分 类 号 _____ 学号 M201372492

学校代码 10487 密级 _____

华中科技大学

硕士学位论文

Ceph 存储引擎中基于固态盘的日志机制优化

学 位 申 请 人 ： 唐颢

学 科 专 业 ： 计算机科学与技术

指 导 教 师 ： 刘景宁教授

答 辩 日 期 ： 2016 年 5 月 29 日

A Dissertation Submitted in Partial Fulfillment of the Requirements

For the Degree of Master of Engineering

Optimizing the logging mechanism based on SSD in Ceph's Storage Engine

Candidate : Tang Hao

Major : Computer Science and Technology

Supervisor : Prof. Liu Jingning

Huazhong University of Science and Technology

Wuhan, Hubei 430074, P. R. China

May, 2016

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即。学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□，在_____年解密后适用本授权书。

本论文属于

不保密□

（请在以上方框内打“√”）

学位论文作者签名:　　指导教师签名:

日期:　　年　　月　　日　　日期:　　年　　月　　日

摘 要

Ceph 是当前流行的分布式存储系统，具有很好的扩展性和可用性。随着存储技术的发展，SSD（即“Solid State Disk”）等新型存储器件逐渐普及，很多用户通过在 Ceph 集群中使用 SSD 来加速应用的访问。在典型的部署场景下，Ceph 利用 SSD 构建分布式缓存池，或者在存储节点上将 SSD 作为 HDD（即“Hard Disk Drive”）的日志设备来提升系统性能。但 Ceph 的存储引擎用来实现事务接口所采用的日志机制需要更新日志和文件，存在两次写入，限制了 SSD 对性能的提升。

在 SSD 作为日志设备的场景下，针对传统日志机制的两次写入问题而提出的热数据延迟写机制，采用内存索引将日志设备变为热数据的临时存储空间，热数据的更新记录写入日志即可实现热数据的更新，降低了写延迟；采用热数据延迟写的方法，仅当回收日志空间时，再将热数据最近的一次更新写回文件系统，降低了写磁盘的次数；采用双日志结构，分别记录冷热数据的更新操作，其中冷数据的操作记录写入日志后直接写回文件系统，减少了日志空间回收时的读开销。

基于固态盘的 Ceph 存储引擎日志机制的优化不仅保证了事务的原子性，还通过热数据延迟写和冷热分离的双日志结构降低了写磁盘的次数。在 Ceph 典型的部署场景下进行的多项性能对比测试表明，对于 fio 产生的没有明显热点的随机写负载，优化机制仍然能够保证系统的性能；对于 filebench 产生的热点明显的 fileserver 负载，优化机制能够大幅度提升系统性能。

关键词： Ceph　双日志结构　热数据延迟写

Abstract

Ceph is a popular distributed storage system with excellent scalability and availability. With the development of materials technology, new storage devices, such as SSDs, become more and more popular. Many users of Ceph Cluster use the SSD to accelerate application access. In a typical deployment scenario, Ceph construct distributed cache pool by SSD, or use SSD as the log device of HDD on the cluster node to improve system performance. But Ceph's storage engine use the logging mechanism which need to update log and file on a write to implement transaction interface, causing "double write" problem and limiting the performance improvement by SSD.

The hot data write delay mechanism which proposed to solve the "double write" problem of traditional logging mechanism when using SSD as a log device, equipment log into temporary storage space of hot data through memory index, thus recording the updates of hot data in log achieves the update of hot data ; when recycling log space, the last valid update of hot data will be written back into file system, achieving postponed writes and I/O merged of hot data, thus reduce number of writes to disk; using double log structure to respectively record the update operation of hot data and cold data, and cold data is write directly to the file system as soon as appending to the log, reducing the reading overhead when recycling log space.

The optimization of logging mechanism not only achieve atomicity of transactions, but also reduces the number of writes to disk by using write delay and double log structure. A number of performance tests show that the optimization can significantly improve system performance under fileserver load because of its significant hot data and can also provide the performance as original system under random write load because of its less hot data.

Key words: Ceph double log structure hot data delay write

目 录

1 绪论

1.1 引言

Ceph 是目前非常流行的统一分布式存储系统，对外提供对象接口、块接口以及 Posix 文件接口，它不仅具有高扩展性和可用性，还能实现分布式分层存储等高级功能。

Ceph 的底层是一个分布式对象存储系统 RADOS，即“A reliable, autonomous, distributed object storage”，RADOS 由集群监控器和存储节点构成，其中监控器负责维护集群的全局信息，存储结点负责存储对象及其属性。为了实现高扩展性，RADOS 使用伪随机算法 Crush 来计算对象到存储节点的映射关系，省去了由监控器维护的映射表；同时，RADOS 将传统分布式存储系统中由中心节点实现的功能，如存储节点的故障检测、数据迁移和一致性维护等，都交由存储节点来完成，避免了监控器成为集群的性能瓶颈。为了实现集群的高可用性，RADOS 使用副本机制实现数据的冗余存储，同时使用多个服务器构成的小集群实现高可用的集群监控器，避免了单点故障。

目前 Ceph 有很多的用户案例，在 2013 年，Inktank 公司在邮件列表中做了相关的调查，共收到了 81 份有效反馈，其中有 26%的用户将 Ceph 用于生产环境中，有 37%的用户将 Ceph 用于私有云中，还有 16%的用户将 Ceph 用于公有云中。近日，开源软件提供商 Suse 正式发布的 SUSE Enterprise Storage，也是基于 Ceph 的商业支持版本开发的。

但由于复杂的 I/O 协议栈，性能成为 Ceph 的软肋。用户的 I/O 请求通过网络到达存储节点后，还要通过多个队列进行请求的合法性检查和处理，然后再交由底层的存储引擎执行，漫长的 I/O 路径降低了整个系统的性能。同时，底层存储引擎为了实现事务原子性采用了日志机制，在处理写事务时需要同时更新日志和目标文件，

带来了两次写入，使得系统的性能雪上加霜。因此降低日志机制对性能的影响成为提升系统性能的重要方案之一。

随意存储技术的发展，SSD 等新型存储器件的逐渐普及，从成本和性能上的均衡考虑出发，很多服务器都配备多块 HDD 和一两块 SSD，在 Ceph 典型的部署场景下，SSD 用来存储日志，而 HDD 用来存储对象及其属性，由于更新操作必须先更新日志，这种部署方式既能够加速用户的访问，又不会花费企业太大的成本。此外，还可以将多台服务器上的 SSD 构建成分布式存储池，将其作为 HDD 存储池的前端，从而构成类似 flashcache 架构的分布式分层存储系统。但日志机制带来了两次写入，限制了 SSD 对系统性能的提升。因此，在 SSD 作为日志设备的场景下，针对日志机制带来的两次写入问题，提出了热数据延迟写机制，大大降低了日志机制对性能的影响。

1.2 研究现状

为了保证数据的一致性，存储系统采用了很多技术，使得系统能够在故障重启时恢复数据的一致性。主要方法包括日志[1-2]、写时拷贝[3]和软更新[4-5]等，其中日志机制因为简单高效而被很多存储系统采用。

Ceph[6-11]的存储节点使用文件系统存储对象，使用数据库存储对象属性，并借助日志机制实现对象及其属性的原子更新，供上层逻辑实现其他高级功能；文件系统 ext3 使用块设备日志 JBD(即“journaling block device”)提供的日志服务实现文件更新的原子性，确保了数据一致性。然而日志机制造成了两次写入，即一次写日志，一次写文件，降低了存储系统的性能，为此 ext3 提供多种日志模式，分别有 full mode，ordered mode 和 writeback mode 等，在 full mode 下日志记录了元数据和数据的更新操作，因而能够确保整个系统的数据一致性，但牺牲了性能；在 ordered 和 writeback 模式下只将元数据写入日志，大大提升了性能，但只能确保元数据的一致性，其中在 ordered 模式下通过控制数据的写入顺序，能够确保非覆盖写时数据的一致性。但这些优化措施必须在用户对数据一致性要求不高时才能使用。

文献[12]将日志技术和写时拷贝技术结合提出了一种原子更新多个文件的方法，从而为应用层提供了原子更新多个文件的事务接口: 通过使用 AdvFS 提供的 clone 接口实现单个文件的原子性更新，在此基础上借助元数据日志实现多文件的原子更新。由于克隆文件只需操作文件的 inode，而且元数据日志带来的开销也很小，因此该方案能高效地实现原子更新操作。但它要求底层的存储系统对数据支持写时拷贝，适用范围比较局限。

文献[13]为了提升分布式文件系统的性能，将日志设备放于客户端，避免每次更新都经过网络。该机制大大降低了存储网络的流量，提升了系统的性能。但由于客户端缺乏数据的冗余备份，日志设备故障会带来数据的丢失。

随着材料技术和存储技术的发展，一系列新型存储器不断进入人们的视野。闪存和非易失性内存因为良好的性能而广受关注，很多存储系统通过引入这些新型存储器件来加速应用的访问[14-19]。因此，如何利用新型存储器件来加快一致性过程成了研究热点。

文献[20]使用 NVMRAM 来存储文件系统元数据，并划分一部分 NVMRAM 区域来存储元数据日志，在此基础上实现了快速的文件系统在线检查，保证了数据一致性。文献[21]提出了 UBJ（即“Union of Buffer and Journal”）机制，在该机制下系统使用 PCM（即“Phase Change Memory”）等非易失性内存作为文件系统的页面缓冲（page cache），并把缓冲页交给 JBD 管理，由 JBD 将缓冲页构成逻辑上的日志。UBJ 机制对文件系统的页面缓冲和日志功能进行了整合，减少了事务提交时的数据拷贝。FusionIO[22]在固态盘上实现了原子更新文件的接口。MySQL 数据库借助该原子更新接口避免了应用层日志带来的两次写入问题，从而大大的提升了性能[23]。基于该接口实现的 kv 存储系统在确保数据持久化的同时也获得了性能上的提升[24]。但该接口只能实现单个文件的原子更新，并且需要特殊硬件的支持。

除了将新型存储器应用于现有文件系统外，还有些研究专门针对新型存储器设计新的文件系统，从而实现更加高效的数据一致性机制。文献[25]针对 PCM 字节寻址的特性，设计了文件系统 BPFS 和相应的硬件架构。BPFS 借鉴写时拷贝技术，

并结合 PCM 就地更新和字节寻址的特性，提出了短路径影子分页技术，从而提供细粒度的原子更新接口。

在 SAN 网络或者分布式存储系统中，新型存储器还被在客户端作为缓存，从而降低网络流量，提升系统性能。如何在该场景下实现数据的一致性和持久性也是研究的热点。为了确保数据的一致性并简化系统的设计，缓存往往采用 writethrough 策略[26]。由于每次更新都要写入后端存储系统，writethrough 策略无法满足写负载为主的用户的性能需求。文献[27]针对服务器端的 SSD 缓存提出了 journaled writeback 策略，从而使系统能够在一致性和性能之间做权衡。在该策略下，写操作以日志的方式记录在 SSD 中，并被推迟写入后端存储系统，因此对数据的多次写操作会被合并为对后端系统的一次写操作，因而提升了缓存的性能。为了确保数据的一致性，它要求后端系统提供事务接口，从而将一段时间内缓存积累的更新操作原子写回后端存储系统。文献[28]则直接将客户端缓存组织成一个分布式缓存系统，通过副本机制确保数据的可靠性，避免了 writethrough 策略导致的性能问题，但是这种设计过于复杂，不亚于实现一个分布式存储系统。

然而这些借助新型存储器件对一致性过程进行优化的机制都存在这样那样的使用限制。非易失性内存的价格一直高居不下，使得使用非易失性内存优化一致性过程的机制的实用性降低，尤其是在 Ceph 这样的分布式存储系统中，大规模部署非易失性内存是一个不菲的开销。FusionIO 提供的原子更新接口只能原子更新单个文件；日志式的客户端缓存则依赖后端存储系统提供的事务接口才能确保数据的一致性，不具有通用性；分布式缓存能够实现数据的高可靠，到在设计上过于复杂，开发维护代价大。

因此在 Ceph 存储节点使用 SSD 作为日志设备的场景下，针对日志机制的两次写入问题，提出了热数据延迟写机制。该机制将日志作为热数据的临时存储空间，利用固态盘的空间优势推迟文件系统中热数据的更新，进而降低了写磁盘的次数，提升系统的性能。它是日志 full 模式的另一种形式，不会降低数据的一致性。相比非易失性内存，SSD 在生产环境中得到了广泛使用，因而基于 SSD 的热数据延迟写机制更有实用性；热数据延迟写机制不依赖特殊的硬件支持，相比 BPFS 和 FusionIO 更加具有通用性；同时，热数据延迟写机制是针对存储节点的优化，不用担心数据冗余存储问题，因而更加简单可行。

1.3 主要工作和论文组织结构

主要围绕日志机制的两次写入问题进行优化，提出了热数据延迟写机制，并对该机制进行了实验验证。主要工作如下。

(1) 介绍了分布式存储系统 Ceph 的架构特点，阐述了 Ceph 底层存储引擎提供事务接口的必要性。

(2) 采用热度计算公式和排序树实现了对象热度值的实时排名，并据此实现对象的冷热识别。

(3) 采用内存索引将日志变为热数据的临时存储空间，在此基础上实现了热数据的延迟写。

(4) 采用双日志结构分别记录冷热数据的更新历史，提高空间利用率，并减轻了空间回收时的读开销。

(5) 结合 btrfs 的快照功能和日志，实现了双日志结构下的故障恢复。

(6) 对 Ceph 存储引擎日志机制的优化措施进行性能对比测试。

全文共分为如下 6 章：

第一章介绍研究背景和相关的国内外研究现状。

第二章分析了分布式存储系统 Ceph 的架构，解释了其存储引擎提供事务接口的必要性；接着介绍了日志机制的原理和研究现状；最后就日志机制的性能问题以及当前优化措施的短处提出了热数据延迟写机制。

第三章阐述了固态盘作为日志设备下，Ceph 存储引擎日志机制优化的工程实现。

第四章针对 SSD 在分布式存储系统 Ceph 中的两个典型场景下，对热数据延迟写机制进行了多项对比测试，验证了该机制的可行性。

第五章对本课题进行了总结，并分析了其中的不足之处和可能的改进方法。

2 相关研究与关键技术

Ceph 是当前流行的分布式统一存储系统，具有良好的可靠性和可用性，它的自动故障恢复机制、快照以及分布式分层存储等功能都依赖于底层存储引擎提供的事务接口。日志是实现事务原子性的重要机制之一，并因为实现简洁、效果良好而被广泛采用。Ceph 底层的存储引擎便是使用日志机制实现事务接口的。但在日志机制下，更新数据前必须先更新日志文件，即存在两次写入问题，严重影响了 Ceph 的性能。

随着存储技术的发展，新型存储器件逐渐普及，涌现了很多利用新型存储器件优化存储系统的技术。这些优化措施并不完全适用于 Ceph 环境，因而根据 Ceph 的架构特点，针对 SSD 为日志设备的场景下提出了热数据延迟写机制来优化日志机制的性能。

