点日志刷写进度落后于其所依赖的日志记录(GSN 40和30),此时,该事务便通知节点1和2分别将这些日志及其前序日志进行刷写. 当所有节点完成刷写并同步状态信息后,该事务便可以进行提交. 值得注意的是,当数据页对应的MDT中所有依赖日志均已刷盘后,该数据页也可以被写入存储层. 该策略通过将多条日志写盘操作整合到一次I/O操作中,显著减少了单次写盘的开销,同时保证了事务提交的安全性和一致性.

4.1.4 策略优势分析

为了清晰地展示日志延迟写入策略的性能优势,本文从同步次数(即日志刷盘操作)的角度展开进一步分析. 假设共享缓存系统中共发生了N(Trans)条事务,每条事务平均需要跨节点传输R个数据库页,则整个系统中发生的数据页传输为N(Convey)=N(Trans)×R. 同时,假设页面传输完成时相关WAL日志已落盘的概率为P(Convey),事务提交时相关WAL日志已刷盘的概率为P(Commit). 在以Oracle RAC为代表的传统共享缓存系统中,每次页面转移和事务提交都要确保所有相关页的WAL日志都已同步刷写到共享存储中,因此,系统中的同步次数NSync(Oracle)可表示为

$$\mathrm{NSync(Oracle)}=N(\mathrm{Trans})+N(\mathrm{Trans})\times R\times(1-P(\mathrm{Convey})) \tag{1}$$

相比之下,EBASE-T在页面转移前无需确保日志已同步刷盘,仅需在事务提交前完成刷盘,因而整个系统同步次数NSync(EBASE-T)可表示为

$$\mathrm{NSync(EBASE\text{-}T)}=N(\mathrm{Trans})+N(\mathrm{Trans})\times(1-P(\mathrm{Commit})) \tag{2}$$

通过对比,两种日志刷写机制的同步次数差值为NSync(Oracle)减去NSync(EBASE-T),可得到N

(Trans)×(*R*×(1−*P*(Convey))−(1−*P*(Commit))). 在共享缓存系统中,跨节点数据页传输数量*R*≥1,且事务提交时的日志落盘概率*P*(Commit)通常高于页面转移时的日志落盘概率*P*(Commit),由此可得*R*×(1−*P*(Convey))−(1−*P*(Commit))必然大于0. 因此,当事务总量*N*(Trans)>0时,NSync(Oracle)减去NSync(EBASE-T)必然大于0. 这表明,与传统共享缓存系统相比,EBASE-T的日志延迟写入策略显著减少了同步落盘操作,进而提升了系统的并发性能.

4.1.5　崩溃异常恢复

在日志延迟写入策略的实施过程中,虽然显著提高了系统性能,但在WAL原则的遵循方面也引入了新的挑战,可能导致崩溃异常场景的出现. 针对这些问题,EBASE-T系统对原有的日志结构和落盘机制进行了针对性的改进,以确保系统的可靠性.

崩溃异常分析:在传统共享缓存数据库(例如Oracle RAC)中,页面修改日志必须在页面转移前完成落盘. 因此,对于同一页面的两次修改,如果后一次修改日志已经落盘,则其前序日志必然也已完成落盘. 这一约束确保了日志的可回放性,因为所有已刷盘日志的依赖关系均得到满足. 然而,在采用日志延迟写入策略的EBASE-T系统中,页面修改日志可以在页面转移前暂不刷盘,从而打破了上述约束,这可能会导致出现不可回放的日志问题. 例如,图7中GSN-50、GSN-53、GSN-55、GSN-56是同一个页的修改日志. 但若节点1在GSN-53日志未落盘的情况下就发生崩溃,则节点2上GSN-55日志即使已经落盘,但其仍为不可回放日志,因为其前序日志GSN-53不存在. 此外,在页面落盘前修改日志必须落盘的规则下,前序日志GSN-53的消失也会使得日志GSN-55的对应修改页面无法落盘,从而成为非法页.

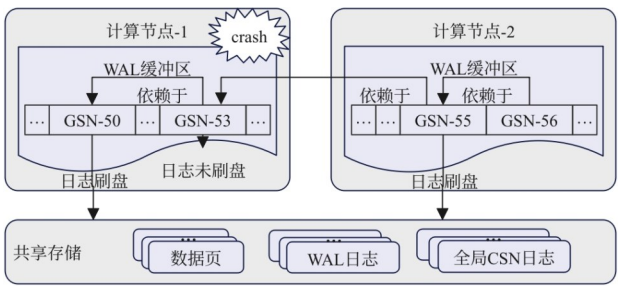


图7　日志异常回放

异常恢复机制:为解决上述问题,EBASE-T系统引入了多种恢复机制,以确保系统在崩溃情况下的恢复能力.

