计算机科学
COMPUTER SCIENCE

面向NewSQL数据库数据协同持久化的研究

左顺, 李永坤, 许胤龙

引用本文

左顺, 李永坤, 许胤龙. 面向NewSQL数据库数据协同持久化的研究[J]. 计算机科学, 2025, 52(1): 131-141.

ZUO Shun, LI Yongkun, XU Yinlong. Study on Collaborative Data Persistence in NewSQL Databases [J]. Computer Science, 2025, 52(1): 131-141.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.231200079

面向 NewSQL 数据库数据协同持久化的研究

左　顺[1]　李永坤[2]　许胤龙[2]
1 中国科学技术大学计算机科学与技术学院　合肥 230026
2 安徽省高性能计算重点实验室　合肥 230026
(shunzuo@mail.ustc.edu.cn)

摘　要　现代 NewSQL 数据库为了提供数据的高可用性,通常会为数据提供多个副本,以便在某个副本不可用时,可以从其他的副本中获取数据。而在数据多副本的情况下,又需要考虑副本间的数据一致性问题,即在某一时刻不同客户端读取某个数据时得到的结果应该是相同的,因此引入了事务处理机制。在一个包含多个写操作的交互式事务处理过程中,由于数据存在多个副本,因此每个写入操作需要对所有的主备副本进行写入操作。然而主备副本通常分散在不同的机器上,因此会引入写远端副本的时延,其最终将会增大整个事务的处理时延。针对该问题,提出了数据协同持久化的方案,其主要思想是让客户端在本地缓存事务的写操作日志,在最终提交事务时,客户端首先将事务中的写操作日志进行持久化,并将该日志发送给事务的协调者节点,让协调者进行日志数据的分发处理,从而达到两者协同持久化事务数据的目的。实验结果表明,相较于同步持久化方案,协同持久化方案不仅能降低交互式事务处理的时延,还能提高约 38%左右的系统极限吞吐率。

关键词:分布式数据库;并发控制;数据持久化;数据一致性;高数据竞争负载
中图分类号　TP311

Study on Collaborative Data Persistence in NewSQL Databases

ZUO Shun[1],LI Yongkun[2] and XU Yinlong[2]
1 School of Computer Science and Technology,University of Science and Technology of China,Hefei 230026,China
2 Anhui Province Key Laboratory of High Performance Computing,Hefei 230026,China

Abstract　To ensure high availability of data,modern NewSQL databases often create several copies of data so that it can be accessed from other copies in case one copy is not available. With multiple data copies,it is essential to consider data consistency between them. This means that the results should be the same when different clients read the same data at a particular moment. Therefore,a transaction processing mechanism is introduced. In the interactive transactional process with multiple write operations,each write operation must be performed on both the primary and backup copies of the data,since there are multiple copies. However,the primary and backup replicas are typically located on different machines,resulting in increased latency when writing to remote replicas,which in turn can ultimately lead to an increase in the processing latency of the entire transaction. In this paper,we present a collaborative data persistence scheme where the client caches the transaction write logs locally. When the transaction is finally committed,the client firstly persists the write logs of the transaction and sends the logs to the coordinator node of the transaction to allow the coordinator to distribute the log data,so as to achieve the purpose of the two cooperating in persistence of the transaction data. Experimental results show that in comparison to the synchronous persistence scheme,cooperative persistence scheme can not only reduce the latency of interactive transaction processing,but also improve the system limit throughput rate by roughly 38%.

Keywords　Distributed database,Concurrency control,Data persistence,Data consistency,High-contention workload

1　引言

数据库的发展主要经历了以下几个阶段:文件系统阶段、层次模型和网络模型阶段、关系模型阶段、对象数据库阶段、NoSQL 阶段和 NewSQL 阶段[1-3]。本文主要针对关系数据库、NoSQL 数据库和 NewSQL 数据库展开讨论。

关系模型阶段的数据库采用关系模型,将数据抽象成表来进行管理。由于其将 SQL 作为数据查询语言,因此也被称为 SQL 数据库。相较于层次模型和网络模型阶段的数据库,其能够提供更强的数据独立性和更方便的数据索引方式。

到稿日期:2023-12-12　返修日期:2024-05-22
基金项目:国家自然科学基金面上项目(62172382)
This work was supported by the National Natural Science Foundation of China(62172382).
通信作者:李永坤(ykli@ustc.edu.cn)

然而，其采用关系模型进行数据管理，导致其数据灵活性支持较低，不能适应现代互联网数据格式高速变化的需求；并且SQL数据库本身系统特定的设计也导致其不能较好地适应大数据背景下的数据存储和访问需求。

正是因为SQL数据库本身存在的问题，NoSQL数据库决定放弃关系模型来进行数据管理，而采用键值存储、列存储、图存储、文档存储等方式进行数据管理[4]，用于支持不同的应用场景。另一方面，为了支持数据的高扩展性，其还激进地放弃了事务处理能力，即放弃了对数据高一致性的支持。NoSQL数据库即指非关系型、分布式、不提供ACID的数据库。

然而，越来越多的现代应用既需要数据的高可扩展性，又需要数据的高一致性，这对于金融系统来说尤为重要。NewSQL数据库应运而生。它是一种现代关系型的数据库，旨在为OLTP读写负载提供与NoSQL相同的可扩展性能，同时仍然提供SQL数据库中事务的ACID保证，以确保数据间的强一致性。在现在的大数据背景下，越来越多的公司开始开发和使用NewSQL数据库，如CockroachDB[5]，TiDB[6]和Google Spanner[7]等数据库。

NewSQL数据库在架构上通常可以分为存算一体架构和存算分离架构[8-10]。存算一体架构指节点既参与事务的处理，也参与数据的存储，该架构的优点是在读取数据的同时，可以直接读取本地节点上的数据。而存算分离架构中，节点被划分为计算节点和存储节点，计算节点主要用于进行请求的处理，而存储节点则用于处理数据的实际写入操作，这种架构的优点是便于构建云原生的系统，且资源利用率更高。

由于NewSQL数据库中通常通过数据多备份的方式提供数据的高可用性，因此其不可避免地会存在数据写入时延过大的问题。在存算一体架构中，主备数据通常分散在不同的机器上，在进行数据写入时，主数据写入通常是本地的，而备数据写入通常是远程的。为了维护一致性，通常是等待主备数据都写入完成后，才返回响应给上层应用，从而导致整体写时延包含至少一个远程写操作的时延，造成写时延放大的问题。而在存算分离架构中，数据进行写入时需要发送请求到远端的存储节点，也会造成写时延放大的问题。对于包含有多个写操作的事务来说，上述问题最终将会导致事务处理时延过长的问题。

如何减少写操作时延过长而导致的事务处理时延过长的问题，对提升NewSQL系统的性能具有重要的意义。由于存算分离和存算一体架构中写操作流程的不同，导致一些针对存算一体架构的优化方案并不适合在存算分离架构中应用，例如文献[11]中提到的将2PC协议与共识协议相结合[12-13]、对事务是否跨域进行分类[14]等。尽管目前业界内也存在一些通用方案可以降低写操作的时延，但是这些方案的适用场景较为受限，都只适合某一类特定的事务负载。

本文的主要贡献如下：

1)针对现有解决方案，如事务内并发写方案、客户端缓存写方案和异步写方案，分析其如何解决写时延过长的问题，并总结不同解决方案的适用场景。

2)提出了协同持久化方案，通过让客户端和事务协调者共同承担事务写操作日志的持久化任务，降低了事务内单个写操作的时延，最终达到降低事务处理时延和提高系统极限吞吐率的效果。