2.1 Ceph 的系统架构

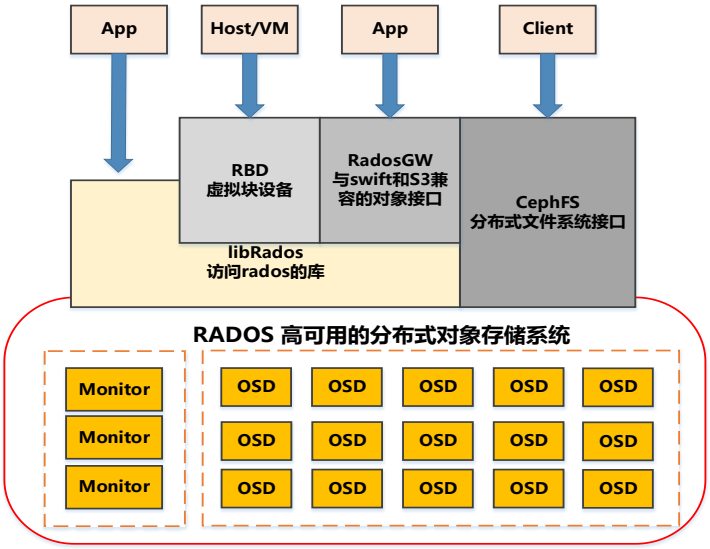

图 2.1 Ceph 总体架构图

Ceph 是一个高可用的分布式存储系统，对外提供多种类型的接口。如图 2.1 所示，Ceph 的底层是 RADOS，即“A reliable, autonomous, distributed object storage”，

应用程序可以通过库 librados 提供的对象接口对 RADOS 进行访问。RADOS 由两部分组成。负责存储数据的对象存储设备 OSD，和维护 Ceph 集群状态的 Monitor，Monitor 自身也是一个高可用小集群，能够避免单点故障。

2.1.1 数据的分布算法

在 RADOS 中，数据以 PG（placement group）为单元在 OSD 间进行备份，PG 到 OSD 的映射关系可以通过伪随机算法 Crush 确定，该算法根据集群中 OSD 的状态和层次结构，将 PG 映射到一组 OSD 上，其中第一个 OSD 上的 PG 为 primary，其他的皆为 replica。Primary 负责接收用户的写请求并在编号排序之后转发给其他副本执行，同时也负责响应用户的读请求；当 OSD 发生故障的时候，primary 还负责维护副本的数据一致性。每个 PG 包含多个对象，对象到 PG 的映射则通过哈希的方式来确定，确保对象在 PG 间均匀分布。

引入 PG 是为了减少 Monitor 维护的元数据信息，避免 Monitor 成为性能瓶颈；同时，OSD 故障处理的任务也被下放到 PG，由 PG 自己收集副本的信息并自动恢复数据，进一步减轻了中心节点的工作量。

2.1.2 故障检测与恢复的机制

在 Ceph 中，Monitor 负责维护全局性的数据，其中一个很重要的结构是 OSDMap，描述了集群中 OSD 的层次结构(每个 OSD 设备所属的机架，机房等)和状态信息(例如 in，down 和 out)。一般来说为了维护一个实时的 OSDMap，Monitor 需要与 OSD 维护一个心跳，但如果集群规模变大，Monitor 就会成为集群扩展的瓶颈。为了避免 Monitor 成为集群扩展的瓶颈，Ceph 通过 OSD 之间的心跳来互相监督，一旦某个 OSD 检测到故障的 OSD 就会报告给 Monitor，由 Monitor 来更新 OSDMap。Monitor 再将 OSDMap 推送给部分 OSD，由 OSD 将其广播到整个集群。

当某个 OSD 收到新的 OSDMap 时，说明集群发生了变动，由于 crush 算法的输入参数包含了 OSDMap，因此部分 PG 到 OSD 的映射也随之改变，这部分受影

响的 PG 会收集副本的 log 信息，并根据 log 确定每个对象的版本号，然后自发地进行数据迁移和同步。

PG 的 log 记录了最近一段时间内的操作序列，这是 OSD 故障处理机制进行数据恢复的凭据。如图 2.2 所示，当 PG 收到来自用户的事务时就需要构建相应日志项追加到 log 尾部，用户事务和该 log 追加操作被合并成一个事务然后交给底层的存储引擎执行，事务的原子性保证了用户的更新操作和 log 中日志项的一一对应。这就要求底层的存储引擎能够提供事务接口，以保证故障处理机制的正确运行。

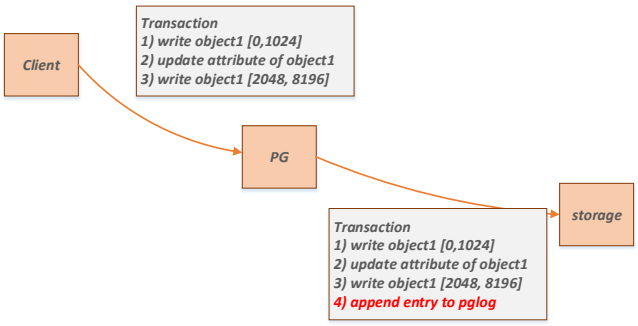

图 2.2 事务的执行

2.1.3 对象的数据和属性

在 OSD 中，对象包括 3 个部分，分别为对象数据，对象的 xattr 属性，以及对象的 omap。其中对象的 xattr 和 omap 有些类似，都是一组 key—value 对，其中 omap 支持范围删除和只读迭代器，用户可以使用迭代器遍历对象的 omap 空间。

用户可以使用 Ceph 对外提供的接口添加、修改或删除对象的 xattr 或者 omap，OSD 本身也利用 xattr 和 omap 来管理对象，例如放置组的 log 就以该放置组下 meta 对象的 omap 存在，利用范围删除接口可以方便地删除过时的日志记录；对象的一些管理信息，例如对象的 object_info_t 结构体（主要记录了对象的大小，状态标示，版本号，修改日期以及快照等信息）以及 snapset 结构体（记录了对象快照和克隆信息），都存储在对象的 xattr 中，借助这些信息 Ceph 才能实现对象锁、写时拷贝快照以及分布式分层存储等功能。OSD 在处理用户发来的更新请求时，需要更新

对象的管理信息和放置组的 log，这些操作构成一个事务并交由底层的存储引擎进行处理。

2.1.4 存储引擎的结构

FileStore 作为 OSD 底层的存储引擎，封装了 OSD 底层的所有 I/O 操作接口。在 FileStore 中，对象的数据存放在相应的文件中，而属性则存放着文件的 xattr 属性中(称为对象的 xattr)，或存放在 leveldb 数据库中(称为对象的 omap)。FileStore 对外提供事务接口，每个事务包含多个更新对象数据或属性的子操作，而为了实现事务的原子性，FileStore 采用了应用层日志。

日志是实现事务原子性的一种很常用的手段，如图 2.3 所示，FileStore 先将事务以 O_DIRECT 和 O_DSYNC 的方式追加到日志尾部，然后再更新目标文件；更新目标文件时，只需将数据写入文件系统的 Page Cache 即可返回；当日志空间耗尽需要回收空间或 Page Cache 中的脏数据超过阈值时，再将 Page Cache 中的脏数据同步到磁盘。因为在更新目标文件前，事务已经在日志中持久化，所以对于非介质损坏类型的临时性故障，都可以通过回放日志中事务的方式来恢复数据。

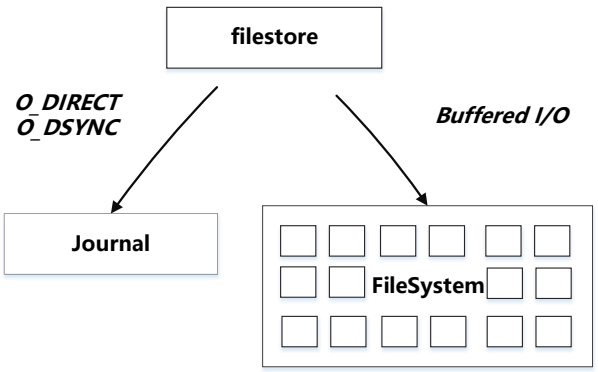

图 2.3 FileStore 结构图

2.1.5 客户端缓存的结构

Ceph 的客户端往往使用内存缓存最近一段时间内读写的数据，避免每次读写都经过网络，从而提升系统的性能。下面简要介绍 Ceph 提供的块设备接口，以及

对应场景下的客户端缓存。

用户可以借助 librados 库提供的对象接口读写存储在 Ceph 中的对象及其属性，而 librbd 库是对 librados 的进一步封装，负责提供块设备接口，它根据用户指定的条带化规则将块设备划分为多个固定大小对象，并将对块设备的读写请求转化为对对象的读写请求。如图 2.4 所示，虚拟化平台 qemu 为虚拟机提供的基于 Ceph 的块设备便是通过 librbd 库实现的。

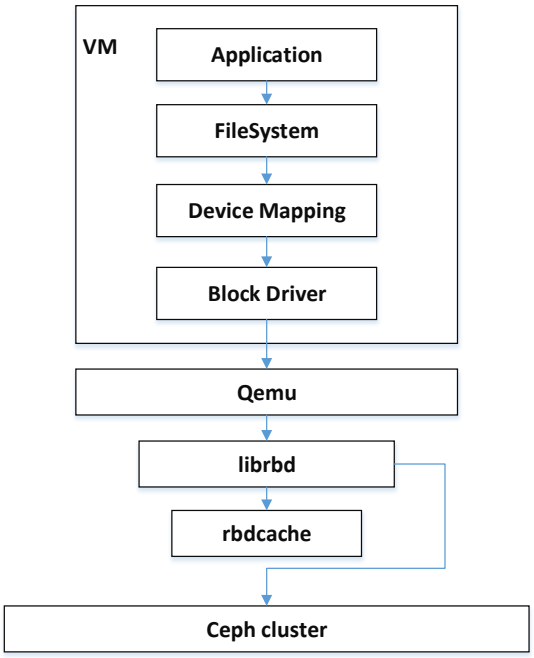

图 2.4 Ceph 的 RBD 客户端

rbdcache 是 librbd 在客户端提供的块设备缓存，它提供读缓存，以及汇聚合并写的功能，从而提升顺序读写的性能。rbdcache 以内存的形式存在，并按照一定的策略将脏数据写回底层的 Ceph。由于是内存缓存，断电或者内核奔溃都会导致数据丢失，为了确保数据的安全，librbd 为 qemu 提供 flush 接口，当 librbd 收到来自 qemu 的 flush 命令时就会将脏数据写回 Ceph。

虚拟机通常会在 librbd 提供的块设备上建立文件系统。众所周知，linux 内核的存储体系提供了两种主要的缓存，即 page cache 和 buffer cache，前者是 linux 为

文件系统提供的缓存，而后者是为块设备提供的缓存。通常情况下应用是使用文件系统存储数据的，所以 buffer cache 一般情况下是指向 page cache 的引用，只有在应用直接访问块设备的时候，buffer cache 才独立缓存数据。这些 cache 中的脏数据通过内核中专门的后台线程周期性地写回到块设备，同时，应用可以通过 fsync 等接口强制 cache 中某个文件的数据写回。

综上所述，在使用 Ceph 提供的块设备为虚拟机提供存储的场景下，虚拟机中 linux 内核提供的 page cache 和 buffer cache，以及 librbd 库提供 rbdcache 构成了 Ceph 的客户端缓存。

2.2 日志技术

2.2.1 数据一致性

数据一致性是指关联数据之间逻辑关系的正确性和完整性。由于存储系统运行过程中难免出现意外的宕机事故，例如内核崩溃、断电或者人为错误导致的意外宕机，而宕机会导致存储系统的操作被打断，关联数据之间的逻辑关系被破坏，因而就会出现数据不一致的现象。

缓存的易失性是导致数据一致性被破坏的原因之一。为了弥补磁盘读写速度慢的不足，存储系统通常会使用内存对磁盘上的数据进行缓存，例如 linux 为文件系统提供的页缓冲（page cache），为块设备提供的块缓冲（buffer cache）等。当意外宕机事故发生时，缓存中的脏数据因为来不及写回磁盘而丢失，使得磁盘上的与之相关联的数据出现不一致。

无法确保事务的原子性则是数据一致性被破坏的更深层次的原因。例如在文件系统中，创建一个文件需要修改文件系统的超级块、inode、目录、索引以及位图等磁盘块，这些数据块的更新是一个完整的事务，应该以原子的方式写入磁盘。如果宕机导致事务中的部分数据块没有更新，就会在逻辑上破坏数据的一致性。同理，在分布式存储系统 Ceph 之中，对象的更新需要修改对象的数据、属性以及相应的

log，如果无法保证更新操作的原子性，那么也会造成数据的不一致。因而实现事务的原子性是实现数据一致性的关键。

2.2.2 日志机制的实现

实现事务的原子性有很多技术，例如日志、写时拷贝和软更新等，其中日志技术因为实现简洁、效果良好被很多存储系统使用。块设备日志 JBD [29, 30]便是日志技术的典范，它的功能是为块设备提供日志支持服务。广为使用的文件系统 ext3 便是借助 JBD 提供的接口实现事务原子性的。这里通过介绍 JBD 机制来讲述传统日志的实现技术。

在 JBD 中，原子操作被称为句柄，例如一个创建文件的操作便是一个句柄。为了提升文件系统的性能，JBD 将多个句柄作为一个事务，并以事务作为基本管理单元。如图 2.5 所示，在一个运行的系统中，共有三种类型的事务，即运行事务，提交事务以及检查点事务。其中运行事务只有一个，表示正在接受句柄的事务，一旦句柄数量达到阈值，便会转化为提交事务；提交事务也只有一个，表示正在写入日志的事务，一旦写入完成提交事务就会变为检查点事务；检查点事务有多个，表示正等待写入文件系统的事务。由于在写入文件系统前，事务已经在日志上持久化，因而在宕机重启时能够恢复被中断的操作，保证了事务的原子性。

为了满足不同用户的性能和一致性需求，JBD 提供了三种日志模式，即 writeback 模式，ordered 模式，以及 full 模式。在 full 模式下，日志记录了元数据和数据的更新操作，因而能够实现数据的一致性。在 writeback 模式下，日志只记录了元数据的更新操作，避免了 full 模式下数据写两次带来的开销，提升了系统的性能，但由于元数据可能指向无效数据块，因而该模式只实现了元数据的一致性。在 ordered 模式下，日志也只记录了元数据的更新操作，因而能够实现元数据的一致性，但它严格控制数据的写入顺序，即写日志前先确保数据写入文件系统，从而在对文件执行 append 操作时能够确保数据的一致性。