(1)日志依赖记录. 在日志存储中新增一个字段,用于记录页面上一次修改所产生的日志存储位置. 如果上一次修改的日志未落盘,则该条日志及其该页后续修改产生的日志不进行回放. 值得注意的是,这样的依赖关系只需在页面转移的第一条日志进行记录. 这是因为在同一个节点内,如果后续日志已经落盘,那么前面日志肯定也落盘. 另外,在写页面转移后的第一条修改日志时,如果发现该页面上一次修改的日志已落盘,那么也无需记录依赖关系.

(2)非法页识别和逐出机制. 具体来说,若记MDT(*P*)[*N*]表示页面*P*在落盘时节点*N*的日志刷写位置,CDT(TX)[*N*]表示事务TX在提交时节点*N*的日志刷写位置,MAX_FLUSH_GSN表示故障节点已刷写日志的最大值,则非法页和需要回滚事务定义为

非法页:若MDT(*P*)[*N*]>MAX_FLUSH_GSN,则页面*P*为非法页.

需要回滚事务:若CDT(TX)[*N*]>MAX_FLUSH_GSN,则事务TX为需要回滚的事务.

在对各存活节点进行非法页识别和逐出时,EBASE-T充分利用了数据并行性,不同进程负责本地缓冲区的不同部分,以提高处理效率.

(3)页面恢复与事务回滚. 当节点发生崩溃时,其他存活节点可以继续处理事务和执行I/O操作,只有待恢复页面的访问请求会在该页面恢复完成前被阻塞. 其相应的页面阻塞标识被记录在GCS的页面元数据上. 待恢复页面可分为两类:一类是失主页面,通过访问GCS节点,系统可以知道哪些页面的owner属于崩溃节点,并将其列为待恢复页面. 另一类是非法页面,对于那些被逐出的非法页,其对应磁盘存储中的页面属于旧版,因而也需要恢复. 针对这两类待恢复页面,EBASE-T系统一方面上报GCS将待恢复页面设置为访问阻塞状态,防止事务对这些页面的访问操作;同时也需要报告负责页面恢复的节点,确保恢复操作能够有序进行. 此外,由于EBASE-T系统采用MVCC机制,并不包含undo日志,因而大幅降低了事务回滚的开销.

4.2　缓存延迟失效

在共享缓存数据库中,系统会利用共享缓存空间来尽可能多地缓存热点数据页,从而避免频繁访问存储层. 但当采用传统写失效机制(详见2.2节)后,写操作和失效操作需要同步完成. 即共享缓存中某个节点上的数据页被修改后,其他节点上的相应缓存数据页也会立即失效,这种同步失效机制可能导致其他节点无法继续服务较早发生的事务请求,并引发频繁的缓存数据页重新载入. 为了解决这一问题,EBASE-T设计了缓存延迟失效策略,使写操作仅传递失效消息,而不强制立即失效其他节点上的相关缓存页面,从而尽可能地服务更多事务请求.

4.2.1　页面失效操作

在EBASE-T的缓存延迟失效机制中,当事务完成

页面修改后，便会生成相应的失效消息，并在失效消息中记录此次修改的CSN号及被修改页面. 例如，图8中写事务在修改完页面P_1后，从GTM节点获取相应的CSN号100，并形成失效消息[CSN-100，P_1]. 接下来，该条失效消息将被提交给GCS节点，由GCS节点依据缓存页路由表将失效消息转发至被修改页面副本所在的节点. 当GCS节点完成失效消息转发后，计算节点便可完成事务提交. 例如，在图8中，页面P_1在节点2上有一个副本，因此，针对页面P_1的失效消息将由GCS节点转发至节点2. 值得注意的是，当副本所在节点收到失效消息后，其并不会直接失效相关页面，而是将消息写入到队列中，随后由相关工作进程逐步回放消息到可见性判断表中. 可见性判断表中每个页面的默认Max_CSN为−1，表示当前页面未被执行任何失效操作. 在失效消息回放过程中，如果可见性判断表中的页面与失效消息有关，则该页面对应的Max_CSN可以更新为该条消息中的CSN，从而表示该页面已经被该CSN号的事务进行了失效操作. 以图8为例，当回放到失效消息[CSN-100，P_1]时，页面P_1的Max_CSN应该被置为100，从而表示该页面已经被CSN-100的事务进行了失效.

