

引文格式:

李润辉, 古亮, 喻之斌. PheonixLSM: 高性能低空间开销的分布式键值存储本地引擎 [J]. 集成技术, 2022, 11(2): 28-40.
Li RH, Gu L, Yu ZB. PheonixLSM: local key-value engine for distributed key-value storage with high performance and low spatial consumption [J]. Journal of Integration Technology, 2022, 11(2): 28-40.

PheonixLSM: 高性能低空间开销的分布式键值存储本地引擎

李润辉^{1,2} 古 亮² 喻之斌^{1*}

¹(中国科学院深圳先进技术研究院 深圳 518055)

²(深信服科技股份有限公司 深圳 518071)

摘 要 分布式键值存储将数据复制到多个存储服务器的本地引擎中, 并通过一致性协议保证各副本数据的一致性。其中, 以日志结构合并树为核心数据结构的实现方式最为常见。然而, 面向通用业务模式设计的日志结构合并树, 并不适合一致性逻辑的特殊业务模式, 会引发增删改性能的降低, 并在全量修复过程中造成空间放大。针对上述问题, 该文提出了一种新型本地引擎 PheonixLSM, 通过增加增删改操作和回刷操作的约束, 消除了分布式键值存储增删改流程中的双写问题, 提升了引擎性能。通过重构日志结构合并树底层的 SST 文件布局, 支持删除实时回收空间, 消除了全量修复时的额外空间放大。实验结果显示, 与原生本地引擎相比, 使用 PheonixLSM 的分布式键值存储系统, 增删改性能提升 90.7%, 全量修复的空间放大从 65.6% 降至 6.4%, 并减少了 72.3% 的修复时间。

关键词 分布式系统; 键值存储; 一致性协议; 空间放大

中图分类号 TP 391 **文献标志码** A **doi**: 10.12146/j.issn.2095-3135.20210422001

PheonixLSM: Local Key-Value Engine for Distributed Key-Value Storage with High Performance and Low Spatial Consumption

LI Runhui^{1,2} GU Liang² YU Zhibin^{1*}

¹(Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China)

²(Sangfor Technologies Inc., Shenzhen 518071, China)

*Corresponding Author: zb.yu@siat.ac.cn

Abstract Distributed key-value storage is a key component in the distributed storage system, which replicates the key-value pairs to local engines in different storage servers and uses the consensus algorithm

收稿日期: 2021-04-22 修回日期: 2021-05-26

作者简介: 李润辉, 博士, 研究方向为键值存储、分布式系统、纠删码和差量压缩; 古亮, 博士, 研究方向为安全攻防、漏洞/文件检测、威胁情报、云计算、大数据及 AI 平台构建和优化; 喻之斌(通讯作者), 博士, 研究员, 研究方向为计算机体系结构、计算系统虚拟化、云计算及大数据分析平台构建与优化, E-mail: zb.yu@siat.ac.cn。

to keep replicas consistent. The log-structured merge tree based local key-value engine is the most popular storage algorithm that designed for general purpose. However, the original LSM-tree structure is not suitable for the specific workload of the upper-layer consensus logic. It usually causes performance loss in the write operations and extra spatial amplification during full-node repair. To solve this problem, a local engine named PheonixLSM is designed for distributed key-value storage. PheonixLSM boosts performance by eliminating the double-sync problem for write operations. It also reorganizes the SST file layout to eliminate extra write amplification during full-node repair. Experimental results showed that, compared with distributed key-value storage using default local engine, that PheonixLSM can achieve up to 90.7% write performance gain and reduce the extra write amplification from 65.6% to 6.4%, and the repair time also can be reduced by 72.3%.