3)通过实验验证了现有方案在某些事务负载下表现一般，且相较于典型的同步写方案，协同持久化方案既能够降低事务的处理时延，又能够为系统带来约38%左右的性能提升。

2 背景

NewSQL数据库系统的架构主要分为存算一体架构和存算分离架构。

如图1所示，存算一体架构指服务端中同时存在CPU和持久化存储资源，其还展示了数据的多副本情况。NewSQL数据库中，通常使用不同节点进行数据的多备份来保证数据的可用性，并通过Raft[15]协议或者Paxos[16]协议来保证不同副本间数据的一致性。传统NewSQL数据库系统大多采用该架构，但是该方案存在以下问题[8]：1)可靠性低，若服务节点出现故障，则存储在该服务节点上的数据就无法访问甚至丢失了，必须使用主从同步来实现更高的可靠性，这会造成资源的浪费；2)资源利用率较低，不能按需增加CPU资源或者存储资源，只能通过增加服务节点来进行资源的扩缩容。

从图1中可知，Ap，Bp和Cp分别是对应数据的主副本，当客户端想要写A数据时，其会发送请求给包含Ap的节点，即图中的最左侧节点。当该服务节点收到请求时，其会发送请求给右侧两个节点，请求它们对A进行更新，收到成功的响应后，该节点再写本地的Ap数据，最后返回响应给客户端即可。

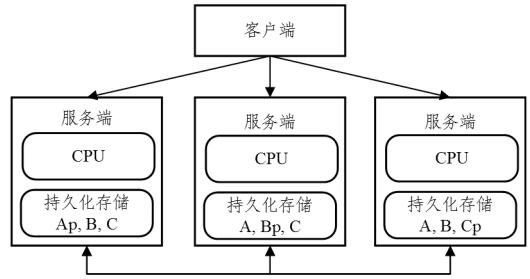

图1 存算一体架构

Fig. 1 Integrated storage and computing architecture

如图2所示，在存算分离架构中，计算节点只需要专注于事务处理，而数据则被存放在存储节点中，供上层服务节点按需使用。存算分离架构具有以下优点[17-18]：1)提高了可靠性，服务节点即使发生宕机也不用担心数据丢失，只要存储池中提供了数据的容灾能力即可；2)资源优化，能够按需增加存储资源或计算资源，保证了资源的高效利用。

从图2可知，数据按照三备份的方式进行存储，客户端想要写A数据时，只需要将请求发送到对应的计算节点即可，计算节点在收到客户端的请求后，会并发向存储节点发送写入请求，以同时写3个备份中的数据，保证数据之间的一致性。

可以发现，上述两种架构都存在写操作处理时延过长的问题，主要原因都在于数据的写入过程包含远端写操作。为行文方便，下文介绍的持久化方案图示都采用存算分离架构作为系统架构进行讲解，但是这些方案的思想在存算一体架构中同样适用。

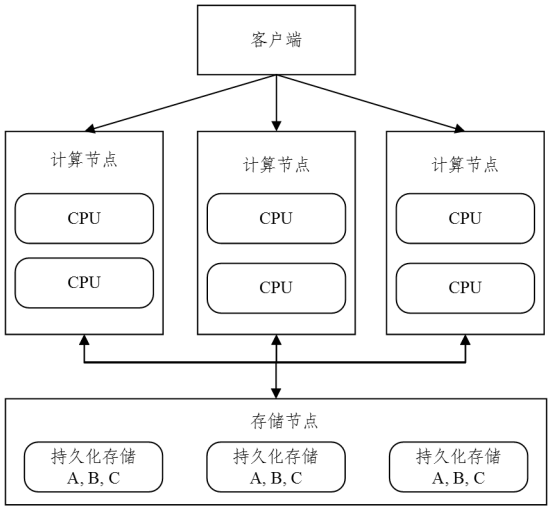

图 2　存算分离架构

Fig. 2　Disaggregated storage and computing architecture

图 3 展示了 NewSQL 数据库系统中的典型事务处理模型,图中的客户端表示实际和服务节点进行交互的节点,通常是支持动态扩缩容和负载均衡的客户端代理或者数据库系统的网关。而持久化存储表示的是远端的存储池,其实现了数据的多备份管理,且所有服务节点都能够访问。协调者节点和参与者节点都是用于真正处理事务中的读写请求的服务节点,但是协调者节点通常还额外保存事务的相关元信息,如事务状态、事务修改的表空间等。协调者节点和参与者节点在物理上都是相同的节点,它们只是在逻辑上进行的细分。具体来说,接收到第一个写操作的服务节点为该事务的协调者节点,而其他服务节点则是参与者节点。

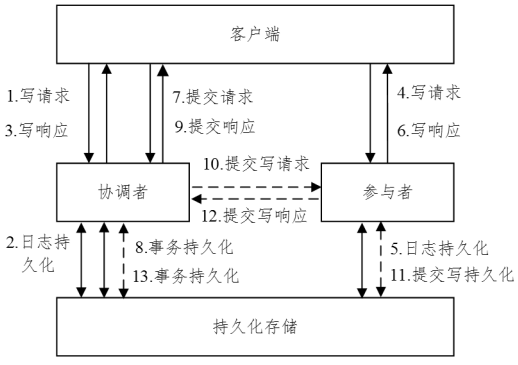

图 3　同步写持久化方案

Fig. 3　Synchronous write persistence scheme

从图 3 中也可以得到只写事务处理的过程,具体如下。

首先,客户端发送第一个写请求到该事务协调者节点,协调者在内部创建该事务记录,保存相关事务信息,如事务状态、包含的写请求等。在创建事务记录完成之后,协调者接着检查内存中的其他事务,判断是否存在读写冲突、写写冲突等。若检测到这样的冲突,则通知客户端中止重试事务,否则进行事务写日志的持久化。持久化完成之后,再返回响应给客户端,告知其可以发送下一条写请求。

客户端在收到对应的写响应后,若发现是中止重试信息,则中止事务,等待一段时间后重试;否则,继续发送下一条写请求给参与者。

参与者在接收到来自客户端的写请求后,同样需要在内存中进行冲突判断,根据是否存在冲突来作出不同的响应给客户端。若没有发现冲突,则在其将持久化操作完成之后,就可以通知客户端继续操作。

客户端在收到来自参与者的请求后,若发现是继续操作响应,则客户端就可以发送提交请求给事务协调者,协调者修改内存中的事务状态并且对该事务状态进行持久化,持久化成功后便可以通知客户端完成事务提交操作。

在后台,协调者根据客户端提交请求中的信息,找到其他的事务参与者,向它们发送提交写请求,用于通知它们将事务的写数据修改为提交状态。在收到来自所有的参与者的提交写响应后,协调者节点就可以在此修改事务状态,并且将其持久化即可。

不难发现,在事务的处理过程中,协调者节点和参与者节点在接收到客户端的写请求后,都需要先进行相应的持久化操作后,才能向客户端返回对应的写响应。由于持久化操作处于事务处理过程中的关键路径上,因此其会造成单个事务的端到端时延过长的问题,下文将该问题简称为持久化延迟问题。

为了更加详细地说明持久化延迟问题,我们可以对事务中的写操作请求进行时间上的分解。具体而言,写操作请求处理时间主要分为 3 个部分,分别是客户端到服务端的网络往返时延 T1、服务端接收到写操作请求后的内部处理时延 T2,以及写操作日志的持久化时延 T3。T1 主要是网络通信造成的时延,约为一个 RTT;T2 主要是服务端中的内存操作,如冲突检测、加锁等操作,该部分时延很低,基本可忽略不计;T3 则包含了服务节点向存储节点的通信时延,约为一个 RTT,同时,存储节点在接收到持久化请求后,还需要写磁盘、SSD 或者 NVM 持久化设备,该部分也存在一部分的时延。持久化延迟问题实际上是由于每个写操作处理时延大约为 T1+T2+T3,造成写操作的时延过大,进而造成只写事务处理时延过大的问题。

3　相关工作

第 2 章中提到了持久化延迟问题,目前,在 NewSQL 数据库系统中,为了消除该问题带来的影响,主要有以下几种方案:事务内并发写方案[19]、客户端本地写方案[20]和异步写持久化方案[21]。下面将针对这些方案进行分析,并总结它们的适用场景。