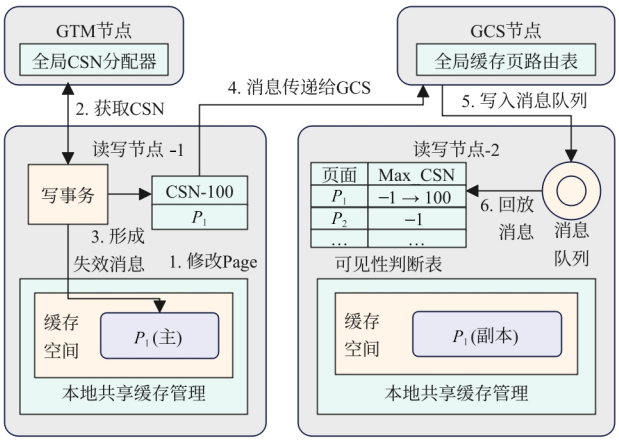


图8 缓存页面延迟失效机制

4.2.2 缓存替换规则

对于那些从远端拉取的数据页，当前节点会将其换入到本地共享缓存空间中，并记录到可见性判断表中(Max_CSN值记为−1). 同时，各节点为缓存中的数据页维护一个换出队列，并依据如下规则将其实时排序. 当节点缓存空间使用超过阈值时，位于缓存队列头部的数据页将会优先被淘汰.

规则1 在换出队列中，将已进行过失效消息回放的数据页(即Max_CSN值不为−1)放置在前面，并将其按照Max_CSN值由低到高排序.

规则2 在换出队列中，将那些未进行过失效消息回放的数据页(即Max_CSN值为−1)放置在后面，并按照最近访问次数由低到高排序.

基于上述替换规则，那些较早被失效且长时间未被访问过的数据页将优先被替换，确保缓存空间中始终保留热点数据页，从而提升系统的缓存命中率和整体性能.

4.2.3 页面可见性判断

基于上述页面失效策略，如果事务请求发现目标缓存页面P_i的Max_CSN值非−1，并且大于当前事务的版本号X，说明产生失效消息的事务在当前事务开始后提交，因而当前事务可以访问目标缓存页面P_i(算法1，第2行). 反之，如果缓存页面P_i的Max_CSN小于当前事务版本号X，则表明缓存页面在事务开始前已经被失效，从而不可见(算法1，第3行). 值得注意的是，如果缓存页面P_i的Max_CSN值为−1，事务请求还需去失效队列Q中判断是否存在页面P_i未被回放的失效消息. 如果不存在，则缓存页面可以被直接访问(算法1，第6行). 如果存在，则还需进一步对比队列中失效消息的CSN号是否大于当前事务版本号X. 一旦失效消息的CSN大于当前版本号X，则页面可以被访问(算法1，第8行)，反之则说明缓存页面不可见(算法1，第10行).

算法1 页面可见性判断

输入: 页面P_i，判断表T，失效队列Q，失效消息M，事务请求的版本号X
输出: 页面可见返回True，否则返回False
1. Max_CSN=T.Get(P_i)//从可见性表中获取目标缓存页面的Max_CSN值
2. if Max_CSN !=−1 && Max_CSN>X then//目标页面已经事务失效过，且事务CSN号大于当前事务版本号
3. return true
4. if Max_CSN !=−1 && Max_CSN<X then//目标页面已经事务失效过，且事务CSN号小于当前事务版本号
5. return false
6. if $Q(P_i)$ == NULL//消息队列中不含目标缓存页面的失效消息
7. return true
8. if $Q(P_i)$.CSN>X//消息队列中目标缓存页面的失效消息CSN号大于版本号
9. return true
10. else return false

5 实验分析

本节首先对EBASE-T的日志延迟写入和页面缓存延迟失效机制进行了有效性验证实验和分析，然后通过整体性能测试，将EBASE-T与开源共享缓存数据库Citus[24]、经典商用共享缓存数据库System-A进行实验对比和分析.

5.1 实验设置

5.1.1 硬件配置

本实验默认运行在8个节点的集群之上,其中每个节点的参数配置如表2所示. 在8个节点中,其中计算节点占4个,存储节点占2个,GCS和GTM节点各占1个. 集群中的节点由100 Gbit/s的RDMA网络相连接. 与此同时,每个计算节点中划分出64 GB内存空间用作构建共享缓存层.

表2 硬件配置

名称	规格参数	数量
操作系统	CentOS 7.7	-
CPU	Intel(R) Xeon(R) Gold 6230N CPU @ 2.30 GHz	2
DRAM	32 GB DDR4-2400	12
NVMe SSD	INTEL® SSD DC P4610	8
Network	ConnectX-5 InfiniBand	1

5.1.2 工作负载

本实验采用Sysbench负载工具[25]来模拟多节点并发读写负载,从而验证日志延迟写入机制和缓存延迟失效机制的有效性能. Sysbench的模拟测试数据集存储在一张200多万行的宽表中,其一行数据包含数百个字段,共计消耗240 GB存储空间. 针对这张数据表,每个计算节点随机执行电信业务中抽象出来的四条读写事务.

(1)写事务-1:UPDATE sysbench_table SET bid=bid+:delta WHERE aid=:aid;

(2)写事务-2:UPDATE sysbench_table SET abalance=abalance+:delta WHERE aid=:aid;

(3)读事务-1:SELECT bid FROM sysbench_table WHERE aid=:aid;

(4)读事务-2:SELECT abalance FROM sysbench_table WHERE aid=:aid.