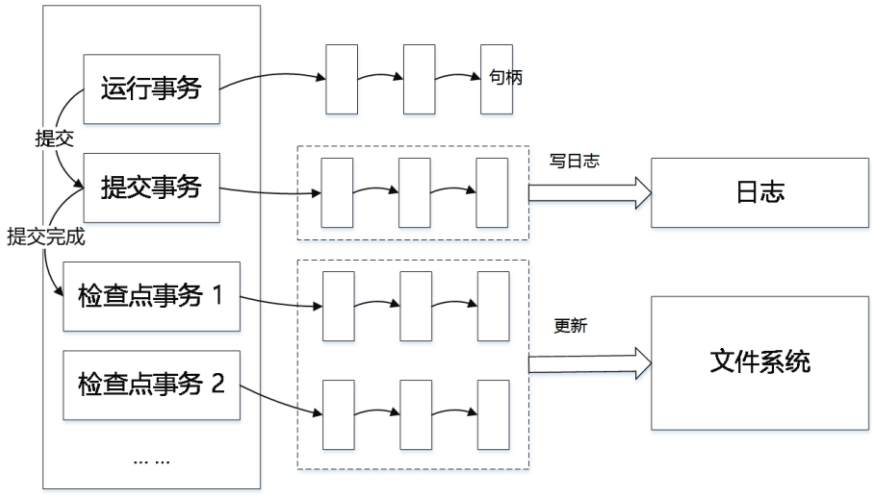

图 2.5 JBD 事务逻辑示图

然而并不是所有的应用场景都能使用 writeback 或 ordered 模式来优化日志机制。例如在 Ceph 之中，系统的很多功能，例如集群的故障恢复机制、快照以及分布式分层存储等功能，都要求底层存储引擎提供事务接口，并保证严格的数据一致性。此时类似于 writeback 或 ordered 模式的优化机制，虽然提升了性能，却可能破坏系统的可靠性。

2.2.3 基于新型存储器的日志机制优化

随着材料和存储技术的发展，很多新型存储器件进入人们视野，例如闪存[31,32]和 PCM[33,34]等非易失性内存等。如何利用这些新型存储器对现有存储系统进行加速成了研究热点。例如使用 device mapper 机制实现的 Flashcache[35]技术，借助非易失性内存存储文件系统元数据和日志的 PRIMS 系统，以及使用非易失性内存整合文件系统页缓冲和日志功能的 UBJ 机制等。这些技术根据存储设备的特性对存储系统的不同方面做了改进，其中 UBJ 机制是使用新型存储器优化文件系统日志机制的经典例子。下面简要阐述一下 UBJ 机制的实现原理。

前面说过，缓存的易失性是导致数据不一致的一个重要原因，而 PCM 等非易失性内存的出现弥补了这一方面的不足。然而非易失性内存的使用并不足以保证数据一致性，还需要实现文件系统更新事务的原子性。UBJ 机制的主要工作就是在使

用非易失性内存作为文件系统缓存时，实现文件系统更新事务的原子性。

在 UBJ 中，每个缓存页使用三个子状态的组合表示其当前的状态，即 normal/frozen，clean/dirty，以及 up_to_date/out_of_date。同时定义了三种类型的操作，即读/写操作，提交操作，以及检查点操作。图 2.6 是在这 3 种类型操作下，缓存页的状态变迁。

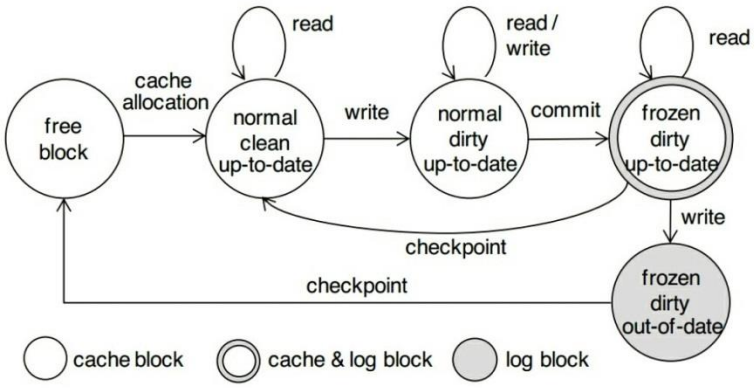

图 2.6 UBJ 缓存的状态变迁图

同 JBD 一样，UBJ 共有三种类型的事务，即一个运行事务，一个提交事务和多个检查点事务，每个事务是一条缓存页链表。其中运行事务中的缓存页状态为 normal/dirty/up_to_date，即可读写的脏页。当 UBJ 执行提交操作时，运行事务变为提交事务，其包含的缓存页的状态被更新为为 frozen/dirty/up_to_date，即只读的脏页。如果对这些缓存页进行写入，UBJ 会产生它们的拷贝，并将数据写入这些拷贝中，后续操作直接从拷贝中读写取数据，同时这些缓存页的子状态会变为 out_of_date 状态。和 JBD 不同，提交事务时只需修改缓存页的状态，而不需要写入磁盘上的日志文件，因而避免了写日志带来的磁盘开销。当所有缓存页状态更新完毕，事务就会变为检查点事务。当 UBJ 执行提交操作时，再将这些已经提交的事务写回文件系统。

UBJ 机制利用了非易失性内存实现了缓存与日志的整合，既避免了写日志的磁盘开销，也避免了以往单纯利用非易失内存做日志设备时的数据拷贝开销。

目前，非易失性内存容量仍然不大，且价格一直高居不下，这些因素限制了

UBJ 和 PRIMS 这类技术在实际系统中的应用和推广，尤其是在 Ceph 这种大规模集群中，大规模部署非易失性能内存是一个不菲的开销，与 Ceph 利用廉价设备构建高可用存储系统的出发点向违背。

2.3 热数据延迟写机制

Ceph 的存储引擎 FileStore 使用的日志机制，类似于 JBD 的 full 模式，记录了对象及其属性的全部更新历史，为上层逻辑提供严格的数据一致性保证，但也带来了两次写入的问题。同时客户端缓存的存在，使得 FileStore 很难借助文件系统的页缓冲进一步提升性能，因而改进日志机制成为提升性能的关键。

Ceph 的很多功能，例如故障恢复和对象管理等，都依赖存储引擎提供的事务接口，对数据一致性有严格的要求，因而牺牲数据一致性来换取性能做法是危险的，它会影响整个系统的可靠性。非易失性内存高居不下的价格，使得类似于 UBJ 这种日志优化方案的可用性降低，特别是在 Ceph 这种拥有成百上千个存储节点的集群环境下，大规模使用非易失性内存是一个不菲的开销。类似 Flashcache[31]的通过直接在现有存储系统中添加新型存储器的技术，并没有解决日志机制的两次写入问题。

在 SSD 作为日志设备的场景下，针对日志机制的两次写入问题，提出了热数据延迟写机制，它利用内存索引将日志变为热数据的临时存储空间，热数据写入日志即可实现热数据的更新；回收日志空间时再将热数据的最近一次更新写回文件系统，实现了热数据的延迟写回，降低了写磁盘的次数；对冷数据则采用传统的日志机制，即更新日志之后随即更新文件，避免了日志空间回收时的读开销。

2.3.1 内存索引

内存索引的功能是为了将日志文件变为热数据的临时存储空间。在 Ceph 中数据以对象的形式存在，当更新热对象的事务被追加至日志末尾时，就会建立该对象的内存索引，标识该对象相应偏移处最新的数据在日志中的位置；建立内存索引之

后，该对象就实现了更新，后续请求就能通过内存索引读取该对象保存在日志中的数据，当要读取的数据不在日志中时，再从磁盘上的文件系统读取。

内存索引还在日志空间回收时被用来快速定位日志头部的的有效数据（即未被后续写操作覆盖的数据）。为了实现快速定位日志头部的有效数据块，内存索引为日志中的每个有效数据片建立一个描述符，描述该数据片的长度，在日志中的位置，以及在对象中的偏移等；并将这些描述符按照数据片写入日志的先后顺序构建成一个链表，回收日志空间时就能按照该链表快速地定位日志头部的有效数据了。

2.3.2 双日志结构

双日志结构是指用两个独立的存储空间分开存放冷热数据的更新历史，避免空间回收时额外的读开销。如果不区分冷热数据，延迟所有数据的写操作，且它们的更新历史存放在同一个日志中，则如图 2.7 所示，热数据因为经常更新而集中在日志的尾部，无须写回到文件系统；但是冷数据因为更新频率低在写入日志后还需要读出来写回文件系统，不但没有提升性能，反而因为增加了额外的读取操作而影响性能。

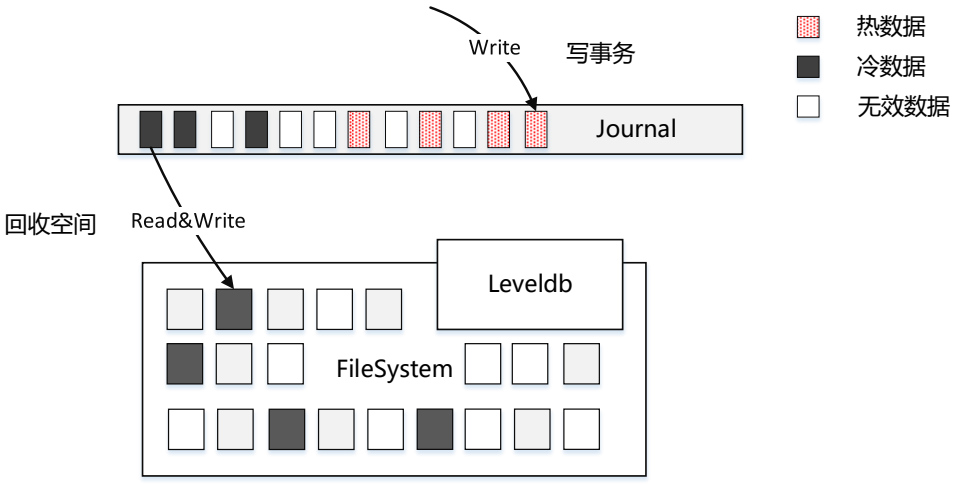

图 2.7 延迟写机制

除此之外，双日志结构能够有效利用日志空间，减少写磁盘的次数。因为日志空间的越大，热数据延迟写回文件系统的时间就越长，写磁盘的次数就越少，而冷

数据在写入日志之后随即写回对应的文件，因而日志空间的大小对于冷数据的写性能影响不大，所以冷热数据分离的双日志结构可以限制冷数据占用的日志空间，将大部分的日志空间留给热数据，有利于减少热数据写磁盘的次数。

如图 2.8 所示，存储引擎在双日志结构下的写流程，其中 HotJournal 和 ColdJournal 分别用来记录热数据和冷数据的更新记录。当用户更新热数据时，系统将对应的操作记录写入到 HotJournal，建立内存索引，并在日志回收时将热数据最近的一次更新写回文件系统；当用户更新冷数据，系统将更新记录写入 ColdJournal，同时也将数据写回文件系统。

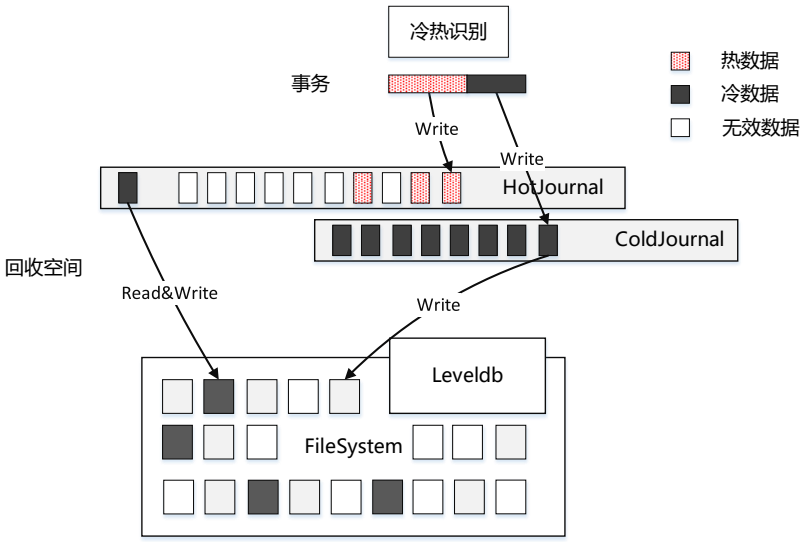

图 2.8 热数据延迟写机制

2.3.3 空间回收机制

空间回收是在日志的空闲空间低于某个阈值时，释放旧事务占用的空间，供后续事务使用的过程。由于 HotJournal 和 ColdJournal 机制不同，它们的空间回收方式也不一样。对 ColdJournal 来说，它记录了冷数据的更新历史，而冷数据的更新同样也存在于内核的脏页中，因此只需要将脏页中的更新持久化回文件系统，就可以释放对应的日志空间。由于脏页持久化回文件系统需要一段时间，为了避免后续事务的阻塞，应该留有足够的剩余日志空间供后续事务使用，比较保险的做法是使

用总空间的 50%作为触发 ColdJournal 空间回收的阈值。为了避免内存中积累过多的脏数据而导致的写回时间过长，同时也为了减少故障重启时恢复时间，因此每隔一段时间（3~10s）就会主动地将内存中的脏数据写回文件系统，进而释放 ColdJournal 的空间。

对 HotJournal 来说，它记录了热数据的更新历史，而热数据并不存在于内核的脏页中，因此需要根据内存索引定位被回收空间中的有效数据，然后将它们读取出来并持久化到文件系统，持久化完成之后就可以释放对应的日志空间。同样，为了避免阻塞后续事务，应该给后续事务留有足够的空闲空间，即设定合理的触发 HotJournal 空间回收的阈值。同时，为了避免回收空间时对日志的大量读操作，还应该控制热数据的数量，例如热数据的总量应该在 HotJournal 总空间的 50%左右。

由于触发 HotJournal 空间回收的阈值越高（推荐值为 0.25），则 HotJournal 用来存储热数据的空间越大，系统性能的提升也就越明显，但也导致系统故障重启时恢复时间越长。注意到在故障恢复时，对 HotJournal 只需根据元数据建立内存索引即可，针对这一特点，可以将元数据的更新单独组织到一个日志中，即使用三个独立的日志分别记录元数据、热数据和冷数据的更新历史，系统故障重启时，只需要快速扫描元数据日志即可构建 HotJournal 的内存索引，实现快速的故障恢复，如何利用三日志结构实现故障快速恢复是后续的优化工作，后面阐述热数据写推迟机制时仍然基于双日志结构。

2.3.4 故障恢复机制

故障恢复机制用来在故障重启时利用日志恢复被打断的事务，从而确保事务的原子性。事务中往往包含多个更新操作，且每个操作的目标对象的冷热程度也不一样，因此事务可能被单独存放在 HotJournal 或 ColdJournal，也可能被拆分后分别存放在 HotJournal 和 ColdJournal；又因为 HotJournal 和 ColdJournal 的空间回收是互相独立的，所以在某个时刻，被拆分的事务可能因为部分数据被提前释放而无法还原。这些原因导致了故障恢复机制的复杂性。

为了简化故障恢复过程，将事务的元信息（即描述各个子操作的信息）和热数据封装为日志项存放在 HotJournal，将事务的冷数据封装为日志项存放在 ColdJournal，且两个日志项的序列号一致，并规定事务在 HotJournal 中的部分必须等 ColdJournal 中的部分被释放才能释放。在这个规定下，系统在故障重启时首先分别确定 HotJournal 和 ColdJournal 中第一个日志项的序列号，假设分别为 a 和 b，然后对于 HotJournal 中序列号位于[a,b)之间的事务，直接根据元数据信息建立热数据的内存索引，对于序列号大于等于 b 的事务则通过合并 HotJournal 和 ColdJournal 中的日志项恢复事务所有的信息，并重新执行事务。通过上述机制就能恢复被打断的事务，确保数据的一致性。