Keywords distributed system; key-value storage; consensus algorithm; spatial amplification

1 引言

近年来,随着移动互联网、人工智能及物联网等新兴技术的迅速发展,对海量数据的存储需求也随之变大,数据规模远超传统业务。如,在安防领域,基于人工智能的人脸识别技术识别出视频流中的人像,并以小文件/对象的形式保存在底层的文件/对象存储系统中,从而产生每秒创建数千个文件/对象的业务流量峰值,远超传统的监控视频存储,这对存储系统的元数据访问性能和承载规模提出了严峻的考验。在存储系统中,元数据的访问量远超数据,且提供了并发控制等复杂逻辑,因此,元数据的访问性能和承载规模也决定了存储系统的访问性能和承载规模的上限。

键值存储,是承载文件/对象存储元数据等关键信息的存储系统核心组件,对其访问性能和承载规模提升的研究具有重要意义。

分布式键值存储,具有更高的性能、更好的容错性和扩展性,因而被广泛使用。其将数据复制到多个存储服务器的本地键值存储引擎(简称“本地引擎”)上,并通过一致性协议来保证不同副本的一致性。在业界被广泛应用,且满足

强一致性的一致性协议(如 Paxos^[1]、RAFT^[2]),均基于日志复制。这类一致性协议(简称“一致性协议”)在处理写请求时,各副本首先持久化请求到本地的日志中,当足够多的副本写日志成功后,才将数据更新到存储于本地引擎的状态机中。基于日志结构合并(Log-Structured Merge, LSM)树的本地引擎(如 LevelDB^[3]、RocksDB^[4]),能够将小 I/O 转化为顺序追加写操作,对于底层硬件更为友好,因而被广泛使用。使用 RAFT 等强一致性协议,及 RocksDB 等基于 LSM 树的本地键值存储作为本地引擎,成为了多个主流开源分布式键值存储系统(如 TiKV^[5]、Cassandra^[6])的选择。然而,为通用业务模型设计的本地引擎,其没有充分适配一致性协议的访问模式,所以还存在较大的优化空间。本文在分布式键值存储的实践中,发现并归纳出了如下问题:

(1) 基于 LSM 树的本地引擎为了提供容错性,会将请求先写入在本地持久化的预写式日志(Write-ahead log, WAL)中,再更新内存数据。当系统发生异常重启时,本地引擎可以通过 WAL 修复丢失的内存数据。然而,一致性协议的日志,也具备通过回放修复状态机至最新状态

的能力。此时,修复同一份数据,需要两次持久化的保护,这一双写会带来高达 47.5% 的性能损耗(见第 5.3 节)。能否通过改造底层引擎,配合一致性协议,消除双写,来达到提升分布式键值存储性能的目的呢?

(2) 一致性协议的修复方法有两种:通过重放日志进行增量修复,以及复制状态机进行全量修复。其中,全量修复需要删除旧状态机,并从其他节点复制新状态机。LSM 树的删除操作异步回收空间,导致新旧状态机数据长期并存,直到旧状态机数据全部被异步删除,这样会造成存储空间浪费严重。在实验中,全量修复会带来超过 60% 的空间浪费(见第 5.3 节)。为维持系统的正常运行,存储服务器必须在物理设备尚余大量存储空间的情况下,就停止服务,以防全量修复耗尽存储空间,预留空间造成存储容量的严重浪费。能否通过对底层引擎的改造,使得全量修复过程能够实时回收空间,消除空间放大,从而避免预留过多的空间呢?

为了解决以上两个问题,本文提出了基于 LSM 树改进的 PheonixLSM 本地引擎。具体贡献如下:(1)通过增加 LSM 树的增删改操作和内存数据回刷顺序的制约,并提供新的查询接口,使得一致性协议日志能够同时起到 WAL 的作用,恢复本地引擎的内存数据,从而避免日志和 WAL 的双写,同时可将上层分布式键值存储吞吐提高 90% 以上,单位计算资源吞吐提高 20% 以上。(2)通过改造 LSM 树的底层数据组织形式和范围删除逻辑,使得在对旧状态机的删除操作中,能够实时回收空间,使全量修复过程中的空间放大从 65.6% 降低到 6.4%,并节省 72.3% 的修复时间。