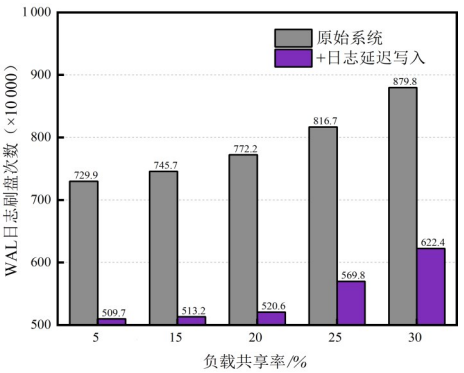
其中,字段aid为数据表sysbench_table中的主键,其他字段(例如,bid、abalance)为非主键. 通过选取$X\%$的aid数值被多个计算节点共同访问,从而触发页面转移和缓存失效. 剩下的$1-X\%$的aid数据会被均匀分配给多个计算节点. 此外,本文还进一步使用TPC-C负载[26]来对比EBASE-T和其他相似数据库间的性能差异. TPC-C实验共设置了15 000个warehouse. 访问请求通过访问代理被随机转发到不同计算节点上.

5.2 有效性分析

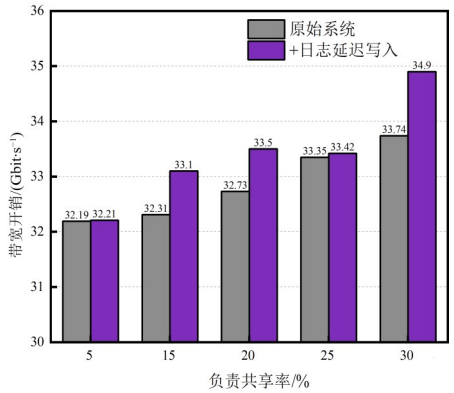
为了验证日志延迟写入机制和缓存延迟失效机制的有效性,本组实验从I/O、带宽以及缓存命中率等角度出发,对比引入上述机制的前后差异. 此外,本组实验还进一步分析上述两种机制相结合后是否能够给EBASE-T系统带来时延和吞吐上的收益.

5.2.1 日志延迟写入机制

如图9所示,本组实验对比了EBASE-T系统采用日志延迟写入机制前后的日志刷写次数和网络带宽开销的变化. 当未采用日志延迟写入机制时,系统的日志刷写次数随着共享率的提升显著增多. 这是因为随着共享率增大,越来越多的缓存页面会被多个计算节点访问,从而容易引发页面流转,进而导致频繁的日志刷写. 当采用延迟写入机制后,页面的转移只会携带"依赖表",并不会触发日志写盘,因而刷盘次数波动不大. 例如,在30%共享率下,未采用延迟写入机制的刷盘次数约为8 798 000次,而采用后则约为6 224 000次. 值得注意的是,图8(*b*)显示采用延迟写入机制后,带宽开销有着略微上涨. 这是因为延迟写入机制虽然避免了页面转移过程中的刷盘,但是额外带来了依赖表传输,因而会消耗一部分网络带宽.



(*a*) 日志刷写次数对比



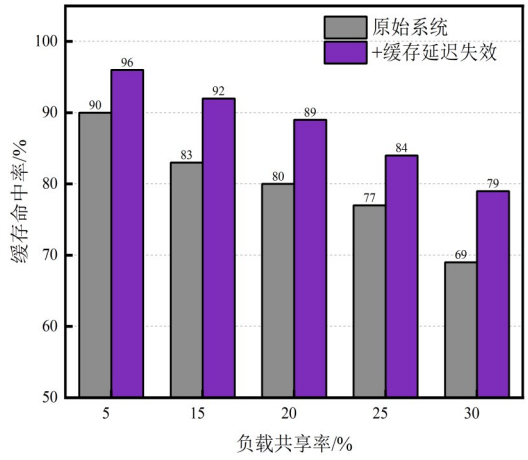
(*b*) 网络带宽开销对比

图9 日志延迟写入机制验证

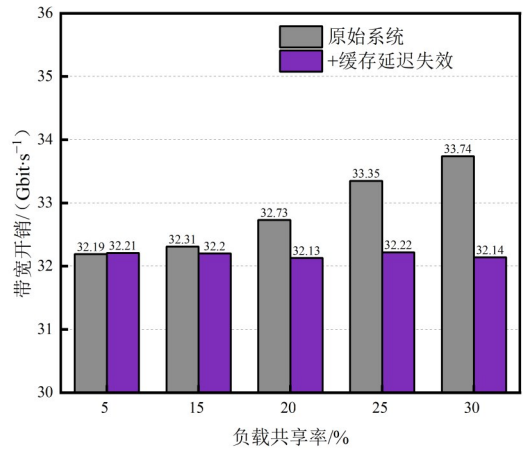
5.2.2 缓存延迟失效机制

如图10所示,本组实验进一步研究了缓存延迟失效机制对EBASE-T的缓存命中率和带宽开销影响. 直观来看,通过采用缓存延迟失效机制,EBASE-T的缓存命中率显著提升,带宽开销也会降低. 尤其在高共享率下,