2.3.5 冷热数据识别

冷热数据识别是根据对象的更新历史判断它们的冷热程度。为了描述对象的冷热程度，为每个对象定义了一个热度值，热度值越高说明对象更新越频繁。热度值使用如下公式来计算。

$$heat = \boldsymbol{a}^{\boldsymbol{b}*\frac{(\boldsymbol{now}-\boldsymbol{last})}{\boldsymbol{time}}} * heat + (1-a) * new\_heat \quad \text{(公式 2.1)}$$

$$last = now \quad \text{(公式 2.2)}$$

其中 now 和 last 分别表示当前时间和上一次更新对象热度值的时间，time 则被设置为写满 HotJournal 所需的时间。a 是一个大于 0 但小于 1 的常数，用来控制对象热度衰减的速率；new_heat 是每次更新对象时，热度的增幅。在 new_heat 为 1 的情况下，热度值 heat 约等于该对象在的周期 time/b 内的平均更新次数。

在上述公式中，对象的热度值随时间呈指数衰减，因而对于周期性的事务，对象的热度值在系统空闲阶段迅速降低，导致对象在事务再次到来时由于热度值太低而无法及时优化。为了能够处理周期性事务，这里引入“虚拟时间”的概念，“虚拟时间”用写入存储系统的字节数来定义，当存储系统空闲的时候，写入系统的数据量减少，“虚拟时间”前进的速度也相应地减缓，对象也就能长时间维持系统忙碌时候的热度值，从而在事务再次到来时能够即使优化。

因为 HotJournal 的空间有限，所以只能选取那些热度超过阈值，且热度排名靠前的对象作为热对象。热对象的数量通过如下公式计算:

$$\mathrm{num}_{\mathrm{hot}} = a \times Size/Size_{object} \quad \text{(公式 2.3)}$$

其中 a 是一个大于 0 小于 1 的常数，用来控制 HotJournal 中有效数据占据的空间比例。Size 为 HotJournal 的空闲空间的大小。因此随着系统的运行，HotJournal 的空闲空间逐渐减少，热对象的数量也会越来越小，直至 HotJournal 的空闲区间区域稳定。

通过以上两个公式就能根据 HotJournal 当前的使用情况和对象的更新历史确定合理的热对象个数和它们的热度值。

2.4 本章小结

本章首先详细介绍了分布式存储系统 Ceph 的架构，解释了底层存储引擎利用日志提供事务接口的原因，接着介绍了日志机制的原理以及当前的研究现状。最后就存储引擎日志机制的两次写入问题以及当前优化方案存在的缺点，提出了热数据延迟写机制，并解释了该机制的基本工作原理和关键技术。下一章将讲述基于热数据延迟写机制的存储引擎的设计与实现。

3 日志机制的优化与实现

本章首先提出基于热数据延迟写机制的存储引擎的实现框架，然后详细叙述存储引擎在工程实现中使用的关键技术，包括双日志结构的实现、故障恢复、事务的 pipeline 处理、持久化层的异步快照、内存索引的组织、对象热度值的实时排名技术等。这几个重要技术保证了热数据延迟写机制的正确运行，同时实现了事务的快速处理，是存储引擎性能提升的工程保障。

3.1 存储引擎的实现框架

如图 3.1 所示，基于热数据延迟写机制的存储引擎包含 6 个子模块，即冷热数据过滤与标记模块，事务执行模块，双日志模块，内存索引和缓存模块，持久化模块，以及日志空间回收模块等。

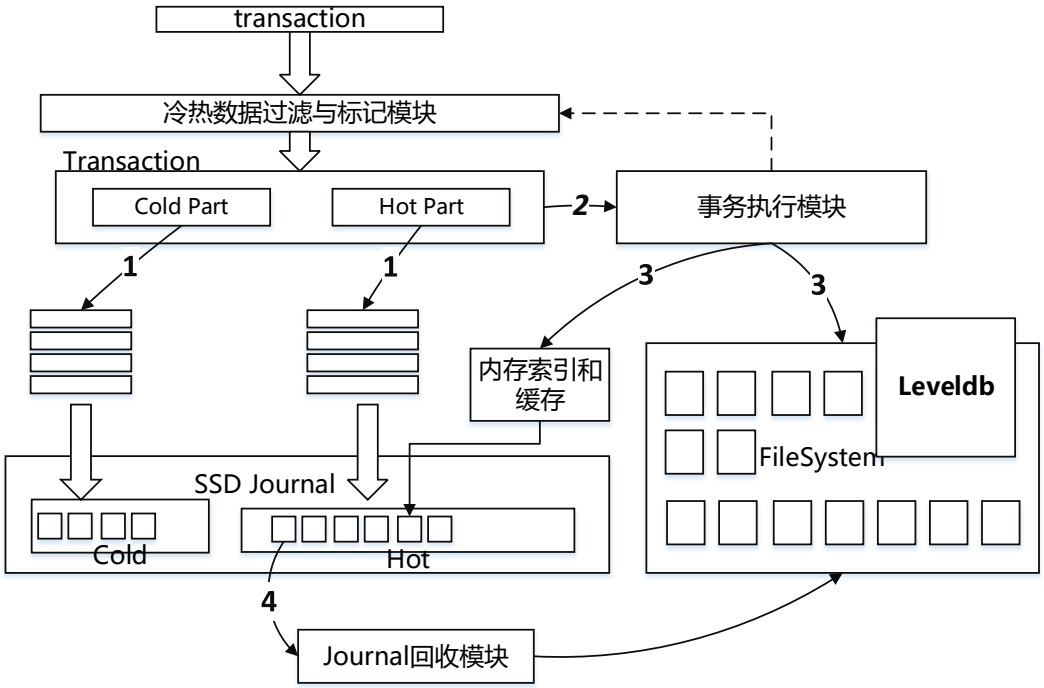

图 3.1 双日志结构体存储引擎架构图

冷热数据过滤与标记模块负责识别事务中的冷热对象，并将其更新操作分别记录到 HotJournal 和 ColdJournal 中。该模块使用一个排序树对对象的热度值进行实

时排序，并将热度值靠前的对象在哈希表中做上标记，标识该对象是一个热对象。上层提交一个事务时，该模块借助哈希表快速分辨出事务中的冷热数据，并对冷热数据做上标记，然后将多个事务的冷数据封装为一个日志项写入到 ColdJournal，将多个事务的元信息和热数据封装为一个日志项写入到 HotJournal。当日志项在 HotJournal 和 ColdJournal 中都成功落盘后，再将事务提交给事务执行模块执行。

双日志模块负责记录对象的更新历史，并对日志空间进行管理。该模块包含两个部分，即 HotJournal 和 ColdJournal。其中 HotJournal 中的每个日志项包含了多个事务的元数据信息，元数据描述了事务中子操作的具体信息，例如写操作的目标对象、偏移和长度，或者更新的属性等；如果写操作的目标对象为热数据，则写入的数据也包含在日志项中，如果目标对象为冷数据，则日志项中就会做上标记，表示写入的数据存放在 ColdJournal 中。ColdJournal 中的日志项则只包含事务中的冷数据。为了跟踪日志的使用状态，该模块使用多个指针指示 HotJournal 和 ColdJournal 的首尾位置，事务执行的位置以及空间回收的位置等。这些指针会周期性地持久化相应的文件中，以便故障重启时能够从相应的位置进行事务回放。

事务执行模块负责解析并执行事务。事务中往往包含了多个修改对象数据和属性的子操作，如果子操作更新对象数据且目标对象被标记为热数据，则建立相应的内存索引，后续操作通过索引就能在 HotJournal 中读取该对象相应偏移处的数据；如果目标对象为冷数据，则直接将数据写回文件系统。如果子操作更新对象的属性，则直接将该属性缓存到内存中。由于放置组间的隔离性，不同放置组的事务可以并发执行，因此事务执行模块采用线程池来实现，同一时刻会有多个事务在并发执行，从而增加系统的吞吐。在执行过程中，还会通过反馈机制来更新每个对象的热度信息，使系统能够实时计算出当前的冷热数据。

内存索引和缓冲区负责建立热数据的索引和属性缓冲区。HotJournal 中存放了热对象的数据，通过建立指向 HotJournal 的索引就能使后续请求直接从 HotJournal 中读取相应对象的数据。该模块还缓存了对象的属性，由于 HotJournal 已经持久化了属性的更新操作，因此将属性直接缓存在内存中后即可返回成功。当回收

HotJournal 空间时，再将被回收空间中的有效脏数据和属性批量写回文件系统，并删除对应的索引信息。

日志空间回收模块负责在日志可用空间不足的时候释放空间共后续事务使用。当 HotJournal 的空闲空间低于某个阈值（例如 0.25）时，就会回收 HotJournal 头部的空间，通过内存索引就能定位被回收空间中包含的有效数据和属性，然后将有效数据从 HotJournal 中读取出来并写回到文件系统，并将内存缓冲区中的对应的有效属性批量写回 leveldb 中。由于热数据经常更新，HotJournal 头部的有效数据在很大概率上会被后续的操作覆盖而变为无效数据，因而只用少量的读取操作就能在 HotJournal 头部回收大量的空间。当 ColdJournal 的空闲空间低于某个阈值(例如 0.5)时，就会将内存中的脏数据刷写回磁盘，然后释放对应的空间。

持久化模块负责持久化对象及其属性。持久化模块使用 btrfs 文件系统存储对象数据，同时还使用建立在 btrfs 之上的 leveldb 数据库来存储对象属性。之所以使用 btrfs 文件系统，是因为它提供了异步快照的功能，在创建快照的时候不会阻塞后续的读写访问，避免过长的 I/O 停顿。存储引擎使用 btrfs 提供的接口周期性地创建地快照，保存文件系统的状态和日志的指针，在重启时先将文件系统回滚到最近的快照，然后从日志指针指示的位置开始回放事务，将系统还原到故障前的状态。

在基于热数据延迟写机制的存储引擎中，事务中的冷热数据被标识出来，并分别写入到 HotJournal 和 ColdJournal 中，由于热数据更新频率高，多次更新操作中只有最新一次更新才会被写回文件系统，因而减少了更新文件系统的 I/O 操作，提升了系统的性能；冷数据更新频率低，无法通过 I/O 合并的方式降低对文件系统的 I/O，因而在写入日志后直接写回文件系统，避免回收日志空间时额外的读取操作。

3.2 双日志结构的实现

双日志结构利用 ColdJournal 和 HotJournal 分别记录冷热对象更新历史的，这里详细描述它的具体实现细节。

一个事务由两部分构成，即用来描述子操作的元数据信息，以及要写入的数据，

图 3.2 是一个事务的元数据部分和数据部分，它包含了 4 个子操作，即两个写入操作和两个属性更新操作。

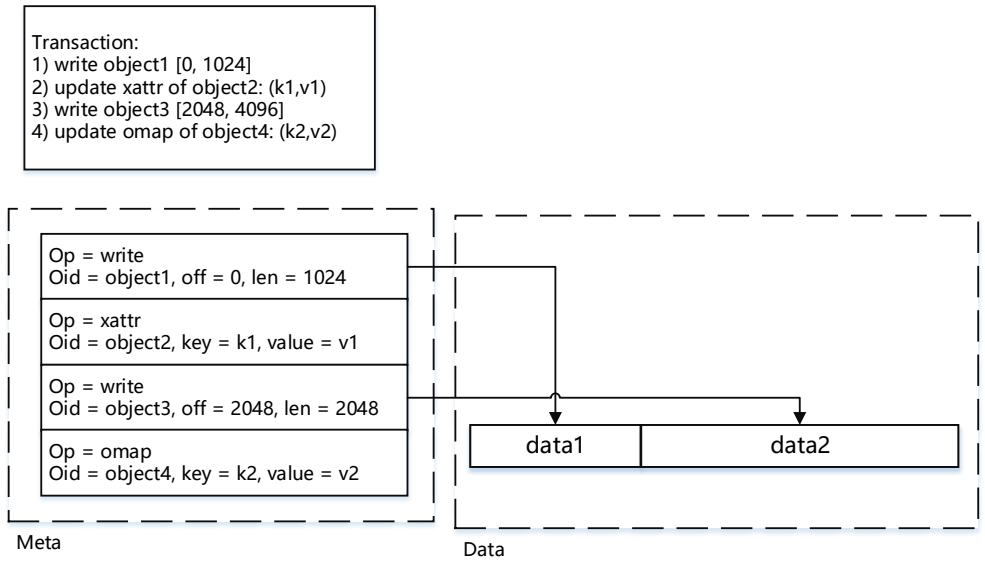

图 3.2 事务结构图

在双日志结构下，事务在经过冷热数据过滤和标记模块标记之后，其元数据和热数据被封装到一个日志项中，并写入到 HotJournal，而其冷数据则被封装到另一个日志项中并写入到 ColdJournal。

图 3.3 是 HotJournal 中日志项的格式，包括 header 和 tail 结构体，前后 pad 区域，以及元数据区域和热数据区域。为了提升系统的吞吐，每个日志项中会封装多个事务，因此其元数据区域包含了多个事务的元数据信息。日志项的 header 结构体描述了该日志项的序列号，包含的事务数量，pad 区的长度，元数据区域和数据区域的长度，冷对象集合，以及校验码等。tail 结构体是 header 的副本，用来代表日志项的结束。pad 是一个空白区域，用来确保数据区域的起始位置刚好以 4KB 对齐，从而加快从日志中读取数据的速度。

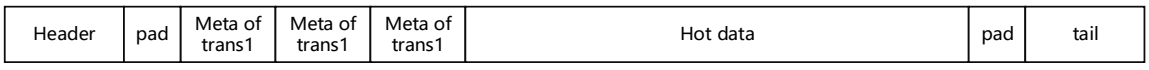

图 3.3 HotJournal 日志项结构

图 3.4 是 ColdJournal 中日志项的格式，包含了 header 和 tail 结构体，前后 pad

区域，以及冷数据区域。其 header 结构体只记录了日志项序列号，pad 区域长度，冷数据区域长度，以及校验码等。tail 是 header 的副本，用来验证日志项的完整性。

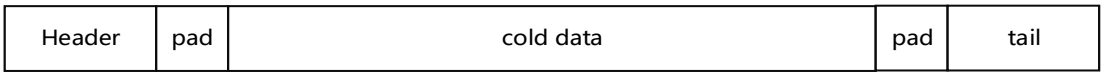

图 3.4 ColdJournal 日志项结构

要注意的是，同一个事务的冷热数据会被分别写入序列号相同的冷热日志项中，且 Hot data 和 cold data 区域中数据的组织顺序与元数据区域中事务的顺序一一对应，因此结合序列号相同的两个日志项就能还原出事务所有的信息。