3.1　事务内写并发方案

由于在事务处理协议中要求每个写操作到达远端服务器后都需要进行持久化操作,因此,一个简单的思路是让客户端并发发送写请求,这样所有的写操作都能并发执行,从而使得写操作的持久化操作在时间上发生重叠。采用该方案的数据库系统有 Chogori-Platform[19]等。

图 4 展示了并发写方案在处理读写事务时的流程,该事务中包含两个 RMW(Read-Modify-Write,读取-修改-写入)操作。从图中可以看到,对于客户端来说,整个事务中的写操作时延相当于只有一个写操作的持久化时延,从而大幅减少了事务的整体处理时延。但是该方案也存在不足:该方案并不

适用于高数据竞争负载的情况，如果发生冲突，将会造成严重的资源浪费，不但并发发送的非冲突写请求会被浪费，这些写操作还可能获取锁资源从而造成事务中止的连锁反应。

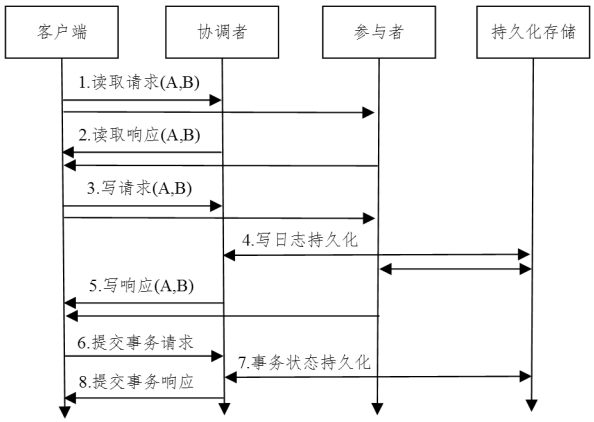

图 4 事务内并发写方案组件交互图

Fig. 4 Component interaction diagram of concurrent write design within a transaction

3.2 客户端本地写方案

另外一种减少持久化延迟的方案称为客户端本地写方案。在该方案中，客户端并不会立即发送写请求，而是在本地缓存写操作，然后在提交事务时将其一起打包发送给协调者，让协调者通过两阶段提交协议进行事务提交。采用该方案的数据库系统有 TiDB[6]，Google Spanner[7]，HBase[20] 等。

图 5 展示了包含有两个 RMW 操作的事务处理过程。

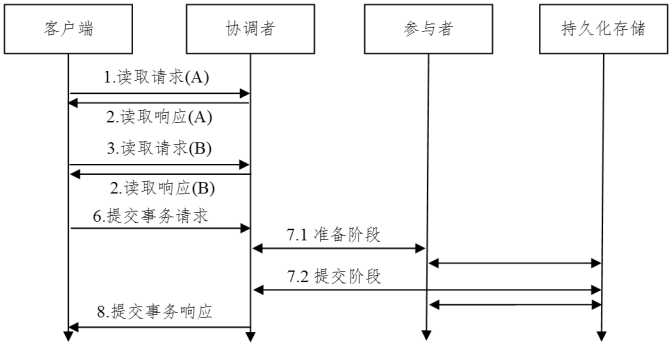

图 5 客户端本地缓存写方案组件交互图

Fig. 5 Component interaction diagram of client local cache write design

从图 5 中可以发现，该方案的主要思路是让客户端延迟发送写操作到对应的服务节点，在最终提交时通过批处理的方式减少对应的请求数目，最终达到降低事务处理时延的目的。

该方案存在以下问题[21]：由于延缓了事务写锁的获取时间(在两阶段提交阶段才获取写锁)，在发生冲突时，可能存在事务内读请求导致的资源浪费现象。另外，从图 5 中也可以看到，客户端发送了写操作请求后并不能立即得到对应的写响应，而是需要等到提交事务时才能知道事务中的写操作能否成功。因此，该方案并不适合交互式的事务处理，后文中将不再讨论该方案。

3.3 异步写持久化方案

前面介绍的两个方案存在一个共同的缺点，那便是它们都不能在高数据竞争负载下良好工作，而异步写持久化方案则刚好能够在高数据竞争负载下工作。采用该方案的数据库系统有 CockRoachDB[21] 等。

图 6 给出了包含 2 个 RMW 操作的事务处理过程，可以发现，该方案主要的思路是让写请求的处理和写请求操作的持久化异步进行。也就是说，远端节点在接收到客户端的写请求后，立即在本地进行相应的处理，如进行冲突检测、锁获取等，然后立即向客户端返回对应的写响应；与此同时，远端节点也在后台异步进行写操作的持久化。

从图 6 中还可以看出，该方案实际上增加了客户端的提交事务时延，因为协调者在真正提交事务前，需要等待参与该事务处理的所有其他参与者都完成对应的写操作持久化任务。该方案的另外一个问题是，相较于同步写持久化方案，其会为系统带来额外的请求响应对，即事务参与者向事务协调者通知其写操作持久化任务已经完成。在相同实验配置下，其极限吞吐率将会低于同步写持久化方案。

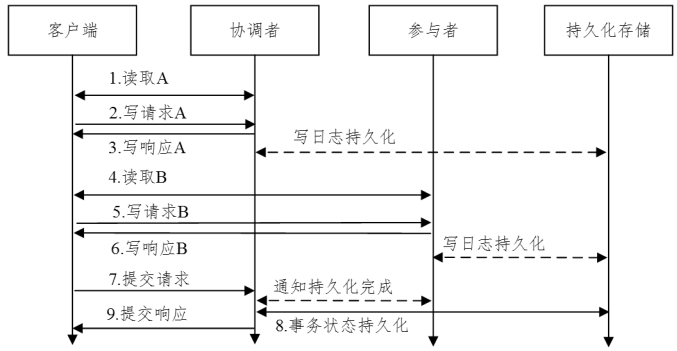

图 6 异步写持久化方案组件交互图

Fig. 6 Component interaction diagram of asynchronous write persistence design

3.4 小结

前文分析了在 NewSQL 数据库中，为了降低持久化延迟，存在哪些可选的方案及其适用的场景。

客户端本地写方案和事务内写并发方案都只适用于低数据竞争负载，在高数据竞争负载下，两者的事务中止率都会相应提高。而在这两种方案中，事务被中止将会带来额外的 CPU 和网络的浪费，前者因为延迟了写锁获取时间，所以可能会导致写操作被浪费；后者则因为并发写操作，所以一旦其中一个写操作与其他事务存在冲突，那么对应的其他写操作都会被浪费。

异步写持久化方案则由于按序发送写请求，能够适用于高数据竞争负载。一旦某个写操作发现冲突，其能够及时中止事务，并且在稍后进行重试，不会造成额外的资源浪费。但是其同样存在对应的问题，协调者在真正提交前需要等待其他参与者发送过来的写操作持久化通知信息，这会为事务带来额外的请求响应对，浪费掉更多的计算资源和网络资源。

综上所述，现有的方案都不能完美解决在分布式数据库中的数据持久化延迟问题，特别是在高数据竞争负载下，异步写持久化方案存在降低极限吞吐率的可能。因此，如何设计一个既能工作在高数据竞争负载下，又能降低单个写操作持久化时延，还能不降低系统的极限吞吐率的方案，成为本文的主要研究问题。

4　协同持久化方案

本章将介绍协同持久化方案，其基本思想是通过让客户端在本地缓存事务内的写操作日志，在提交事务前对其进行持久化，而在实际提交给协调者时附带上本次事务持久化的信息，最终让两者协同完成事务日志的持久化操作，以达到降低事务处理时延的效果。另外，还针对事务恢复和存储空间的回收进行了设计说明。

4.1　事务处理过程

和前文一致，使用存算分离架构来介绍协同持久化方案，但是该方案同样适用于存算一体架构的系统。在该方案的主要思路是让客户端承担部分日志持久化的任务，即远端的服务节点在接收到来自客户端的写请求后，并不会主动进行写日志持久化任务，而是将该任务分发给对应的客户端，从而减轻了远端服务节点的 CPU 和网络负载。另一方面，由于客户端管理事务内所有写操作的持久化任务，因此其可以通过批任务的方式实现持久化请求的压缩，只需要在事务提交请求前进行一次持久化任务即可，从而实现客户端和服务节点协同进行事务数据持久化的过程。