本文后续内容的组织如下:第 2 节列举相关工作;第 3 节介绍技术背景;第 4 节说明 PheonixLSM 的设计;第 5 节说明 PheonixLSM 的原型实现和部署,实验结果以及分析;第 6 节

对本工作进行了讨论分析;第 7 节总结全文。

2 相关工作

本节首先列举了常见的本地键值存储系统以及核心数据结构,然后对 LSM 树的优化改造方向进行了归纳。

本地引擎有多种形态,有将数据存储于内存的 Redis^[7]、Memcached^[8]和 MemC3^[9]等,也有将数据持久化于本地存储(机械硬盘或固态硬盘)的 LevelDB^[3]和 RocksDB^[4]等。B+ 树^[10]和 LSM 树是两种常见的持久化引擎的核心数据结构。B+ 树的数据全量存储在首尾相连的叶子节点中,具有良好的读和范围查找性能,MySQL 的本地引擎 MyISAM^[11]、InnoDB^[12]等都基于 B+ 树。然而,B+ 树不适用于随机插入操作,其会产生大量的随机小 I/O。相比之下,LSM 树能将随机小 I/O 转为连续大 I/O,写性能较好,且能较好地平衡读放大、写放大和空间放大,近年来得到了广泛应用。LSM 树最早应用在 Google 推出的 LevelDB^[3]中,被用作核心数据结构,Facebook 此后推出了 RocksDB^[4],该结构在 LevelDB 的基础上针对固态盘做了许多优化。

近年来,基于 LSM 树的存储引擎被广泛应用在 Facebook^[13-14]等企业的诸多场景中。关于 LSM 树的写放大问题:PebblesDB^[15]放宽了 LSM 树的约束,使 L_1 及更低层中,SST 文件的 key 范围可以适度重合,从而降低写放大;WiscKey^[16]采用了键值分离的策略,解决大 value 带来的索引效率低,以及写放大严重的问题。关于 LSM 树如何适配新硬件:增加 NVMeSSD 作为缓存层提升性能^[17],使用 FPGA 卸载合并任务,减少对读写业务的影响^[18]等。关于 LSM 树读性能:采用动态布隆过滤器减少磁盘 I/O^[19],针对 LSM 树的数据结构开发新缓存策略^[20],优化范围查询中的读放大^[21],调度后台回刷合并工

作以减少对上层业务的干扰, 降低尾时延^[22]等。

3 技术背景

本节首先介绍分布式键值存储的主要组成部分, 然后介绍 LSM 树的基础, 最后以 RAFT 为例介绍基于日志复制的一致性协议。

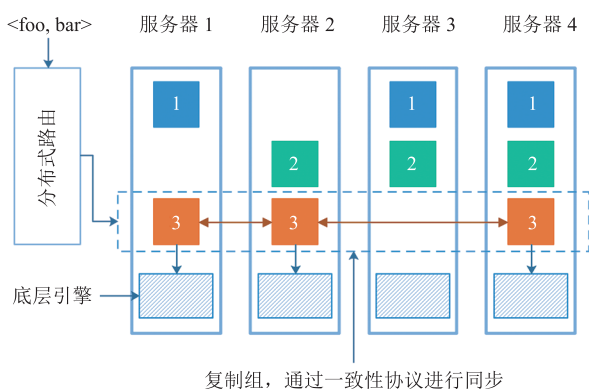


图1 分布式键值存储架构及请求处理流程

Fig. 1 Architecture of distributed key-value storage and the request processing flow

分布式键值存储将键值对存储在通过网络互相连接的存储服务器上, 从系统架构上分为3个主要组件: 分布式路由、一致性协议和本地引擎。

图1是一个典型的分布式键值存储的架构及

请求处理流程。分布式路由模块将 key 空间映射到多个复制组, 任意 key 都会被映射到唯一的复制组。如, 当写入键值对<foo, bar>时, 分布式路由模块确定 foo 所属的复制组, 以及该复制组节点所在的服务器; 之后, 对应的复制组通过一致性协议, 将键值对<foo, bar>写入各复制组节点所在的服务器的本地引擎中。查询操作也通过分布式路由模块确定 key 所属的复制组, 之后再对应复制组读取数据。