在双日志结构下，HotJournal 和 ColdJournal 都是一个环状的日志项数组，新提交的日志项被追加到 HotJournal 和 ColdJournal 的尾部，旧日志项则在回收空间的时候从 HotJournal 和 ColdJournal 尾部被释放，从而腾出空间给新的日志项使用。为了方便管理日志的空间，下面将采用指针的方式来跟踪日志空间的使用情况。

如图 3.5 所示，HotJournal 使用了 6 个指针来管理空间。其中 start_pos 和 write_pos 分别指向 HotJournal 的首尾；[trans_done_pos，write_pos)区间内的事务刚写入 HotJournal，还没有被执行；[data_evict_pos, trans_done_pos)区间内的事务已经执行，但包含的有效热数据还没有写回到文件系统；[data_sync_pos, data_evict_pos)区间内的事务已经被执行，其中的有效数据已经被空间回收线程写回文件系统的 Page Cache；[start_pos, data_sync_pos)内的事务已经执行，且该区间内的有效数据已经写回到文件系统，并同步回磁盘了，data_sync_pos 指针也被记录在磁盘上。

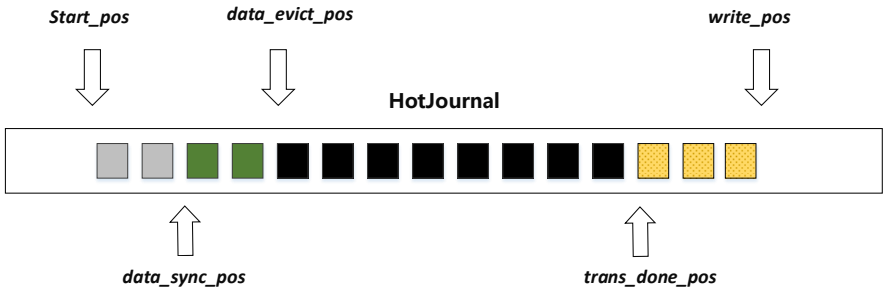

图 3.5 HotJournal 指针

如图 3.6 所示，ColdJournal 只需要 3 个指针进行空间管理。其中 start_pos 和

write_pos 分别指向 ColdJournal 的首尾，start_pos 之前的冷数据已经同步回磁盘，[start_pos, trans_done_pos)之间的冷数据已经写回文件系统，但还没有同步回磁盘，trans_done_pos 之后的冷数据则刚刚写入 ColdJournal。

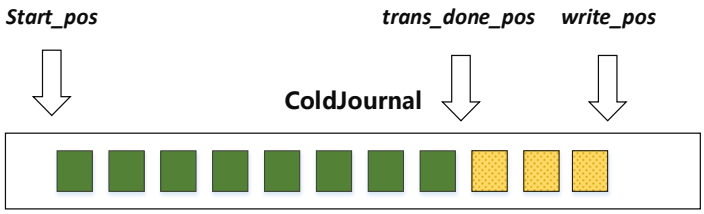

图 3.6 ColdJournal 指针

下面通过一个事务在双日志结构中的生命历程来讲述如何利用这些指针对日志空间进行管理的。

1、事务刚被写入到 HotJournal 时，它处于[trans_done_pos，write_pos)；其冷数据则落在 ColdJournal 的[trans_done_pos，write_pos)；

2、事务被执行后，trans_done_pos 向前推进，此时它落在于[data_evict_pos, trans_done_pos)区域；而冷数据则落在[start_pos, trans_done_pos)区域；

3、当可用空间[write_pos, start_pos)低于某个阈值（例如 0.8）时，回收线程回收 HotJournal 头部的空间，有效数据被读取出来并写回到文件系统，此时 data_evict_pos 向前推移，事务落在[data_sync_pos, data_evict_pos)区域；

4、对文件系统进行周期性快照，data_sync_pos 被更新为 data_evict_pos，此时事务落在[start_pos, data_sync_pos)区域；而 ColdJournal 的 start_pos 指针被更新为 trans_done_pos，事务的冷数据部分则落在 start_pos 之后的区域；

5、当新事务需要新空间时，HotJournal 的 start_pos 向前推移，指向事务包含的有效数据的内存索引被删除，事务生命终结。

3.3 故障恢复的实现

故障恢复是利用持久化层的快照和双日志中记录的事务信息，将系统恢复至故

障前的状态的过程，这里详细阐述故障恢复过程的细节。

持久化层使用的文件系统为 btrfs，利用 btrfs 的异步快照功能保存某时刻文件系统的状态，再结合日志就能恢复机器故障前的状态。如图 3.7 所示，创建快照的时候，HotJournal 的 data_sync_pos 指针会被更新为 data_evict_pos 指针，同时将 data_sync_pos 指针和 trans_done_pos 指针会被保存到对应的文件中；ColdJournal 的 start_pos 被更新为 trans_done_pos 指针，也被保存到文件中。这 3 个指针用来在机器重启的时候恢复数据。

图 3.8 是重启时事务回放的流程。可以看出，对于在 HotJournal 的[data_sync_pos, trans_done_pos)区域中的事务，由于其冷数据已经写回文件系统，因此只需要根据事务的元数据信息构建热数据的内存索引并建立属性缓冲区即可；对于在 HotJournal 的 trans_done_pos 之后的事务，由于快照回滚时其对文件系统的更新都被删除，因此需要结合 ColdJournal 中序列号相同的日志项还原出这些事务所有的信息，然后再重新提交给事务执行模块执行。当文件系统再次恢复到故障前的状态后，根据日志的当前使用情况重新设置所有的指针，此时即可重新对外服务。

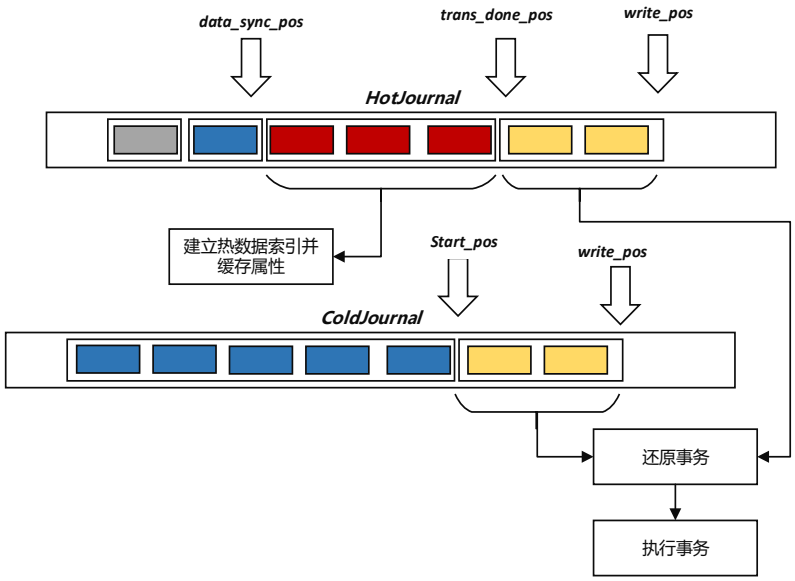

图 3.7 Journal 的故障恢复

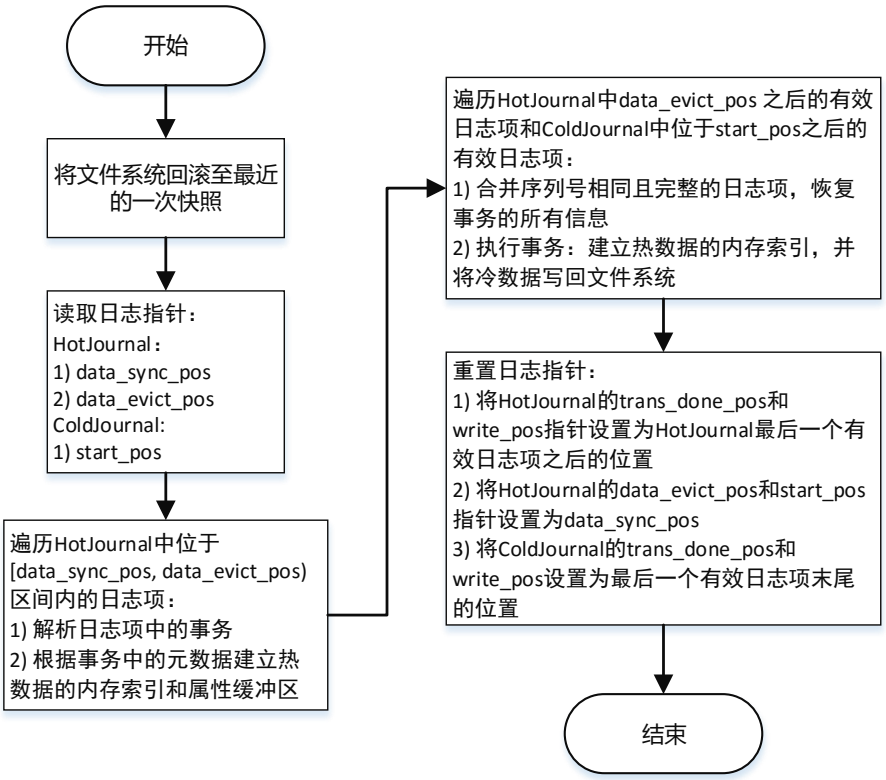

图 3.8 双日志结构下故障恢复的流程图

3.4 事务流水线处理的实现

事务流水线处理利用多线程提高事务处理速度的一种技术，这里阐述它的原理和实现。

在双日志存储引擎中，事务要经过冷热数据标识，封装为日志项，写入日志，解析并执行事务等过程。为了提高并发性并减少等待时间，存储引擎使用流水线的方式来处理事务，即将一个事务的处理分为多个阶段，每个阶段用一个线程进行处理，从而在同一个时刻就会有多个事务在并发处理，且每个事务处于不同的阶段。其处理过程如下。

1、将事务放入等待队列。

2、线程1从队列中批量获取多个事务，并识别事务中的冷热数据，然后根据识别出来的冷热数据构建写入HotJournal和ColdJournal的日志项；利用aio接口

向HotJournal和ColdJournal发起异步写入操作。

3、线程2等待异步写入完成，将对应的事务放入事务执行模块的等待队列。

4、线程池(事务执行模块)从等待队列中获取事务，然后解析并执行该事务。由于是线程池，因此同一时刻会有多个事务在并发执行。执行完事务后，将事务放入回调等待队列。

5、线程3，即回调线程，从等待队列中依次获取完成的事务，并执行相应的回调函数。

通过将事务的各个处理阶段交由不同的线程进行处理，从而实现事务的流水线处理，减少了等待时间。

3.5 持久化模块的实现

持久化模块负责在磁盘上利用文件系统或者数据库来存储对象及其属性。这里详细介绍它的实现细节。

持久化模块使用 btrfs 文件系统存储对象，每个对象都对应到一个文件；使用 leveldb 数据库存储对象的属性，包括对象的 xattr 和 omap。由于 leveldb 最终也以文件的形式将属性存储在 btrfs 文件系统中，所以使用 btrfs 提供的接口创建快照时可以持久化某时刻对象的数据和属性。

在系统重启的时候，文件系统会回滚到最近的一次快照，该快照之后写入文件系统的数据都会被遗弃。因此在将有效数据或脏数据写回文件系统之后，释放 HotJournal 或 ColdJournal 对应空间之前，先对文件系统创建一个快照，确保快照成功创建之后才能释放对应的日志空间，避免因为日志空间的过早释放而导致数据丢失。

在创建快照时还需要记录此刻日志的指针，以便故障重启时能够从日志正确的位置开始进行事务重放。需要保存的指针有 HotJournal 的 data_sync_pos 和 trans_done_pos 指针，以及 ColdJournal 的 trans_done_pos 指针，这里以 trans_done_pos 指针为例讲述用来保证日志指针正确的机制。

由于事务执行模块采用线程池的方式来实现，同一时刻会有多个事务正在执行，且事务执行的顺序可能和日志中记录的顺序不一致，因此必须使用相应的机制来保证 trans_done_pos 指针的正确性，即该指针之前的事务都已经执行，之后的事务尚未开始。这里采用如下的方式来确保 trans_done_pos 的正确性。

将日志项异步写入日志并得到响应后。

（1）获取锁。

（2）检查 creating_snapshot 标示，如果为 true，则条件等待 creating_snapshot 为 false。

（3）将 inflighting_requests 加 n（日志项项中的事务数），将 trans_doing_pos 设置为当前日志项的位置。

（4）释放锁。

（5）将日志项中的所有事务交由事务执行模块中的线程池来执行。

事务执行模块每当执行完一个事务时。

（1）获取锁。

（2）将 inflighting_requests 减 1。

（3）如果 creating_snapshot 为 true，且 inflighting_requests 为 0，则向等待创建快照的线程发送一个唤醒信号。

（4）释放锁。

在创建快照时，按照如下流程执行。

（1）获取锁。

（2）将 creating_snapshot 设置为 true。

（3）条件等待 inflighting_requests 为 0。

（4）将 trans_done_pos 设置为 trans_doing_pos。

（5）释放锁。

（6）调用 btrfs 异步接口对文件系统创建一个快照。

（7）获取锁。

（8）将 creating_snapshot 设置为 false。

（9）向等待 creating_snapshot 为 false 的线程发送信号。

（10）释放锁。

（11）等待异步快照完成。

（12）释放 Journal 空间等。

如果创建快照时刚刚写入日志的日志项为 a，则通过以上流程，存储引擎会阻止 a 之后的事务流入事务执行模块，并在 a 之前的事务都执行后时才创建快照，从而确保了 trans_pos_done 指针的正确性。同时，向 btrfs 发起异步快照请求之后无需等待快照落盘，即可开始 a 之后事务的执行，从而避免了在创建快照到快照成功落盘阶段的事务执行模块的长时间停顿。

为了避免开始执行快照到快照落盘的时间过长，同时也为了让系统够得到一个平稳的吞吐，需要限定持久化层在 page cache 中的脏数据量。为此，在持久化层中加入了“写回限定器”，使得持久化层按照如下的策略执行写回操作。

（1）使用 LRU 链表维护最近被更新的文件描述符，使用一个全局变量来维护脏数据数量。

（2）当脏数据量或者脏文件描述符的数据超过阈值时，删除 LRU 链表末尾的文件描述符，即更新频率最低的文件，并调用 sync 接口将脏数据强制写回文件系统。循环执行该过程，直到脏数据或者脏文件描述符低于阈值。

（3）当定时器超时时，从 LRU 链表末尾开始，强制写回一段时间内没有更新操作的文件描述符。

“写回限定器”使得文件系统 page cache 中的脏数据量能够限制在某个阈值下，从而避免快照操作耗费太长的时间，也保证了系统吞吐的稳定。

3.6 内存索引和缓存的实现

在底层存储中，PG 与目录相对应，对象与目录中的文件对应，因此内存索引也按目录的层次结构组织。如图 3.10 所示，索引模块通过一个目录哈希表查询与

PG 对应的目录，该目录又使用一个对象哈希表来索引其包含的对象。每个对象由分散在 HotJournal 中的多个有效数据片段构成，索引模块使用红黑树将这些分散的片段组织成一个按对象中偏移量有序的树，树的每个节点都指向一个描述符，其结构体如图 3.9 所示。

```
struct BufferHead {
    ObjectRef owner; // 有效数据片段所属的对象
    struct extent{
        uint32_t start;
        uint32_t end;
     } ext;                 // 分片在对象中的偏移量
    uint32_t bentry;        // 所属日志项的起始地址
    uint32_t boff;          // 在日志项中的偏移量
    bool dirty;             // 是否写回文件系统
}
```