从图 7 中可知协同持久化方案中事务的处理流程。首先，和同步写持久化方案相同，客户端发送第一个写请求到远端服务节点，该远端节点便成为事务的协调者。协调者在内存中进行相应的处理后，如冲突检测等操作，便立即发送写响应给客户端，其中写操作日志就包含在写响应中。客户端接收到写响应后，会将其中的写操作日志在本地进行缓存；接着，客户端发送事务中的其他的写请求给远端节点，此时，非协调者的服务节点都成为了该事务的参与者节点。和协调者节点一样，参与者节点在处理完成后，也将对应的包含有写操作日志的写响应发送给客户端。客户端在接收到写响应后，将其中的写操作日志在本地进行缓存。在客户端提交前，需要先将本地缓存中的写操作打包发送给存储节点，也就是说，将该事务的写操作日志进行持久化。在该持久化操作完成之后，客户端才能真正向事务协调者发送对应的提交请求。为了方便事务恢复过程，提交请求中包含了对应的写操作日志和写操作日志被持久化成功后的地址和大小。最后，事务协调者接收到对应的提交请求后，会将事务的状态进行持久化，持久化完成之后，便立即发送提交响应给客户端，方便发送下一个事务处理的请求。同时，协调者也在后台异步发送提交写请求给其他事务参与者，通知他们修改对应的锁状态，并且让他们同样进行持久化操作，记录下对应的写操作日志。

从上面的事务处理流程中可以看到，协同持久化方案有以下优点：1）相较于同步写持久化方案和异步写持久化方案，该方案能够减少远端服务节点的持久化请求次数，因此可以减轻服务端的负载。在协同持久化方案中，事务的写操作日志和事务状态日志被分离，写操作日志交给客户端完成，而只将事务状态日志交给远端节点完成，从而完成了事务日志的分离，减轻了远端节点的持久化任务负载。2）相较于事务内并发写方案和客户端本地缓存写方案，该方案能够及时发现发生事务冲突的操作，从而快速中止事务，防止造成更严重的资源浪费。在协同持久化方案中，事务内的操作是一个接着一个的，一旦上一个请求和其他的事务存在冲突，下一个请求便必定不会发送出去，不会存在下一个请求同样也发送出去，但是事务最终仍被中止的情况，从而极大地降低了事务因中止重试而产生的资源浪费的现象。另外，在协同持久化方案中，如果发现了冲突，客户端在发送中止事务请求之前也不用进行写操作的持久化，这进一步降低了资源的消耗。3）协同持久化方案也保证了事务的快速恢复策略。在客户端发送提交请求给协调者前，其会先进行写操作日志的持久化操作，持久化操作成功之后会得到对应的地址和大小，用于之后的事务恢复阶段的读取。客户端在随后提交事务前也会将写操作日志持久化成功后的地址和大小一起发送给协调者，用于进行事务状态的持久化。协调者在之后的提交写阶段，会将写操作日志分发给其他的参与者节点，以便它们快速进行事务恢复过程。

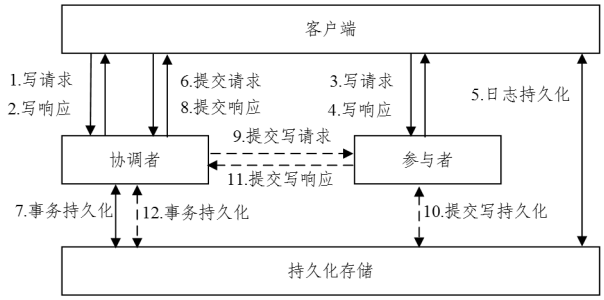

图 7　协同持久化方案

Fig. 7　Collaborative persistence design

上文虽然描述的是协同持久化方案在存算分离下的事务处理过程，但其流程同样适用于存算一体架构。需要特别指出的是，在存算一体架构中，当服务端接收到来自客户端的写入请求时，其并不会对本地的主数据进行写入，而是简单记录一下锁表信息即可，真正完成主备数据写入的时间是在提交写请求时，当接收到提交写请求，服务节点才会完成数据写入并释放锁资源。

4.2　事务恢复流程

4.1 节中谈到了客户端在发送提交事务请求时，需要将写操作日志和该日志持久化后的地址和大小信息一同放在提交事务请求中，以方便远端节点进行事务恢复过程。本节详细描述协同持久化方案中的事务恢复机制，确保在协同持久化方案中，事务恢复过程也能保持较快的速度。

一般来说，需要被持久化的事务数据主要包括两个部分，分别是事务写操作日志和事务状态日志。事务写操作日志保存着事务内所有写操作的数据，而事务状态日志则详细记录了事务的状态。下面分别介绍这两种日志的格式和功能。

事务的写操作日志基本格式为：txnId key1 value1 [key2 value2…]，表示事务标志为 txnId 的事务对 key1 进行了写入，并且写入的值是 value1。如果事务内存在多个写操作，则可以在后面添加更多的键值对来记录事务修改。事务写操作日志在客户端向存储节点发送持久化请求的过程中，一旦存储节点完成了持久化操作，其就会返回对应的地址和大小信息给客户端，方便之后通过该信息再次获取相应的事务写操作日志。存储节点存储数据的方式不尽相同，一般分为两种

基本的存储方式，即原地存储和追加存储。原地存储即将更新后的数据直接在数据原来存在的地方进行更新，如 Redis 数据库[22]等；而追加存储则是将所有修改操作的数据进行追加写保存，如 RocksDB[23]，LevelDB[24]等。下文假设存储节点使用追加存储方式存储数据，数据之间的组织方式使用 Plog 文件进行，其通过块进行连接，每个块大小固定，写端只能在对应的 Plog 节点上进行追加写入，不允许随机写操作，因此，从外部表现上看，Plog 相当于是一个只允许追加写的文件。

事务状态日志的基本格式是 txnId status，其中 status 表示日志状态，具体来说，其具有以下状态：

1）inProgress：表示事务记录为 txnId 的事务正在执行。这条日志用于事务协调者在第一次接收到事务的写操作时，代表客户端开启了一个新的事务，该事务正在处理过程中。实际上，在协同持久化方案中，由于写操作数据持久化操作只发生在提交事务请求之前，因此该状态实际上可以不被持久化，在后面的流程分析中将忽略该状态。

2）committed：表示事务已经被提交，事务协调者在接收到客户端的提交请求时，用于表示该事务已经操作完成，需要记录下事务的状态。注意，在事务状态为 committed 状态下，还需要额外增加事务写操作日志的地址和大小，对于采用 Plog 作为存储节点的实际存储结构，对应的日志信息为 txnId committed plogId offset size，表示该事务已经提交。该事务的写操作日志记录在 plogId 的文件中，并且在该文件 offset 的偏移处存储的就是该事务的日志数据。

3）aborted：和 committed 状态相似，但是其表示事务被中止的状态，也就是说，在事务因冲突而中止时，便会将事务的状态记录为 aborted 状态。

4）finalized：表示事务已经被成功提交且完成了锁释放等操作，用于事务协调者向其他的参与者节点发送提交写请求并得到正确的响应时。

5）commit：这是一个特殊状态，因为其并不表示事务的状态，而是表示事务的写操作状态，其用于表示在该服务节点上，客户端写了哪些数据。该条日志在事务状态修改为 committed 后，记录在该节点修改的数据，如 txnId commit key1 value1。

以上便是整个事务处理过程中涉及的日志数据，接下来将会分析包含有两条写操作的事务 txn：write $a=10$，write $b=20$。

图 8 为不同节点按照时间顺序写入不同日志的流程图。从该图可知，客户端只需要持久化对应的事务写操作日志即可，而事务协调者则需要持久化事务状态日志，事务参与者只需要持久化写提交日志即可，以便其进行快速恢复。