缓存延迟失效机制带来的优势更为明显. 例如,在30%共享率下,缓存延迟失效机制使得缓存命中率由69%提升到了79%,带宽开销由33.74 Gbit/s降至32.14 Gbit/s. 这是因为缓存延迟失效机制提升了缓存页面的服务时间,避免了大量的缓存数据重载.



(*a*) 缓存命中率对比



(*b*) 网络带宽开销对比

图10 缓存延迟失效机制验证

5.2.3 融合验证

接下来,本组实验将从带宽、吞吐以及时延角度来进一步探讨上文优化机制(即日志延迟写入和缓存延迟失效)融合后的有效性. 图11(*a*)分析了日志延迟写入和缓存延迟失效技术综合后的带宽消耗情况. 在15%和30%的负载共享率下,与原始系统相比,EBASE-T的带宽总体变化不大,但与只采用日志延迟写入技术相比,有着明显的降低. 这是因为缓存延迟写入技术避免了频繁的数据重新加载,抵消掉了一部分日志延迟写入技术中的依赖表传输开销. 图11(*b*)分析了不同共享缓存空间大小给系统吞吐带来的影响. 在共享缓存系统中,数据页会尽可能地放置在共享内存中. 当共享内存空间越小,系统就越会进行频繁的缓存换入换出,从而导致性能下降. 虽然日志延迟写入技术引入了额外的依赖表,但其空间占用只有数十字节(原理详见4.1节),因而对性能影响不大. 此外,当进一步引入缓存延迟失效机制后,有限缓存空间的数据页可以更长地服务于各节点请求,因而带来的更高的性能. 例如,在30%的共享率,当分别采用了日志延迟写入和缓存延迟失效机制后,系统吞吐分别提升11%和19.5%. 与此同时,如图11(*c*)和图11(*d*)所示,在15%共享率,当采用日志延迟写入机制后,整个系统吞吐从每秒处理56 692条事务提升到了61 285条事务,平均每条事务的处理时延则从7.52 ms降低到了7.23 ms. 这是因为日志延迟写入机制可以减少多节点并发访问共享页面所导致的WAL日志写入频次. 当在30%的共享率下,日志延迟写入所带来的性能优势则更为明显. 值得注意的是,当进一步采用缓存延迟失效机制,整个系统可以再次获得性能提升. 这是因为缓存延迟失效可以大幅降低并发负载下共享数据页的重新载入次数.

5.3 系统加速对比

5.3.1 加速比

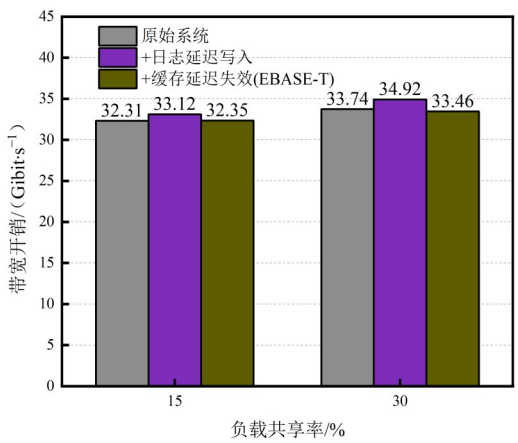
图12展示了在Sysbench负载下,EBASE-T(包含日志延迟写入机制与缓存延迟失效机制)与其他数据库的性能对比. 对比对象选取Citus、不采用本文优化机制的EBASE-T以及一款商用的共享缓存数据库系统System-A. 在15%和30%的共享率下,随着节点数量的不断增多,相较于单节的加速比也随之增大. 但值得注意的是,EBASE-T的加速比一直领先于Citus、未优化的EBASE-T以及商用共享缓存数据库System-A,展现了良好的扩展性. 由此可见,当采用日志延迟写入和缓存延迟失效机制后,页面转移带来的日志刷盘和缓存失效导致的页面重载都被大幅降低,使得EBASE-T有着较优的加速比.