图 3.9 BufferHead 结构图

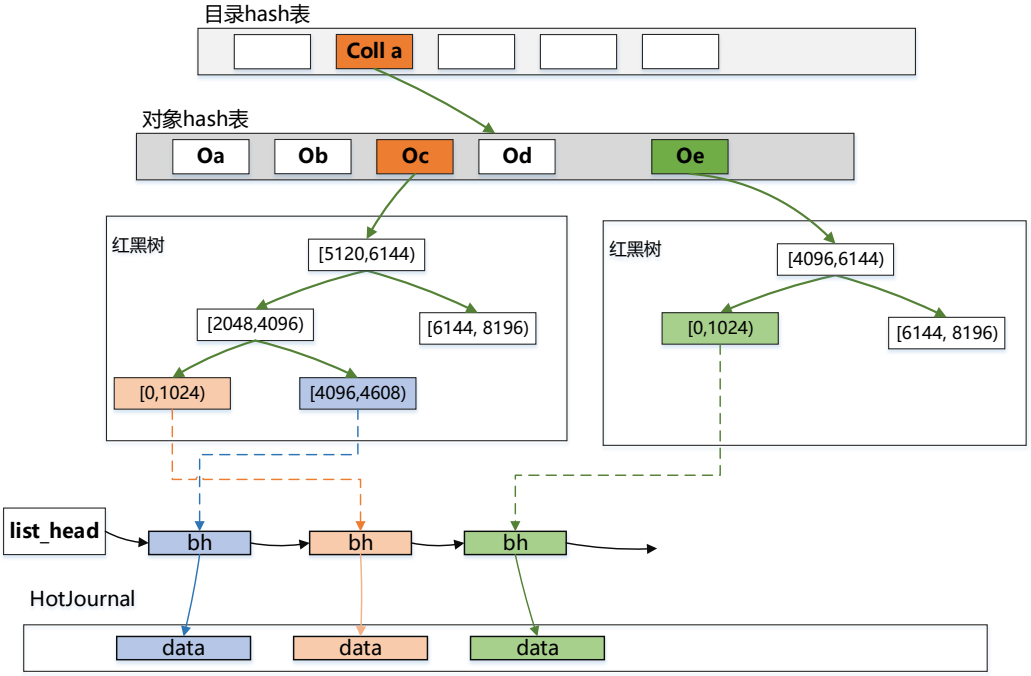

图 3.10 对象的内存索引结构图

描述符 BufferHead 用来描述 HotJournal 中数据片段所属的对象，在对象中的偏移，以及在 HotJournal 中的位置。为了便于 HotJournal 空间的回收，这些描述符按照有效数据写入 HotJournal 的先后顺序依次存放在一个链表中，后台线程在回收

空间的时候从链表头开始依次将对应的有效数据片段写回持久化层，释放对应的空间，并删除指向描述符的索引。

由于对象某偏移处的数据可能被多次更新，因而导致描述符 BufferHead 指向的数据片段部分或者全部失效，此时内存索引会对 BufferHead 进行如下处理。

(1) 当 BufferHead 指向的数据完全被覆盖时，红黑树对应的节点指向新的 BufferHead；旧的 BufferHead 则通过将成员 extent.start 和 extent.end 都设置为 0 标示对应的数据片段无效。

(2) 当 BufferHead 指向的数据片段被部分覆盖，且数据片段的前半段或者后半段仍然有效时，则修改 BufferHead 的成员 extent.start 或者 extent.end，标示剩余的有效数据在对象中的偏移；修改成员 boff，标示剩余有效数据在日志项中偏移；修改红黑树，使得对应偏移处的节点指向该 BufferHead。

(3) 当 BufferHead 指向的数据片段的中间部分被覆盖时，则从红黑树中删除指向该 BufferHead 的节点。然后创建两个新的 BufferHead，使其分别指向首尾两部分有效数据，但这两个 BufferHead 不会加入到 list_head 指向的 BufferHead 链表中。

当回收线程处理链表头部的描述符 BuffetHead x 时，如果成员 extent.start 和 extent.end 都为 0，则说明是无效数据片段，直接删除并释放该 BufferHead 结构体即可；否则，根据内存索引模块处理 BufferHead 规则中的（2）和（3），其指向的区间内可能包含有效数据，此时在红黑树中遍历该区间内所有的节点指向的 BufferHead，如果它们与 x 来自同一个日志项，则删除并释放它们，同时将对应的有效数据片段写回持久化。

对象热度值随负载实时变化，在某个时刻它可能被标记为热数据，而下一刻又可能被标记为冷数据，因此它的数据可能部分存放在 journal，部分存放在持久化层。当用户读取某个对象相应偏移处的数据时，存储引擎通过遍历该对象对应的红黑树索引，建立一个 BufferList 链表，和一个 extent 链表，前者标示能够从 HotJournal 中读取的数据片段，后者标示需要从持久化层中读取的剩余数据片段，然后根据这两个链表分别从 HotJournal 和持久化层读取所有的数据片段，并根据所在的偏移将

这些零碎的数据片段整合成一个数据片段返回给用户。

对象属性缓存和上面的类似，如图 3.11 所示，使用红黑树来索引对象的属性，树节点指向一个用来描述该属性的结构体，同时该结构体也记录了对应的日志项位置。将这些描述属性的结构体按照写入 HotJournal 的先后顺序依次排列在一个链表中，以便在回收了相应日志项前，将相应的属性写回文件系统。

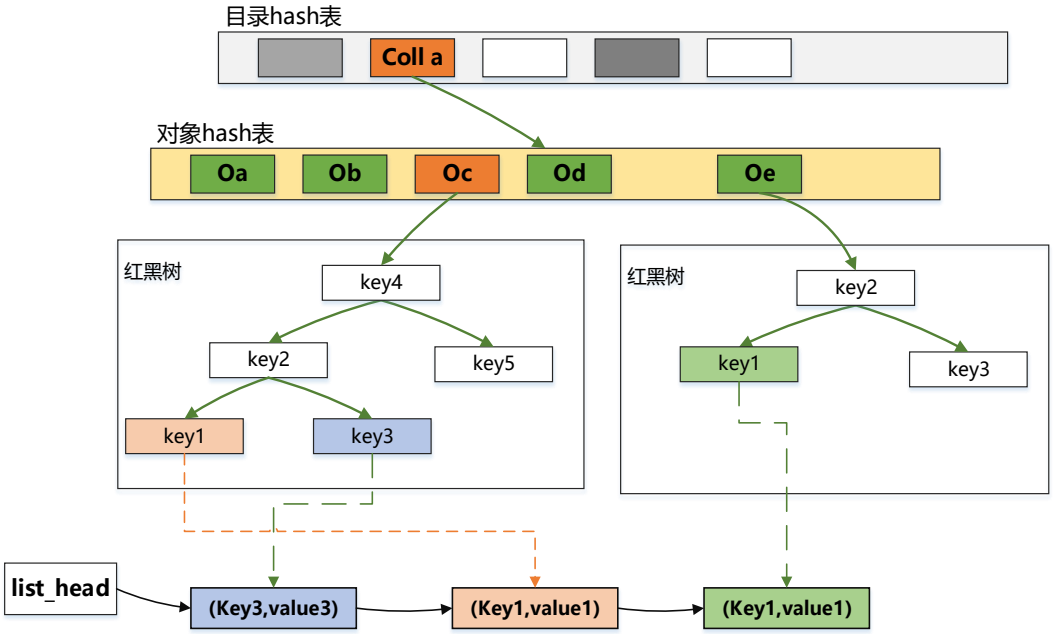

图 3.11 属性缓冲区

除此之外还需要跟踪属性缓存使用的内存空间，当内存使用超过某个阈值时，按照链表的顺序将对象的属性写回文件系统或者 leveldb。

3.7 热度值实时排名的实现

在执行事务的时候需要更新对象的热度信息，然后将热度信息反馈给冷热数据过滤与标记模块。对象的热度值通过 2.3.5 章节中的公式 2.1 和公式 2.2 来更新，而热对象的数量则通过 2.3.5 章节中的公式 2.3 来计算。

然而不是每次在更新对象时都会更新其热度值，只有在被更新的区域曾经被更新过时，对象的热度值才会更新。因此，每个对象都使用一个 64 比特的 bitmap(每个 bit 代表 64kb 的区间)来跟踪一段时间内被修改的区域。当对象被修改时，相应

的比特就会被置为 1；当对象在设定的时间内没有被修改时，则被重置为 0。只有当修改的区域对应的比特也为 1 时，才会更新对象的热度值。

由于每个对象的热度值都在随时间衰减，且又要随时获得热度靠前的对象，这里借助被修改的红黑树和哈希表实现热度的实时排名。图 3.12 是使用的具体数据结构。

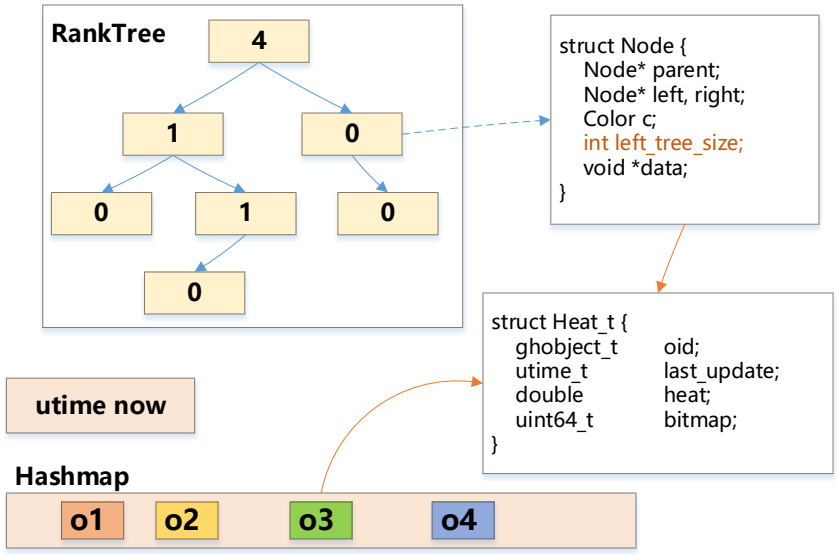

图 3.12 热度值实时排名结构

now 是一个全局的 utime_t 类型的变量，表示更新或读取对象热度值时的“虚拟时间”。

Hashmap 是一个哈希表，通过哈希表就能定位用来描述某个对象热度值的结构体 Heat_t。在 Heat_t 中，记录了每个对象上次更新的时间 last_update 和热度值 heat，通过前面的热度计算公式可以知道，只需要结合全局变量 now 就能计算出当前对象的热度值；而且只需要更新一下全局变量 now，就能实现所有对象热度值的衰减。Heat_t 中的 bitmap 用来跟踪对象被修改的区域。

RankTree 是一个被修改的红黑树，树节点加入了一个新成员变量 left_tree_size，标识左子树中节点的个数，通过 left_tree_size 就能快速得到被删除（添加）节点的排名，或者通过排名迅速得到相应的节点。例如，图 3-15 中 RankTree 根节点的 left_tree_size 为 4，表示其左子树中共有 4 个节点，如果此时要查询排名为 3 的节

点，则直接在左子树中递归查找排名为 3 的节点；如果查找排名为 4 的节点，则直接返回该根节点；如果查找排名为 7 的节点，则在右子树中递归查找排名为 2 的节点。其他操作也以此类推。