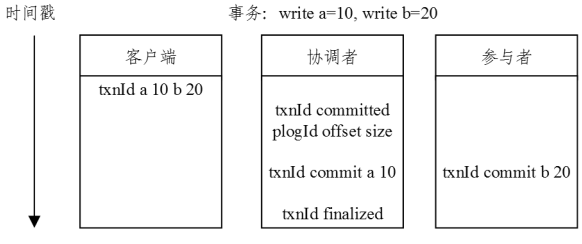

图 8 数据持久化流程图

Fig. 8 Flow chart of data persistence

图 9 详细展示了每种日志是什么时候进行持久化的，以及系统中的各个组件是如何协同工作的。从图中可知，当客户端发送第一条写请求给协调者时，协调者将会在内存中创建对应的事务记录，用于记录下当前的事务状态，随后便返回写响应给客户端。当客户端发送完所有的写请求给远端服务节点，准备进行提交事务请求时，需要先进行事务写操作日志的持久化任务（txnId a 10 b 20）。当协调者节点收到对应的事务提交请求时，若检测到没有冲突，则会对事务状态进行持久化（txnId committed plogId offset size）。之后，协调者将会发送提交写请求给其他事务参与者，事务参与者在接收到对应的请求后，会将事务在本节点进行了哪些写操作进行持久化。图 9 中，对于协调者来说，将会持久化 txnId commit a 10；而对于事务参与者来说，则会持久化 txnId commit b 20。最后，当事务协调者收到所有的参与者节点的提交写响应后，再次持久化事务的状态（txnId fina-lized），表示该事务已经正常提交并且其他参与者节点也已经完成了提交写请求所需要进行的持久化任务。

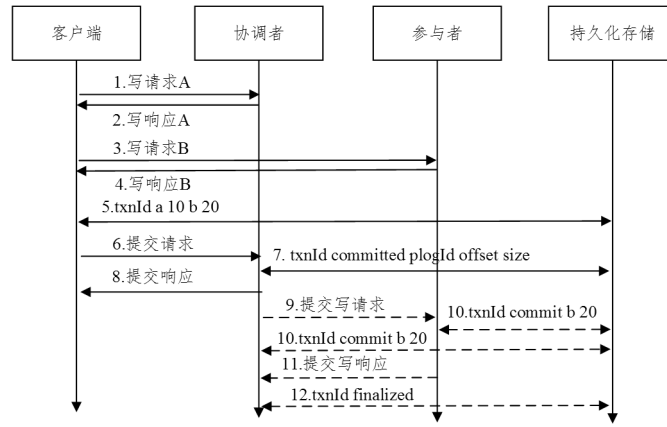

图 9 协同持久化组件交互图

Fig. 9 Component interaction diagram of collaborative persistence design

上文已经详细介绍了事务日志格式和事务数据持久化时机，接下来阐述事务恢复机制的具体过程。对于远端节点来说，其首先顺序访问对应的存储节点（Plog），从中解析出对应的事务状态，然后根据事务的状态进行相应的恢复操作即可。事务参与者和事务协调者的恢复机制存在一些不同，下面将详细介绍两种节点的事务恢复过程。

图 10 展示了事务协调者恢复过程可能遇到的 3 种状态。对于每种状态，恢复过程如下：

1）当在日志中并未发现该事务的任何日志时，此时并不需要进行任何其他的操作，只需要等待该事务发生超时并执行中止操作即可。

2）当在日志中有对应的事务状态处于 committed 状态，则表示该事务被成功提交，此时事务协调者将会根据 plogId offset size 信息到存储节点获取对应的事务写操作日志，根据其中的写操作日志进行重放操作，并且为每个参与者节点发送提交写请求、分发对应的写操作日志。

3）当在日志中有对应的事务状态处于 finalized 状态，则表示该事务被成功提交且清理，此时只需要根据之前的 txnId

commit a 10 信息进行恢复即可。

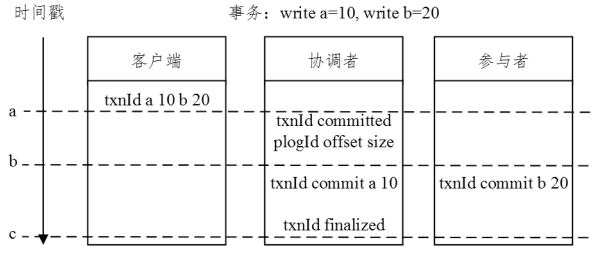

图 10　协调者的日志状态

Fig. 10　Coordinator's log status

下面将介绍事务参与者节点进行日志恢复的过程。

图 11 展示了事务参与者可能遇到的状态。对于每种状态，恢复过程如下：

1）当在日志中发现事务的 commit 日志时，直接对日志进行重放即可。因为发现该日志则意味着协调者已经将事务的 committed 状态记录了下来，最终还是会进行事务的提交流程，因此参与者节点只需要进行写提交操作即可。

2）当在日志中没有发现写事务的 commit 日志，此时情况略微复杂，需要先等待一段时间。如果在等待过程中收到协调者发送过来的提交写请求，则可以直接将其中包含的写操作日志进行重放。等待事件结束后，则可以主动向其他的协调者节点进行询问，查看是否存在尚未完成的提交写任务，并且时戳小于该参与者节点时，开始恢复时间戳的事务，若不存在则可以接收来自客户端的请求，恢复服务，否则需要继续等待和询问。

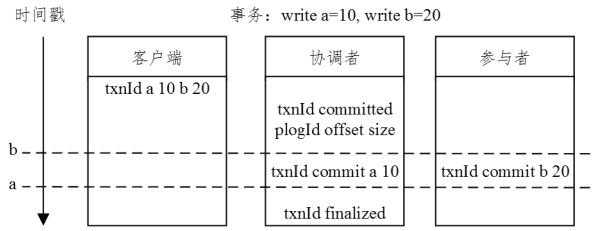

图 11　参与者的日志状态

Fig. 11　Participants' log status

以上便是事务协调者和事务参与者节点的事务恢复过程。总之，服务节点根据自己独占的存储空间上的事务日志，按照规则进行恢复即可。

4.3　客户端存储管理

在协同持久化中，事务的写操作日志实际上被存储了两份，第一份是客户端在事务提交前进行了一次持久化，第二份便是事务协调者节点通过提交写请求向其他的事务参与者节点分发对应的写操作日志，这样便造成了存储冗余。为了降低存储空间上的开销，还需要设计存储空间回收机制。

从图 9 中可知，只要事务状态达到 finalized，那么客户端进行持久化的写操作日志便可以被回收了。为此，可以新增一条回收日志请求，里面包含了 plogId 和 offset 等信息。当协调者将事务的 finalized 状态持久化后，就可以发送回收日志请求给客户端，即通知客户端该存储区域可以开始进行对应的回收操作了。

同时，客户端在本地也应该维护一个哈希链表，用于管理远程持久化存储的使用情况，如图 12 所示。

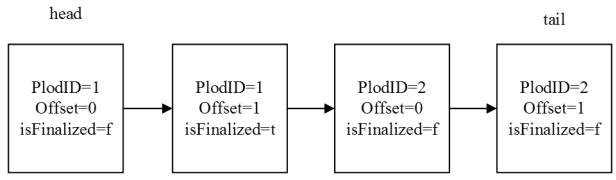

图 12　客户端存储空间管理

Fig. 12　Storage management of client

当客户端收到来自协调者的回收日志请求时，需要进行以下处理：

当回收日志请求中的 plogId 和 offset 不同于 head 指向的 plogId 和 offset 时，只需要修改对应项中的布尔变量 isFinalized 为 true 即可。

当回收日志请求中的 plogId 和 offset 与 head 指向的 plogId 和 offset 相同时，才将 head 按序移动到下一个 isFinalized 为 false 的项。