5.3.2 吞吐对比

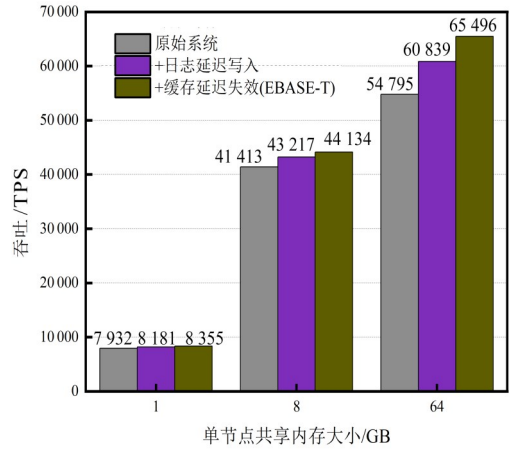
与此同时,图13进一步对比了不同系统在不同负载下的吞吐性能. 在前文的Sysbench负载下,相较于开源数据库系统,EBASE-T依然表现出了较强的吞吐能力. 但和工业界最顶尖的共享缓存数据库System-A相比,还是表现出较小的性能差距. 此外,在经典的TPC-C负载下,EBASE-T的吞吐性能分别为Citus和EBASE-T(未优化)的1.68倍和1.24倍. 和System-A相比,EBASE-A仍然可以达到其91.1%的负载处理吞吐.

5.3.3 恢复能力

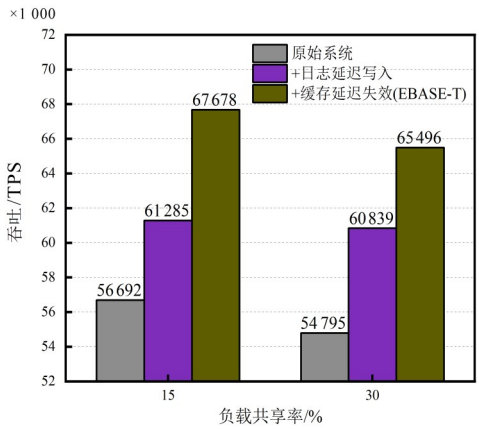
图14进一步评估EBASE-T的故障恢复性能. 此组试验搭建了一个包含两个计算节点的集群,其共同运行30%共享率的Sysbench负载. 为了模拟崩溃场景,此组实验随机终止了计算节点2的运行,并立即尝试拉起