图 3.13 是更新对象热度值的流程图，其中 n 标识热对象的个数。图 3.14 是通过 bitmap 判断是否更新对象热度值的流程图。

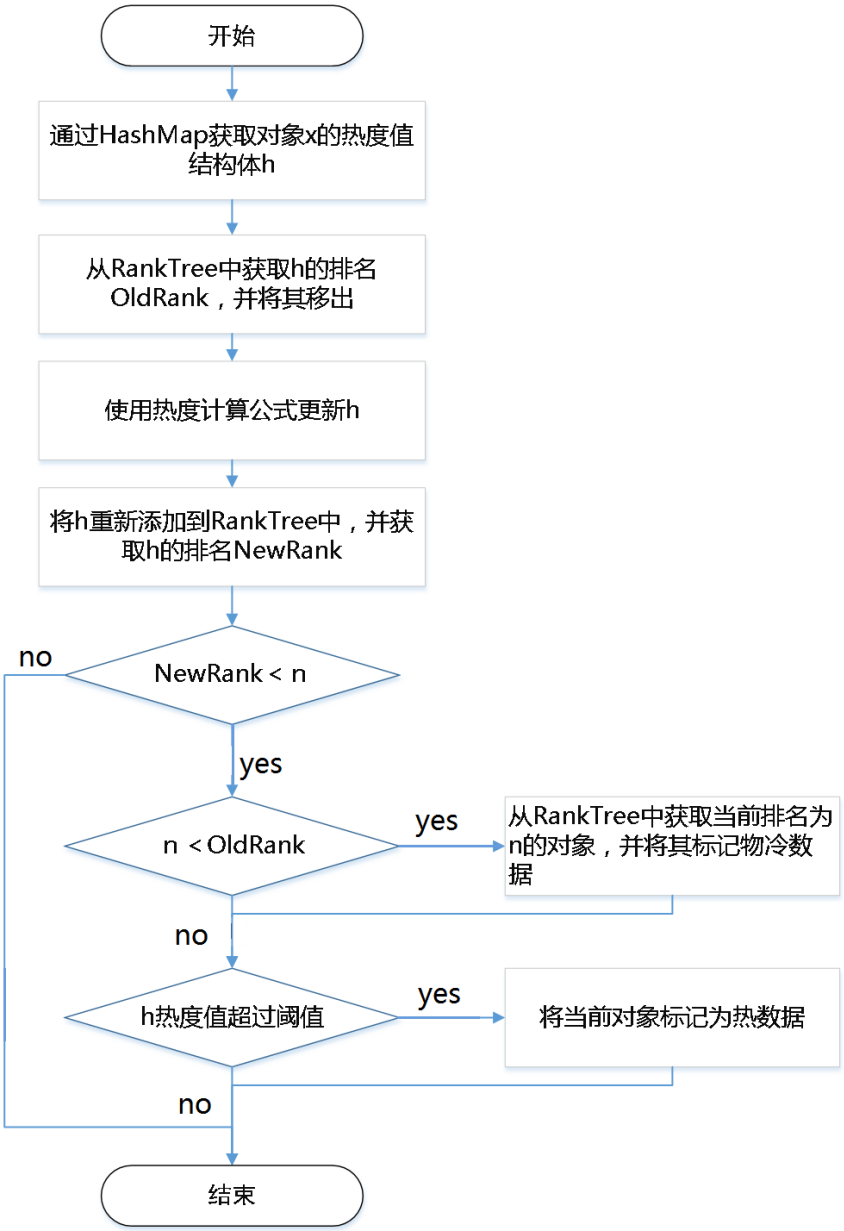

图 3.13 热度值更新流程

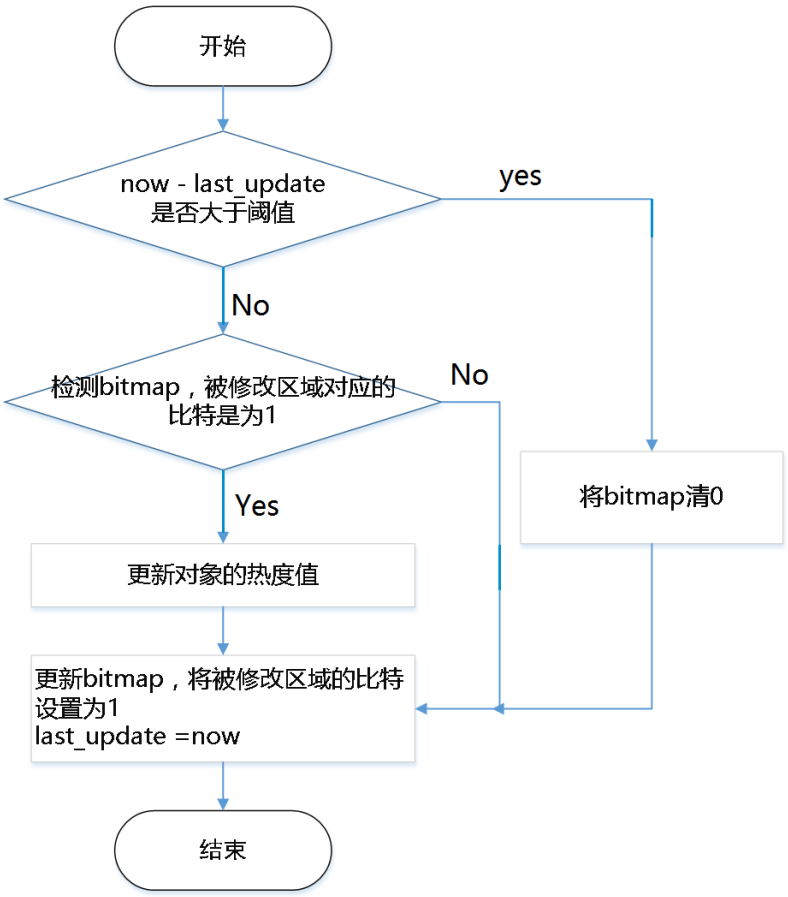

图 3.14 根据 bitmap 判断是否更新热度值

3.8 本章小结

本章详细叙述了热数据延迟写机制在工程实现中所采用的几个关键性技术，例如双日志结构，多指针的空间管理，事务的 pipeline 处理，异步快照，内存索引和缓存，以及热度的计算公式和实时排名等，这些技术确保了热数据延迟写机制的正确高效运行。下一章将对热数据延迟写机制进行相应的测试。

4 测试结果及分析

本章节针对 SSD 在 Ceph 下的两种使用场景（将 SSD 作为多块磁盘的日志或构建分布式 SSD 存储池）构建了两个测试环境，并在两个不同的测试环境下使用两个不同的测试工具 filebench[36-38]和 fio[39.40]进行测试。实验表明，对于 filebench 中 fileserver 负载，采用热数据延迟写机制后系统的性能得到了很大的提升；对于 fio 产生的随机写负载，采用热数据延迟写机制前后系统的性能相差不大。同时针对热数据延迟写机制对 OSD 恢复时间的影响进行了测试，测试表明该机制增加了 OSD 的故障恢复时间。

4.1 测试环境与配置

表 4.1 服务器配置参数

类别	配置
处理器	Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
网卡	千兆以太网卡
存储空间	内存: 12GB；固态盘: sata2 接口, 容量 250GB；磁盘: 2*1T
操作系统	Ubuntu 12.04.4 LTS
服务器数量	3
Ceph 版本	0.87
Qemu 版本	2.2.0

4.2 测试内容

4.2.1 测试一

使用两台服务器搭建一个 Ceph 集群，其中每个服务器中共有两个存储节点，

每个存储节点使用一块 1TB 的磁盘用来存储数据，同一台服务器的两个存储节点将日志放置到同一块 SSD 的不同分区。

在该环境下创建 6 个 100GB 大小的虚拟磁盘，每个磁盘的副本数为 2，并运行 6 个虚拟机分别对这 6 个虚拟磁盘进行读写，其中虚拟机中使用 filebench 工具产生 fileserver 测试负载。测试将对比采用热数据延迟写机制前后 Ceph 性能的差异。

总共有 3 组测试，其中两组使用了热数据延迟写机制，一组为原来的日志机制。在采用热数据延迟写机制的时，一组测试将 HotJournal 和 ColdJournal 分别设置为 12GB 和 4GB，另一组测试将 HotJournal 和 ColdJournal 分别设置为 16GB 和 4GB。测试将运行 800s，测试结果来自 filebench 自身的统计信息，以及 Ceph 自带的统计工具，下面是测试的结果。

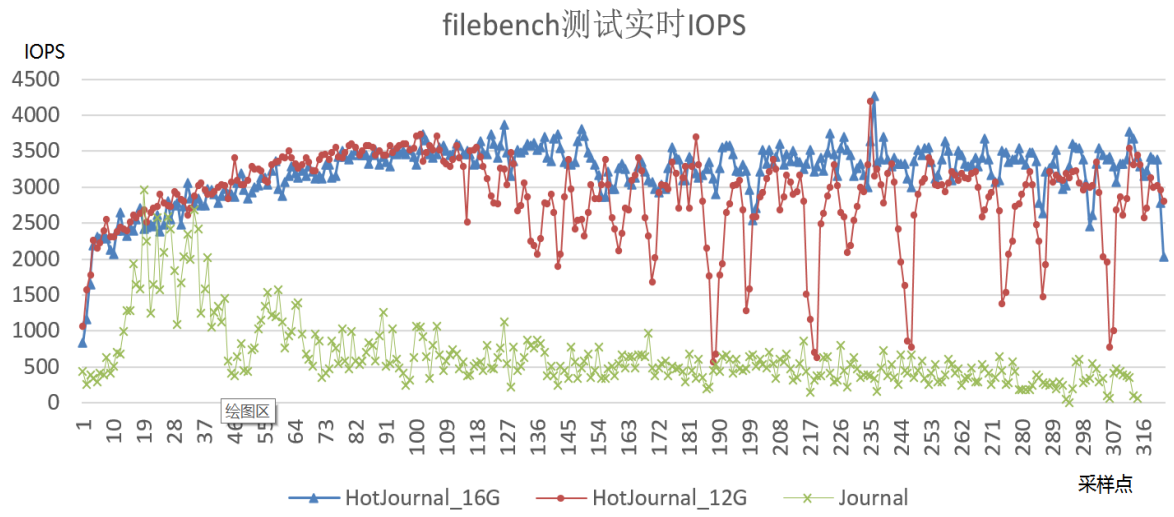

图 4.1 filebench 测试下 Ceph 的实时 IOPS

表 4.2 filebench 测试下虚拟机吞吐

虚拟机编号	vm1	vm2	vm3	vm4	vm5	vm6	总吞吐
12GB_HotJournal(MB/S)	23.5	21.8	23.0	21.2	20.4	22.5	132.4
16GB_HotJournal(MB/S)	25.2	23.9	25.1	24.6	23.4	23.5	145.7
Journal(MB/S)	6.5	5.8	6.9	7.4	6.7	6.6	39.9

对比采用热数据延迟写机制前后 Ceph 的性能，可以看出热数据延迟写机制能够大大提升系统的性能。这是因为 filebench 产生的 fileserver 负载有明显的热点，热数据延迟写机制能够区分冷热数据，并将热点数据的多次写操作合并为对文件系统的一次写操作，因而提升了系统的性能。

观察采用热数据延迟写机制时的 IOPS 曲线，可以发现 IOPS 在开始阶段持续上升，当 HotJournal 可用空间低于某个阈值而触发了空间回收作业时，IOPS 会下降。对比 HotJournal 大小为 12GB 和 16GB 下的 IOPS 曲线图，可以看出 16GB 下的 IOPS 曲线更加稳定，原因可能为大容量的 HotJournal 使得被回收空间中的有效数据比例更低，因而回收单位空间时需要更少的对 HotJournal 的读操作和对文件系统的写操作。

观察采用原有日志机制时 IOPS 的实时曲线，可以发现在开始阶段系统的 IOPS 很高，但随着数据量的增大系统的性能越来越低。借助使用 linux 自带磁盘监控工具 iostat 可以看到，此时磁盘的利用率达 90%~100%，而固态盘的利用率只有 20%~60%，说明磁盘成为性能的瓶颈。磁盘的机械构造，使其处理随机读写请求时需要耗费大量的时间在磁盘寻道上，这是磁盘成为性能瓶颈的重要原因。

4.2.2 测试二

使用两台服务器搭建一个 Ceph 集群，其中每个服务器中共有两个存储节点，每个存储节点使用一块 1TB 的磁盘用来存储数据，同一台服务器的两个存储节点将日志放置到同一块 SSD 的不同分区。

创建 6 个 100GB 大小的虚拟磁盘，磁盘副本数为 2，并运行 6 台虚拟机分别对它们进行读写，虚拟机使用 fio 产生负载，表 4.3 是 fio 的运行参数。图 4.2 是使用热数据延迟写机制前后的实时 IOPS。

可以看出，采用热数据延迟写机制前后系统的性能差别不大，它们的 IOPS 曲线基本相同。因为 fio 产生的随机写负载，没有明显的热点数据，因此热数据延迟写机制无法通过合并热点数据的写操作来提升性能。同时可以发现，随着数据量的

增加，系统的性能逐渐降低，这是因为磁盘寻道成为性能的瓶颈，并随着数据量的增加而显现出来。

表 4.3 fio 运行参数

bs	128kb
Thread	2
ioengine	libaio
Direct	1
iodepth	20
Rw	randwrite
runtime	800

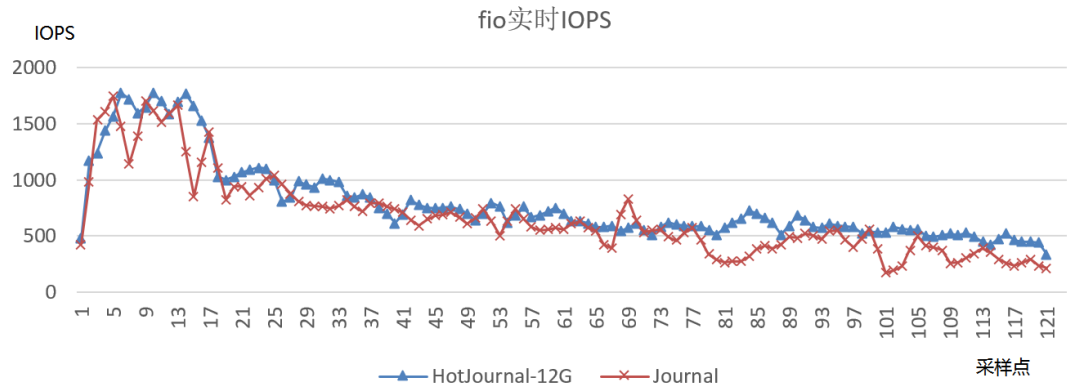

图 4.2 fio 测试下 Ceph 的实时 IOPS

4.2.3 测试三

使用三台服务器构建 Ceph 集群，共有三个节点，每个节点对应一块 SSD。该场景下，节点的日志和文件系统都位于同一块 SSD 的不同分区上。采用热数据延迟写机制时，节点的 HotJournal 和 ColdJournal 大小分别为 16GB 和 4GB。

测试将创建 9 块大小为 100GB 的虚拟磁盘，并运行 10 个虚拟机分别对其进行读写，虚拟机使用工具 filebench 产生 fileserver 负载，并运行 800s。图 4.3 是测试

得到的 Ceph 实时 IOPS，表 4.4 是 9 个虚拟机的测试结果。

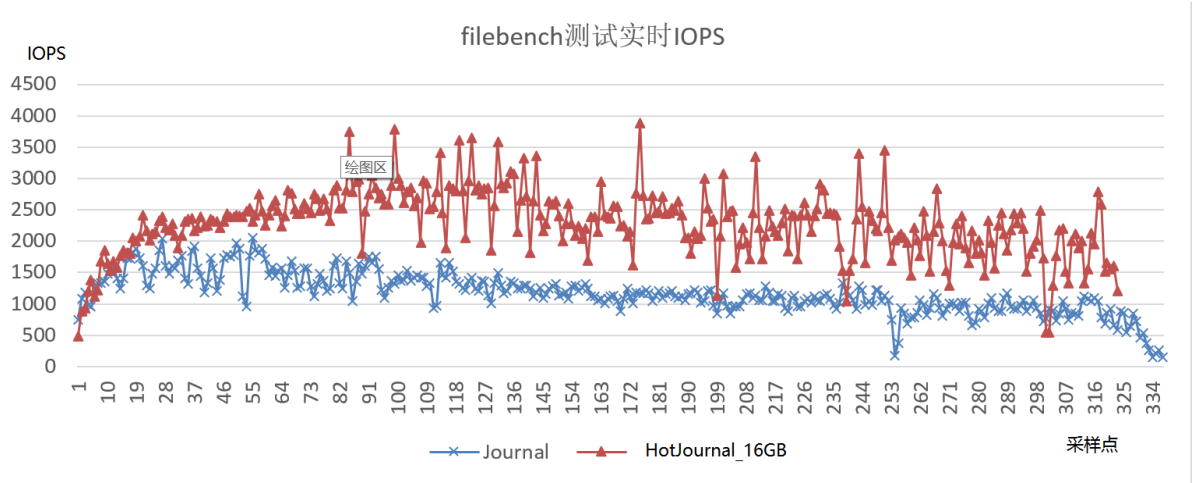

图 4.3 filebench 测试下 Ceph 的实时 IOPS

表 4.4 filebench 测试下虚拟机的吞吐

虚拟机编号	vm1	vm2	vm3	vm4	vm5	vm6	vm7	vm8	vm9	总吞吐
Journal(MB/S)	8.9	8.6	10.9	10.3	8.4	9.3	9.0	9.4	9.4	84.2
写延迟 Journal(MB/S)	14.1	15.2	13.5	14.5	14.8	14.2	15.2	13.7	14.0	129.2

对比改进前后的 IOPS 实时曲线可以发现，热数据延迟写机制也能够提升日志和文件系统在同一块 SSD 设备时系统的性能。说明热数据延迟写机制能够很好地合并对热点数据的写操作，从而提升系统的性能。

观察采用热数据延迟写机制时的 IOPS 变化可以发现，IOPS 在最开始阶段持续上升，一直上升到 3000~3500，持续一段时间后 IOPS 下降并保持在 2500 左右，这是因为 HotJournal 的空闲空间接近预先设定的阈值，触发后台线程回收 HotJournal 的空间，增加了额外的负载，因此性能下降。