当 head 在移动过程中发现下一个 plogId 和上一次的 plogId 不同时，说明上一个 plogId 对应的存储块已经被全部回收完毕，可以通知远端存储服务器进行回收操作。

如图 12 所示，当收到的日志回收请求中的 plogId 和 offset 与 head 相同时，head 将会后移。由于下一项中的 isFinalized 为 true，需要再次后移，此时对应的 plogId 为 2，不同于之前 plogId 为 1 的情况。因此，此时客户端可以发送存储空间回收请求给远端的存储服务器，通知其进行回收，从而完成对 plogId 为 1 的回收操作。

4.4　协同持久化方案的局限性

协同持久化方案存在以下局限性：首先，将部分持久化任务分派给客户端进行处理会增加客户端的负载，如果客户端接近满载，将会导致单个事务的处理速度下降，时延增加，因此需要注意客户端的负载情况，及时对客户端进行水平扩展；其次，协同持久化方案本质上还是悲观并发控制，因此其目标事务负载还是那些数据竞争程度较高的事务负载，对于数据竞争程度较低的负载，其性能不如客户端本地写方案；再次，该方案的优化点主要在于对写操作的优化上，因此，该方案对于写密集的负载提升效果更大，而对于读密集的负载提升效果较为一般；最后，由于该方案选择最后才将日志进行持久化，恢复时需要参与者向协调者发送请求获取恢复信息，因此其故障恢复时间更长。

5　实验验证

本章将通过实验测试不同的持久化方案对事务吞吐率的影响，并针对上述方案的优缺点进行数据上的验证和分析。

实验环境如下：使用开源库 Chogori-Platform（乔格里平台）作为实验基座，其原生提供了如图 2 所示的写操作同步持久化方案，文中提出的其他方案，如事务内并发写方案、异步写持久化方案和协同持久化方案均在其上基础上进行修改实现。实验集群包含 6 台物理机器，每个物理机器上具有 2 个 CPU，CPU 型号是 Intel Xeon Gold 5218R，其核数为 20，且使用了超线程技术（40 线程数），运行测试时的频率为 4.000 GHz；物理机器上均配备了 6 块 DDR4 32 GB 2666 MHz

的内存。集群之间使用 RDMA 进行网络通信，RDMA 网卡型号是 Mellanox ConnectX-6 DX Dual Port 100GbE，对应驱动版本为 MLX5，使用的协议为 RoCE v2，连接集群机器的交换机速率为 1Gbps，经测试，集群间小包(16 B)的吞吐率约为 0.88 Gbps。

在进行实验时，默认设置客户端或服务端所占核数为 1。乔格里平台采用的是存算分离架构，服务端由服务节点和存储节点组成，服务节点用于进行事务处理，存储节点则接收服务节点的存储请求，并将数据进行落盘操作。因此，在实验设置中，服务端的服务节点和存储节点应该是跨机器部署的。具体来说，若某个服务端的服务节点分配在第 i 个机器上，那么其存储节点将会被分配在第 $(i+1) \bmod 6$ 机器上，以保证服务节点进行持久化时是通过跨机器网络传输的。另外，将不同的服务节点部署在不同的物理机器上，确保它们之间也是跨机器网络通信的。实际上，由于实验机器全部通过内网进行连接通信，因此本文的实验主要测试在单数据中心下多分区事务的性能。

实验在 YCSB 框架中进行，但是运行的事务负载却与标准的 YCSB 事务负载不同。由于这些方案主要是针对写操作进行优化的，因此在进行实验时运行的事务都是只写事务，并且每个事务中的写操作个数都是 30。事务内写键的 ID 是随机选取的，通过限制事务中每个写操作的键的区间范围来控制事务之间的数据竞争程度，即每个写操作的键 ID 的大小都是从 0 到键空间大小中随机选择一个，也就是说，键空间越小，事务之间发生冲突的概率越大。

5.1 不同持久化方案在高低负载下的系统吞吐率

首先，为了测试不同持久化方案在低负载下的系统吞吐率，采用如下实验设置：6 个服务节点，1～4 个客户端，客户端的并发事务数目默认设置为 1，通过控制客户端的数目来改变系统的整体负载。

图 13 展示了系统在不同客户端数目下的写操作吞吐率对比。从图中可以看出，当系统处于低负载时，即客户端数目在 1～3 时，异步写持久化方案的确能够提升系统的吞吐率，系统负载越低，提升幅度则越大，这和预期一致。异步写的主要思路就是利用服务节点在后台执行持久化任务并且通知事务协调者节点进行信息统计，以便决定在事务提交时是进行等待，还是正常进行提交事务流程。当系统的负载比较低时，上述任务的执行时间将会和其他服务节点处理写操作的时间发生重叠，从而减少了端到端的事务处理时间。具体来说，当第一个写操作发送给协调者时，协调者在内存中处理完成后，就可以发送对应的写响应给客户端，客户端在收到该响应后，就会发送下一个写请求到其他的服务节点。与其同时，协调者也会在后台进行写持久化任务，该部分的时延将不会影响到事务的整体处理时延。总的来说，异步写方案利用了服务节点在后台执行任务的特点，减少了整个事务端到端的处理时延。

从图 13 中还可以看出，并发写持久化方案在客户端数目为 1～3 时，其系统吞吐率略优于协同持久化方案，这是因为此时系统的负载较低，并发写请求在发送到服务端时，能够得到较快的处理，从而减少了事务的端到端时延，最终提高了系统吞吐率。而在客户端数目为 4 时，并发写方案的性能劣于协同持久化方案，这是因为此时服务端负载增大，服务端对大量的写请求进行处理时存在排队现象，导致事务处理时延增加，最终导致吞吐率低于协同持久化方案。

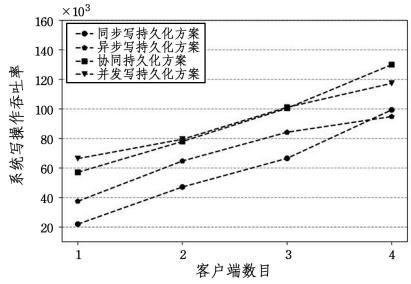

图 13 不同写方案下的低负载测试

Fig. 13 Low load test for different write designs

从图 13 中可以总结出，在低负载且无任何冲突的情况下，并发写持久化方案的系统吞吐率优于其他持久化方案，而协同持久化方案的吞吐率则处于第二的水平，异步写由于减少了端到端事务时延，其吞吐率处于第三的水平，而原来的同步写方案性能最差。

其次，为了测试不同持久化方案在高负载下的吞吐率，采用了如下实验设置：6 个服务端，4 个客户端，同时改变客户端的并发事务数目，以达到改变系统负载的效果。注意，客户端的并发事务数目指客户端同时能够执行事务的最大个数，并发事务的数目越多，系统的事务负载就越严重。

图 14 展示了高负载下的同步写和异步写方案的系统吞吐率。从图 14 可以看出，当系统处于高负载的情况下，异步写持久化方案下的吞吐率表现不如其他持久化方案。该实验结果同样符合之前的分析，即当系统的负载越来越大，异步持久化方案中的后台持久化任务将不会很快得到执行，从而导致事务协调者在接收到来自客户端的提交请求时，需要进行等待，当所有的持久化任务都完成才能进行事务的提交流程，从而导致事务端到端的时延较大，最终致使异步写持久化方案下的吞吐率低于同步写方案的吞吐率。另外，由图 14 还可知，在极限负载下(并发数目大于 8 时)，异步写方案的性能大约只有同步写方案性能的 48%。导致这个结果的主要原因在于，相较于同步写方案，异步写方案每个写操作都额外引入了一个新的远程服务调用的过程，即用于通知事务协调者进行事务写操作状态的追踪过程，该远程过程调用同样消耗了部分 CPU 和网络资源，从而导致异步写的极限性能低于同步写的性能。