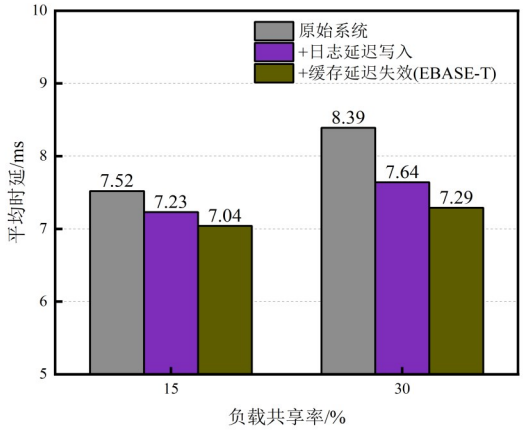
(*a*) 不同共享率下带宽对比



(*b*) 不同共享内存下的吞吐对比

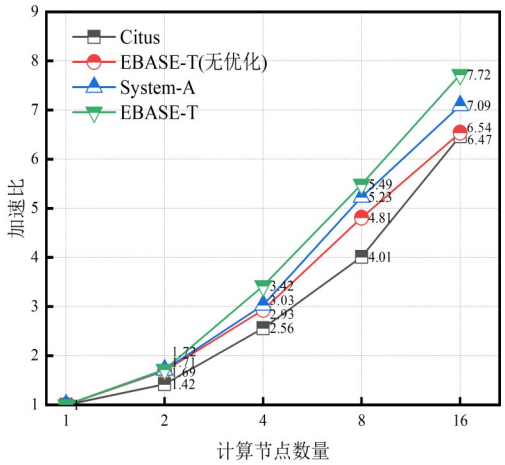


(*c*) 不同共享率下吞吐对比

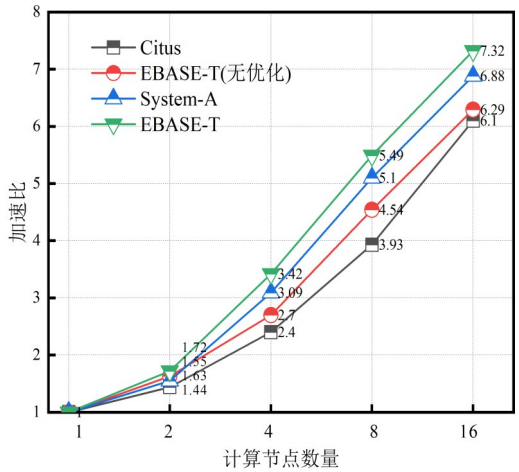


(*d*) 不同共享率下时延对比

图 11　优化机制融合后有效性验证



(*a*) 15% 共享率下的加速比



(*b*) 5% 共享率下的加速比

图 12　数据库系统加速对比实验

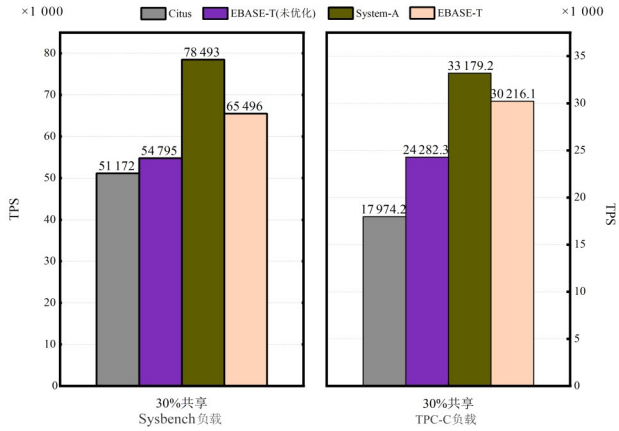


图13 不同负载下的吞吐对比

该节点. 图14展示了此次实验中计算节点1和计算节点2的吞吐量变化情况. 值得注意的是,计算节点1在整个过程中继续为应用提供服务,其吞吐量受到了计算节点2恢复过程较低的影响. 与此同时,节点2在经历约1.5 s后便迅速恢复正常运行. 这种快速恢复得益于EBASE-T的架构设计:节点2能并行进行日志回滚和脏页驱逐,大幅降低了恢复开销,加快了恢复过程.

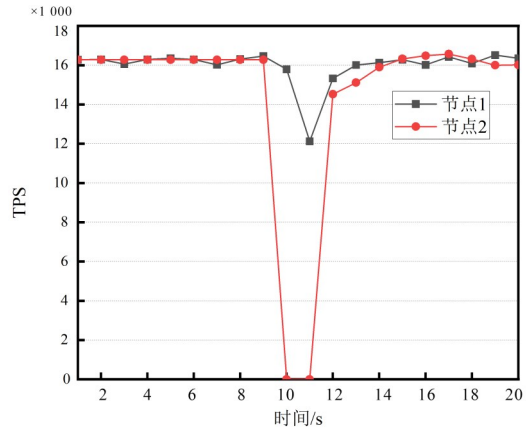


图14 EBASE-T恢复能力测试

这一测试充分体现了EBASE-T的高可用性和高效恢复能力.

6 结论

为了满足海量边缘设备的数据处理需求,当前云端数据库系统普遍采用共享缓存架构. 本文深入研究了基于分布式共享缓存架构的数据库系统所面临的问题与挑战,并探讨了一种高性能共享缓存数据库框架的设计与实现. 与现有的共享缓存数据库系统相比,本文提出了两项关键优化:首先提出了一种基于依赖表的日志延迟写入机制,有效减少了缓存页面转移过程中的WAL日志刷写频次. 其次,设计了异步缓存延迟失效机制,使得缓存页面能够更长时间地服务于不同事务,提高缓存命中率的同时避免了频繁的缓存重新载入. 实验结果表明:这些优化机制使得EBASE-T系统吞吐量提升了19.5%,时延降低了13.1%. 同时,基于这些优化机制的数据库在扩展加速能力上展现出优于工业界经典同类数据库的表现.

展望未来,随着非易失内存技术的不断进步,基于共享缓存架构的数据库系统可以考虑构建基于非易失内存的持久化缓存层,以进一步降低日志落盘开销. 此外,随着以RDMA为代表的高性能网络设备的普及,如何利用这些新型网络设备的特性来优化共享缓存架构中的缓存流转和消息传输,也将是本文后续工作关注的重点.

参考文献

[1] CORBETT J C, DEAN J, EPSTEIN M, et al. Spanner: Google's globally distributed database[J]. ACM Transactions on Computer Systems, 2013, 31(3): 1-22.

[2] YANG Z K, YANG C H, HAN F S, et al. OceanBase: A 707 million tpmC distributed relational database system[J]. Proceedings of the VLDB Endowment, 2022, 15(12): 3385-3397.

[3] LYU Z H, ZHANG H H, XIONG G, et al. Greenplum: A hybrid database for transactional and analytical workloads[C]//Proceedings of the 2021 International Conference on Management of Data. New York: ACM, 2021: 2530-2542.

[4] HUANG D X, LIU Q, CUI Q, et al. TiDB: A raft-based HTAP database[J]. Proceedings of the VLDB Endowment, 2020, 13(12): 3072-3084.

[5] PAZ J R G. Microsoft Azure Cosmos DB Revealed: A Multi-Modal Database Designed for the Cloud[M]. Beach Park: Apress, 2018.

[6] VERBITSKI A, GUPTA A, SAHA D, et al. Amazon aurora: Design considerations for high throughput cloud-native relational databases[C]//Proceedings of the 2017 ACM International Conference on Management of Data. New York: ACM, 2017: 1041-1052.