4.2.4 测试四

集群环境同测试三，使用三块 SSD 构成一个 Ceph 集群，并创建 3 个 100GB 的虚拟磁盘，并运行 9 个虚拟机对虚拟磁盘性能进行测试，虚拟机使用 fio 产生

测试负载，fio 使用参数如表 4.3 所示。使用 Ceph –w 命令获得的实时 IOPS，结果如图 4-2 所示。

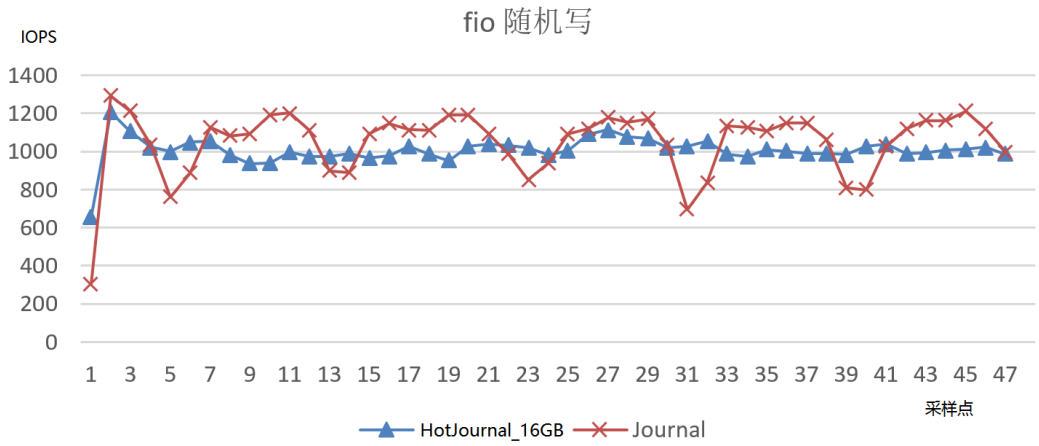

图 4.4 fio 测试下 Ceph 的实时 IOPS

可以看到改进前后性能相差不大，这是因为 fio 为随机读负载没有任何热点数据的存在，无法使用 HotJournal 进行进行改进。

4.2.5 测试五

使用 fllebench 产生数据并运行 120s，然后重启 OSD 并测试重启时事务恢复所需要的时间。采用热数据延迟写机制的 OSD 使用 16GB 的固态盘空间作为 HotJournal，使用 4GB 的固态盘空间作为 ColdJournal，并使用磁盘作为数据盘；用作对比测试的采用传统日志的 OSD 使用 4GB 的固态盘空间作为日志设备，使用磁盘作为数据盘。表 4.6 是两者恢复时间的对比。

表格 4.5

日志模式	恢复时间
热数据写推迟机制(16GB HotJournal)	16.0s
传统日志机制(4GB)	4.0s

filebench 产生数据并运行 120s 之后，在采用热数据延迟写机制的 OSD 中，其 HotJournal 占用了 7GB 的空间，而 C5oldJournal 则用至 3.82GB，并回收了 2.80GB 的空间，即 ColdJournal 占用的空间为 1.02GB，OSD 恢复的时间为 16.0s。用作对

比的 OSD 采用传统日志模式，其恢复时间则为 4s。

对比两组测试可知，采用热数据延迟写后，OSD 的故障恢复时间增长。利用日志进行故障恢复过程需要扫描 HotJournal 中所有的日志项，并将日志项头部的元数据提取出来建立内存索引，因此故障恢复过程中会产生大量的对 HotJournal 的随机读取操作；而且元数据并非连续存储，因此无法利用预读来提升性能；同时为了提升性能，日志的大部分空间都用作 HotJournal，使得元数据扫描的过程变得漫长。这些原因导致采用热数据延迟写机制后 OSD 的故障恢复时间被大大延长。

4.3 本章小结

本章在 Ceph 的两个部署场景下对热数据延迟写机制进行了多项性能测试对比，其中在工具 filebench 产生的 fileserver 负载下，系统的性能有很大的提升；而在 fio 产生的随机写负载下，系统的性能没有明显改善。由于热数据延迟写机制通过热度排名机制识别冷热数据，并采用延迟写的方式对热数据进行 I/O 合并，从而提升系统性能，在 fileserver 负载下热点数据明显，因此 I/O 所带来的性能提升比较明显；fio 产生的随机写负载热点数据不明显，因此 I/O 负载带来的性能提升也就不明显。这说明热数据延迟写机制能够正确运行，并能在热点明显的负载下显著提升系统的性能。同时也对热数据延迟写机制对 OSD 故障恢复时间的影响进行了测试，测试表明热数据延迟写机制延长了 OSD 的故障恢复时间，需要进一步改进。

5 总结与展望

日志是实现事务原子性的一种高效简洁的机制，在很多存储系统中得到了应用。但由于在更新数据前必须先更新日志文件，因此存在两次写入，严重影响了系统性能。为此，本文结合 Ceph 的架构特点以及当前日志机制优化措施的短处，提出了热数据延迟写机制，实现了对 Ceph 底层存储引擎的性能优化。

本文主要的研究工作如下。

（1）结合分布式存储系统 Ceph 的架构特点详细分析了 Ceph 底层存储引擎提供事务接口的必要性；

（2）利用热度计算公式和排序树实现了对象热度值的实时排名，并据此实现对象的冷热识别；

（3）采用内存索引将日志变为热数据的临时存储空间，从而实现热数据的延迟写；

（4）借鉴 f2fs 文件系统的实现原理，利用双日志结构分别记录冷热数据的更新历史，从而减轻了空间回收时读开销；

（5）采用快照和双日志结构的方式实现故障重启时的数据恢复。

（6）实验证明，在热数据明显的负载下，能够显著提升系统的性能；对于热数据不明显的负载，热数据延迟写机制也能够保证原有日志机制的性能。

目前，热数据延迟写机制还有很多地方没有完善，下面对这些不足之处加以说明，并指出可行的解决方案，便于日后完善。

（1）热数据延迟写机制实质上是以固态盘的空间优势来换取性能上的提升，日志空间越大，性能提升越明显，故障恢复所消耗的时间就越长，从而也就增加了集群断电重启所需要的时间。但同时，事务回放时只需要将元数据读入就能建立所有热数据的内存索引，而元数据占用的空间很少，因此可以将元数据单独存放到一个子日志中，故障恢复时一次读取操作就能加载大量的元数据到内存，从而大大降低恢复的时间。

（2）存储引擎没有持久化热对象哈希表，因而故障重启的时候需要重新根据负载计算对象的热度值来区分冷热数据。在热数据延迟写机制下，HotJournal 维护了相当一段时间内对象的更新记录，因而在存储系统对外提供服务前借助这些记录预先初始化各个对象的热度值，从而在后续执行用户请求时能够及时区分冷热数据。

致 谢

一弹指之间，3 年的研究生学习生活已经结束，回首细想，有很多值得感谢人和事。

首先要感谢的是我的父母，谢谢你们对我的关心和牵挂，在我心情阴霾和身体不适的时候对我的细心照顾，让我时时刻刻都能感受到亲情的温暖。感谢我的伯父伯母，每次回家总是做美味的饭菜给我吃，也谢谢你们对我的理解和信任。感谢我的小妹，正如那句“it is never too late to have a happy childhood”，你让我又感受到了童年的欢乐。再次谢谢我的家人。

感谢我的导师刘景宁教授，谢谢你一直以来在学业上对我的引导，在我找不到方向的时候帮忙想办法，给了我很多宝贵的意见。同时为非常感谢你对我的理解和包容，桀骜不驯的我总是任性地按着自己的想法做事，是你的包容让我能够自由地发挥所长。感谢我的指导老师童薇老师，谢谢你在学业上对我的指导，每次讨论问题都能指出想法的不足之处，让我醍醐灌顶。也感谢你对我的鼓励和帮助，让我在学业上陷入困境的时候仍然乐观地努力前行。

感谢验室其他老师，谢谢你们创造了这么好的实验室学习环境，让我度过了开心的 3 年研究生学习生活。

感谢李宁、吴运翔和鲁振伟等师兄对我的帮助，和你们的交流之中，我受益匪浅。感谢刘柯南、伊武哲、吕力、刘景超、傅瑶以及余晨晔等同学，在忙碌的学习之中，你们给了我很多快乐，谢谢你们的陪伴。感谢我的室友刘景超、余晨晔和徐思维，以及对面宿舍的周双鹏、吕力和胡畔，很开心在闲暇时候与你们玩卡牌、夜聊以及分享各种有意思的事情，正是因为你们我的宿舍生活才如此快乐精彩。

参考文献

[1] 张文江, 吴庆波. Linux 日志文件系统研究.计算机工程与应用, 2006, 42(9): 50-52

[2] Tweedie, S. C. Journaling the Linux ext2fs filesystem. in: Proceeding of the 4th Annual Linux Expo, Southern California,1998, 1-8

[3] Dave H., James L., and Michael M.. File System Design for an NFS File Server Appliance. in: Proceedings of the USENIX Winter Technical Conference, San Fr ancisco, California, 1994, 1-6

[4] Gregory R. G., Yale N.P. Metadata Update Performance in File Systems. in: Proc eedings of the 1st Symposium on Operating Systems Design and Implementation, Monterey, California, 1994, 1-12

[5] McKusick, M. K., Ganger, G. R. Soft Updates: A Technique for Eliminating Mos t Synchronous Writes in the Fast Filesystem. in: Proceedings of the USENIX An nual Technical Conference, Monterey, CA, 1999, 1-17

[6] Feiyi Wang, Mark Nelson, Sarp Oral, et al. Performance and scalability evaluatio n of the Ceph parallel file system. in: Parallel Data Storage Workshop. 2013:14-1 9

[7] 王建宇. OpenStack 平台与 Ceph 统一存储的集成. 中国管理信息化, 2016(4)

[8] Johanes J, Johari M F, Khalid M, et al. Comparison of Various Virtual Machine Disk Images Performance on GlusterFS and Ceph Rados Block Devices. in: Inter national Conference on Informatics & Applications. 2014

[9] Weil S A, Brandt S A, Miller E L, et al. Ceph: A scalable, high-performance di stributed file system. in: Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006: 307-320

[10] Gudu D, Hardt M, Streit A. Evaluating the Performance and Scalability of the C eph Distributed Storage System. in: Big Data (Big Data), 2014 IEEE Internationa l Conference on. IEEE, 2015:177-182

[11] Donvito G, Marzulli G, Diacono D. Testing of several distributed file-systems (H DFS, Ceph and GlusterFS) for supporting the HEP experiments analysis. in: Jour nal of Physics: Conference Series. IOP Publishing, 2014, 513(4): 042014

[12] Verma R, Mendez A A, Park S, et al. Failure-atomic updates of application data

in a linux file system. in: Proceedings of the 13th USENIX Conference on File and Storage Technologies. USENIX Association, 2015: 203-211

[13] Hatzieleftheriou A, Anastasiadis S. Host-side filesystem journaling for durable sha red storage. in: 13th USENIX Conference on File and Storage Technologies (FAS T 15). 2015: 59-66

[14] Narayanan D, Thereska E, Donnelly A, et al. Migrating server storage to SSDs: analysis of tradeoffs. in: Proceedings of the 4th ACM European conference on C omputer systems. ACM, 2009: 145-158

[15] Marsh B, Douglis F, Krishnan P. Flash memory file caching for mobile computer s. in: System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii Internati onal Conference on. IEEE, 1994, 1: 451-460

[16] Kgil T, Mudge T. FlashCache: a NAND flash memory file cache for low power web servers. in: Proceedings of the 2006 international conference on Compilers, a rchitecture and synthesis for embedded systems. ACM, 2006: 103-112

[17] 刘彬. 基于 Nutanix 下台的云媒资探索. 电视技术, 2015(6):65-67

[18] Cully B, Wires J, Meyer D, et al. Strata: scalable high-performance storage on vi rtualized non-volatile memory. in: Proceedings of the 12th USENIX conference o n File and Storage Technologies. USENIX Association, 2014: 17-31

[19] Srinivasan M, Callaghan M. FlashCache. 2010

[20] Greenan K M, Miller E L. PRIMS: Making NVRAM suitable for extremely relia ble storage. in: Proceedings of the Third Workshop on Hot Topics in System De pendability (HotDep'07). 2007

[21] Lee E, Bahn H, Noh S H. Unioning of the buffer cache and journaling layers w ith non-volatile memory. in: FAST. 2013: 73-80

[22] Fusionio I. Fusion-io Delivers Open APIs for Flash-Aware Application Acceleratio n[J]. Fusion-io, Inc

[23] N. Talagala. Atomic writes accelerate MySQL performance, Oct. 2011 http://www. fusionio.com/blog/atomic-writes-accelerate-mysql-performance/.

[24] L. Marmol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. R amsundar, and S. Ganesan. NVMKV: A scalable and lightweight flash aware key-value store. In HotStorage, June 2014

[25] Condit J, Nightingale E B, Frost C, et al. Better I/O through byte-addressable, persistent memory. in: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009: 133-146

[26] Byan S, Lentini J, Madan A, et al. Mercury: Host-side flash caching for the data center. in: Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on. IEEE, 2012: 1-12

[27] Koller R, Marmol L, Rangaswami R, et al. Write policies for host-side flash caches. in: FAST. 2013: 45-58

[28] Bhagwat D, Patil M, Ostrowski M, et al. A practical implementation of clustered fault tolerant write acceleration in a virtualized environment. in: 13th USENIX Conference on File and Storage Technologies (FAST 15). 2015: 287-300

[29] Cao M, Tso T Y, Pulavarty B, et al. State of the art: Where we are with the ext3 filesystem. in: Proceedings of the Ottawa Linux Symposium (OLS). 2005: 69-96

[30] Sovani K. Linux: The journaling block device. Retrieved July, 2006, 15: 2007.

[31] Park C, Talawar P, Won D, et al. A high performance controller for NAND flash-based solid state disk (NSSD). in: Non-Volatile Semiconductor Memory Workshop, 2006. IEEE NVSMW 2006. 21st. IEEE, 2006: 17-20

[32] Chen S W. Solid-state disk: U.S. Patent Application 12/068,757. 2008-2-11

[33] Wong H S P, Raoux S, Kim S B, et al. Phase change memory. Proceedings of the IEEE, 2010, 98(12): 2201-2227

[34] Lai S. Current status of the phase change memory and its future. in: Electron Devices Meeting, 2003. IEDM'03 Technical Digest. IEEE International. IEEE, 2003: 10.1. 1-10.1. 4

[35] 杨宗. Flashcache 的实现原理与优化研究. 华中科技大学, 2013

[36] McDougall R, Mauro J. Filebench tutoria. Sun Microsystems, 2004

[37] Yongju Song, Junghoon Kim, Dong Hyun Kang, et al. Analyses of the Effect of System Environment on Filebench Benchmark. 2016, 43(4):411-418

[38] McDougall R. Filebench: Application level file system benchmark. 2014

[39] fio http://linux.die.net/man/1/fio

[40] 用 fio 测试磁盘性能 http://wsgzao.github.io/post/fio/