从图 14 中还可以看出，在系统处于高负载的情况下，协同持久化方案优于其他任何持久化方案，并且该方案的极限吞吐率相较于同步持久化方案或并发写持久化方案，提升了约 38% 左右。这是因为相较于这两种方案，协同持久化方案减少了系统处理单个事务所需的持久化请求个数，其在客户端缓存写操作的日志数据，只在最终的提交事务请求前才会进行事务日志数据的持久化。也就是说，其通过将同步写方案中的按照每个写操作进行持久化的方式改为按照事务为粒度进行持久化，从而大大降低了写操作时延，减少了事务的整体处理时延。

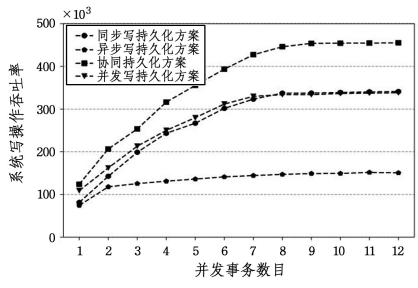

图 14　不同写方案下的高负载测试

Fig. 14　High load test for different write designs

另外，虽然并发写方案相较于同步写方案能够提升吞吐率，但是其并不能提高系统的极限吞吐率，这是因为并发写方案只是简单并发发送了请求，并没有进行其他的优化操作。因此对于系统来说，其极限吞吐率不会增加。

5.2　不同持久化方案在不同数据竞争程度负载下的吞吐率

为了测试不同持久化方案在不同数据竞争程度负载下的吞吐率，所采用的事务负载如下：每个事务中包含 30 个写操作，实验集群包含 6 个服务端，4 个客户端，客户端并发数目为 1，在不同的键空间下进行系统写操作吞吐率的统计。其中，键空间表示的是每个事务随机选择的键的取值空间，事务内的写操作每次在发起时从其中均匀选取某个值作为键的 ID。因此，键空间越小，不同事务间写操作的键的取值相同的概率就越大，事务之间的数据竞争程度也就越大，产生的冲突也越大。

图 15 展示了不同的数据竞争程度下的系统吞吐率。

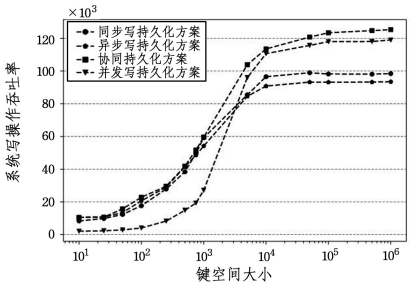

图 15　不同数据竞争程度下的系统吞吐率

Fig. 15　System throughput rate at different levels of data competition

从图 15 中可以看出，在键空间较大时，即事务之间数据竞争程度较轻的情况下，不同方案的事务吞吐率结果和低负载下的测试相一致；而在键空间逐渐减小时，即事务间冲突增大的情况下，不同持久化方案下的吞吐率都有所下降。当键空间下降到 1 000 以下时，所有方案的吞吐率开始大幅下降，且并发写方案的下降幅度最大。这是因为在使用并发写的情况下，一旦事务中某个写操作与其他事务发生了冲突，那么即使该事务中的其他写操作被正常处理，最终客户端还是会将事务中止掉，因为其需要保证事务中的写全部提交或者全部不提交；另一方面，由于最终会中止的写操作占用了锁资源，又会影响新事务的执行流程，导致新的事务一直在等待或者重试，直到原来的事务释放对应的锁资源。可以看到，相较于并发写持久化方案，其他方案在冲突程度较大的负载下能够获得更高的吞吐率，这也验证了并发写持久化方案并不适合在较高负载下工作。

图 16 为不同键空间大小下系统的事务中止率统计图。

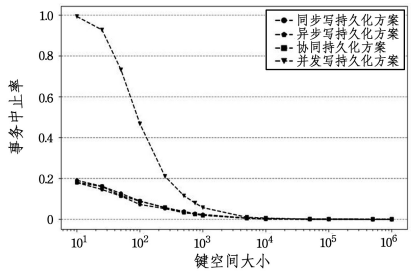

图 16　不同数据竞争程度下的事务中止率

Fig. 16　Transaction abort rate at different levels of data competition

从图 16 中可以看出，在键空间小于 1 000 时，并发写方案下的事务中止率大幅提升，这证实了并发写方案并不适合在高数据竞争负载下工作；同时，实验结果也表明，协同持久化方案和异步写持久化方案能够在高数据竞争负载下良好工作。

5.3　不同持久化方案下的故障恢复时间对比

为了对比不同的持久化方案下的故障恢复时间，采用如下实验设置：3～6 个服务端，4 个客户端，每个客户端并发事务为 1，单个事务内只包含有 30 个写操作。由于当前系统并未实现快照功能，因此选择在系统正常运行 5 s 时，通过随机中止某个服务节点，然后再启动该服务节点，统计该服务节点何时才能通过日志进行恢复，以达到中止前的状态。相较于同步写方案，由于异步写方案和并发写方案并没有修改原有恢复流程，因此只统计同步写方案和协同持久化方案下的故障恢复时间。

图 17 给出了这两种方案的故障恢复时间。可以看到，协同持久化方案的恢复时间比同步写方案长，这是因为同步写方案会在每次写入操作时将日志信息存入持久化存储，这样在服务节点宕机时，就可以直接读取自己独占的远程日志进行恢复；而在协同持久化方案中，考虑参与者恢复流程，由于其需要向其他节点发送消息，查看是否存在已经被 committed 但未被 finalized 的事务，如果存在，那么就需要等待来自这些协调者的提交写请求，并写入日志进行，重建内存索引结构。可以发现，正是由于协同持久化延迟了写日志在各自节点上的持久化时间点，才导致其恢复时间相对较长。

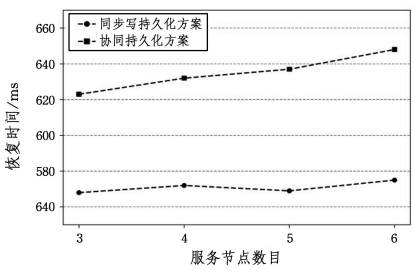

图 17　不同写方案故障恢复时间对比图

Fig. 17　Comparison of fault recovery time for different write designs

从图 17 中还可以看出，随着服务节点数目的增加，协同持久化方案的恢复时间逐步增加，而同步写方案的恢复时间则基本保持不变，这是因为同步写方案恢复时不需要和其他

节点进行交互，只需要读取自己独占的远程日志进行恢复即可；而协同持久化方案的恢复时间受到恢复节点和其他服务节点进行交互的影响，导致其恢复时间增加。

结束语 本文首先针对当前典型事务处理中的同步写方案进行了事务层面的分析，提出了存在于事务处理过程中的持久化延迟问题。针对持久化延迟问题，调研了业界内不同的处理方案，如并发写方案、异步写方案和客户端缓存写方案等，但是这些方案都存在问题。例如，并发写方案和客户端缓存写方案都只在事务之间数据竞争程度较小的情况下有较好的吞吐率，而在事务之间数据竞争程度较大的情况下会造成额外资源的浪费；另外，客户端缓存写并不适合交互式的事务处理，这限制了该方案的应用场景，即其只适用于非交互式事务处理系统；而异步写方案虽然能够在数据竞争程度较大的负载下正常工作，但是其相较于同步写方案引入了额外的请求和响应，因此在极限负载下会降低系统的吞吐率。

针对现有方案存在的问题，本文提出了一种新颖的持久化方案——协同持久化方案，该方案既能够解决持久化延迟导致的单个事务处理时延过大的问题，又能保证系统在极限负载下获得性能提升；另外，在事务之间数据竞争较大的情况下，事务被中止时不会产生额外资源的浪费，保证了其在高数据竞争负载下也能有良好的表现。不过，由于该方案中增加了客户端的网络和存储负载，如果客户端接近满负载，那么经过该客户端的对应事务处理时延将会增大。但实际上，客户端的功能比较单一，可以很方便地在该层实现水平扩展，以减轻协同持久化带给客户端的影响。另外，协同持久化方案适用于处理写密集以及较高冲突的事务负载。