[7] PANDIS I. The evolution of Amazon redshift[J]. Proceedings of the VLDB Endowment, 2021, 14(12): 3162-3174.

[8] DAGEVILLE B, CRUANES T, ZUKOWSKI M, et al. The snowflake elastic data warehouse[C]//Proceedings of the 2016 International Conference on Management of Data. New York: ACM, 2016: 215-226.

[9] 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. New York: ACM, 2019: 1743-1756.
[10] LAHIRI T, SRIHARI V, CHAN W, et al. Cache fusion: Extending shared-disk clusters with shared caches[C]// Proceedings of the 27th International Conference on Very Large Data Bases. New York: ACM, 2001: 683-686.
[11] BARSHAI V. Delivering Continuity and Extreme Capacity with the IBM DB2 Purescale Feature[M]. New York: IBM, 2012.
[12] CAI Q C, GUO W T, ZHANG H, et al. Efficient distributed memory management with RDMA and caching[J]. Proceedings of the VLDB Endowment, 2018, 11(11): 1604-1617.
[13] KATSARAKIS A, GAVRIELATOS V, SIAVASH KATEBZADEH M R, 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. New York: ACM, 2020: 201-217.
[14] LENOSKI D, LAUDON J, GHARACHORLOO K, et al. The stanford dash multiprocessor[J]. Computer, 1992, 25(3): 63-79.
[15] WANG Q, LU Y, XU E, et al. Concordia: Distributed shared memory with in-network cache coherence[C]//Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST´ 21). San Jose: USENIX, 2021: 277-292.
[16] CAO W, LI F 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). Piscataway: IEEE, 2022: 2859-2872.
[17] RUAN C Y, ZHANG Y Q, BI C, et al. Persistent memory disaggregation for cloud-native relational databases[C]// Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. New York: ACM, 2023: 498-512.
[18] YANG X J, ZHANG Y Q, CHEN H, et al. PolarDB-MP: A multi-primary cloud-native database via disaggregated shared memory[C]//Companion of the 2024 International Conference on Management of Data. New York: ACM, 2024: 295-308.
[19] LI G L, TIAN W G, ZHANG J Y, et al. GaussDB: A cloud-native multi-primary database with compute-memory-storage disaggregation[J]. Proceedings of the VLDB Endowment, 2024, 17(12): 3786-3798.
[20] BOUTIN E, ABRAHAM S. Amazon aurora multi-master: Scaling out database write performance[EB/OL]. (2019-08-13)[2024-09-23]. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Amazon_Aurora_Multi-Master_Scaling_out_database_write_performance_DAT404-R1.pdf.
[21] 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.
[22] LI C, MARKL V, MUKHERJEE N, et al. Distributed architecture of oracle database in-memory[J]. Proceedings of the VLDB Endowment, 2015, 8(12): 1630-1641.
[23] WANG T Z, JOHNSON R. Scalable logging through emerging non-volatile memory[J]. Proceedings of the VLDB Endowment, 2014, 7(10): 865-876.
[24] CUBUKCU U, ERDOGAN O, PATHAK S, et al. Citus: Distributed postgresql for data-intensive applications[C]// Proceedings of the 2021 International Conference on Management of Data. New York: ACM, 2021: 2490-2502.
[25] KOPYTOV A. Sysbench: A system performance benchmark [EB/OL]. [2024-09-23]. http://sysbench sourceforge net/.
[26] LEUTENEGGER S T, DIAS D, LEUTENEGGER S T, et al. A modeling study of the TPC-C benchmark[C]//Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. New York: ACM, 1993: 22-31.

作者简介

尚碧筠　女,2004年5月出生于江苏省南京市. 本科. 主要研究方向为数据库、大数据和机器学习.
E-mail: shangbiyun@126.com

魏　星　男,1994年6月出生于江苏省扬州市. 博士. 主要研究方向为数据库、大数据和机器学习.
E-mail: wei.xing6@zte.com.cn

周士俊 男,1979年7月出生于江苏省扬州市. 本科. 主要研究方向为数据库.
E-mail: zhou.shijun@zte.com.cn

冀文冠 男,1993年9月出生于山西省晋中市. 本科. 主要研究方向为数据库.
E-mail: ji.wenguan@zte.com.cn

董诗琦 女,1997年12月出生于江苏省徐州市. 硕士. 主要研究方向为数据库.
E-mail: ji.wenguan@zte.com.cn

屠要峰 男,1972年12月出生于河南省开封市. 博士,研究员. 主要研究方向为数据库、大数据和机器学习.
E-mail: tu.yaofeng@zte.com.cn

董振江 男,1970年2月出生于河南省开封市. 博士,教授,博士生导师. 主要研究方向为人工智能、大数据、信息安全.
E-mail: dongzhenjiang@njupt.edu.cn