未来的工作主要包含以下几个方面：

1）前文考虑到的方案主要是从现有流程进行优化的，实际上我们可以考虑新硬件直接带来的优化效果，如 RDMA 单边请求进行持久化的方案。目前使用 RDMA 主要有两种方式：第一种是利用 RDMA 提供的内核旁路技术，实现对应的远程服务调用，从而减少对本地和远端 CPU 资源的占用；第二种则是利用其提供的单边读写技术，降低远端服务器的 CPU 负载。RDMA 单边请求进行持久化的方案的大概思路如下：RDMA 单边读写技术可用于上文所提到的持久化方案中，一方面，可以将其应用于持久化操作中，这样在进行持久化操作时，只需要发送节点提供 RDMA 远程写操作即可，这样远程存储节点将不会在处理持久化请求上消耗任何 CPU，大大降低了存储节点的 CPU 负载；另一方面，可以将其应用于客户端读写中，这样也能降低远端服务节点的 CPU 开销，不过该优化可能需要修改服务节点的索引结构，以使其能高效地支持 RDMA 单边读写操作。

2）实际上，在上述客户端协同持久化方案中，需要客户端先在自己独占的存储空间中进行持久化，然后再携带相关持久化信息到协调者中，让协调者节点再次进行持久化，此时持久化的数据产生了冗余。实际上，客户端可以直接将整个事务的写操作日志一起发送给协调者节点，让协调者节点进行持久化即可，并且通过协调者在之后的环节中进行日志分发，这样相较于原来的方案，就减少了一次持久化的次数，可以进一步降低事务的处理时延，减少系统的 CPU 和网络开销。

3）本文的优化主要针对的是交互式的事务处理方案，即客户端在发送读写请求后需要立即得到对应的响应，从而决定下一步该发起什么操作，如继续进行事务操作还是直接中止事务操作。实际上，针对非交互式事务处理，同样也有很多优化措施，如上文提到的客户端缓存写方案，或者业界内的事务预分析处理机制，如 Calvin 系统[25]、FaunaDB 系统[26]等，其通过对需要处理的事务进行预分析处理，然后按照顺序执行，从而最大程度降低事务间的数据竞争。

4）本文的优化主要聚焦在如何降低事务中的写操作时延上，并未针对 NewSQL 数据库的系统架构进行对应的优化。实际上，针对存算一体架构和存算分离架构，在涉及跨域事务处理上，两者存在明显区别。在存算一体架构中，由于主数据和计算资源是在同一节点上的，其写入时间较短，在执行跨域事务时，通常可以先写主数据，然后后台写备数据，最终提交时只要保证所有主备数据都写入即可，从而整体上减少单个事务的执行时延，这也是异步写的主要思路。但是在存算分离架构中，由于数据和计算资源是通过网络连接的，这将导致更大的写入时延，因此，针对存算分离的数据库系统，可以从其他方面对其进行优化，如使用悲观写和乐观读的并发协议，控制事务的 p999 尾延迟在一个可控范围内[27]，或者在协调者提交事务时等待一段时间，以便将事务修改的日志聚集起来，将底层存储的多个小写请求转化为一个大写请求，降低底层存储层的压力，以最大化系统的事务吞吐率[28]。

参 考 文 献

[1] DANIELSEN A. The evolution of data models and approaches to persistence in database systems[J/OL]. https://www.fing.edu.uy/inco/grupos/csi/esp/Cursos/cursos_act/2000/DAP_DisAvDB/documentacion/OO/Evol_DataModels.html.

[2] FATIMA H, WASNIK K. Comparison of SQL, NoSQL and NewSQL databases for internet of things[C]// 2016 IEEE Bombay Section Symposium(IBSS). IEEE, 2016: 1-6.

[3] BINANI S, GUTTI A, UPADHYAY S. SQL vs. NoSQL vs. NewSQL-a comparative study[J]. Database, 2016, 6(1): 1-4.

[4] MONIRUZZAMAN A, HOSSAIN S. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison[J]. arXiv: 1307.0191, 2013.

[5] TAFT R, SHARIF I, MATEI A, et al. Cockroachdb: The resilient geo-distributed sql database[C]// Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020: 1493-1509.

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

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

[8] Hewlett Packard Corporation. The machine: A new kind of computer[EB/OL]. https://www.hpl.hp.com/research/systems-research/themachine/.

[9] Intel Corporation. Intel rack scale design architecture[EB/OL].

https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html,2021.
[10] MARCOS K A,NADAV A,IRINA C,et al. Remote regions:a simple abstraction for remote memory[C]// 2018 USENIX Annual Technical Conference. 2018:775-787.
[11] DU X Y,LI T,LU W,et al. Cross-domain Data Management [J]. Computer Science,2024,51(1):4-12.
[12] YAN X,YANG L,ZHANG H,et al. Carousel:Low-latency transaction processing for globally-distributed data[C]// Proceedings of the 2018 International Conference on Management of Data. 2018:231-243.
[13] MU S,NELSON L,LLOYD W,et al. Consolidating concurrency control and consensus for commits under conflicts[C]// 12th USENIX Symposium on Operating Systems Design and Implementation(OSDI 16). 2016:517-532.
[14] REN K,LI D,ABADI D J. Slog:Serializable,low-latency,geo-replicated transactions[J]. Proceedings of the VLDB Endowment,2019,12(11):1747-1761.
[15] ONGARO D,OUSTERHOUT J. In search of an understandable consensus algorithm[C]// 2014 USENIX Annual Technical Conference(USENIX ATC 14). 2014:305-319.
[16] LAMPORT L. Paxos made simple [J/OL]. https://lamport.azurewebsites.net/pubs/paxos-simple.pdf.
[17] YIZHOU S,YUTONG H,YILUN C,et al. Legoos:A disseminated,distributed OS for hardware resource disaggregation [C]//13th USENIX Symposium on Operating Systems Design and Implementation. 2018:69-87.
[18] PENGFEI Z,JIAZHAO S,LIU Y,et al. One-sided rdma-conscious extendible hashing for disaggregated memory[C]// 2021 USENIX Annual Technical Conference. 2021:15-29.
[19] Futurewei Cloud. Chogori-Platform[EB/OL]. https://github.com/futureweicloud/chogori-platform.
[20] General HBase tuning[EB/OL]. https://www.ibm.com/docs/en/db2-big-sql/5.0.2?topic=performance-general-hbase-tuning.
[21] Cockroachlabs. How Pipelining consensus writes speeds up distributedSQL transactions[EB/OL]. https://www.cockroachlabs.com/blog/transaction-pipelining/.
[22] CARLSON J. Redis in action[M]. Simon and Schuster,2013.
[23] DONG S,KRYCZKA A,JIN Y,et al. Rocksdb:Evolution of development priorities in a key-value store serving large-scale applications[J]. ACM Transactions on Storage(TOS),2021,17(4):1-32.
[24] LevelDB[EB/OL]. https://github.com/google/leveldb.
[25] THOMSON A,DIAMOND T,WENG S C,et al. Calvin:fast distributed transactions for partitioned database systems[C]// Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. 2012:1-12.
[26] FunnaDB[EB/OL]. https://fauna.com/blog/inside-faunas-distributed-transaction-engine-dte.
[27] CHEN Y,YU X,KOUTRIS P,et al. Plor:General transactions with predictable,low tail latency[C]// Proceedings of the 2022 International Conference on Management of Data. 2022:19-33.
[28] ZHOU X,YU X,GRAEFE G,et al. Lotus:scalable multi-partition transactions on single-threaded partitioned databases[J]. Proceedings of the VLDB Endowment,2022,15(11):2939-2952.

ZUO Shun, born in 1999, postgraduate. His main research interests include distributed databases and data persistence.

LI Yongkun, born in 1986, professor, is a member of CCF(No. 33184M). His main research interests include data storage and file systems, especially storage and system problems in application scenarios such as cloud computing and big data, virtualized memory management, key-value systems, flash, NVRAM.

(责任编辑:何杨)