3.1 LSM 树的基础

LSM 树的结构及键值对生命周期如图2所示, 其主要由3部分组成: 内存数据结构 Memtables, 以及持久化的 WAL 和 SST 文件。SST 文件被组织成不同的层 (L_0 层、 L_1 层等)。

写请求首先持久化到 WAL 中, 之后键值对被插入全局唯一的可读写 Memtable 中, 请求完成。当可读写 Memtable 中的数据量超过一个阈值时, 其转为只读, 后续请求的键值对被写入新的可读写 Memtable 中。只读 Memtables 中的键值对会在后台的回刷和合并操作中逐层下沉至底层。

合并操作选择 L_{i-1} ($i \geq 1$) 层的某些 SST 文件, 以及 L_i 层与该 SST 文件的 key 区间有重合的 SST 文件。合并后的键值对, 将按 key 的字典序写入新的 L_i 层的 SST 文件中, 当 L_{i-1} 及 L_i 层

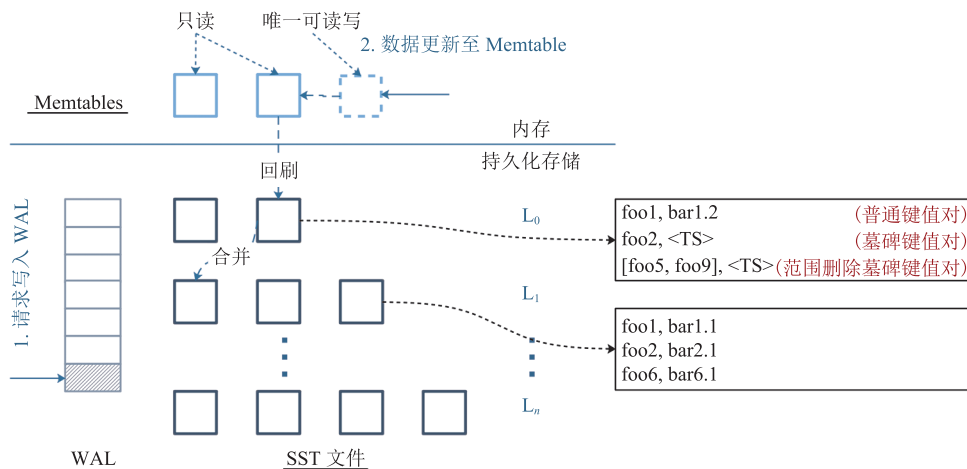


图2 LSM 树以及键值对的生命周期

Fig. 2 LSM-tree and lifecycle of key-value pairs

中都有某个 key 的键值对时,在合并操作后只有最新的键值对会被保留,SST 文件视图更新,旧 SST 文件被删除。

查询操作会按写入从新至旧的顺序依次查找 Memtables 和 L_0 中的各 SST 文件,然后从 L_1 开始逐层向下查找。当查找到对应 key 的键值对时,查询操作完成。若查询至底层时仍没有与之对应的键值对时,则返回 key 不存在。

LSM 树的删除请求和写请求流程相同,但是删除请求插入的是一条存储特殊 value 的墓碑键值对。当查询操作读到墓碑键值对时,立即返回 key 不存在(如图 2 中的 foo2)。范围删除采用同样的方法,插入一个特殊的范围删除墓碑键值对,当查询的 key 在该墓碑标记的范围内时,则返回 key 不存在(如图 2 所示,当查询 foo6 时,由于其处于范围删除墓碑键值对 [foo5, foo9] 的范围内,因此查询也会返回 key 不存在)。

3.2 基于日志复制的一致性协议

一致性协议为复制组提供了强一致性保证,使复制组即使在故障场景下,依然能够对外提供一致的状态机视图。本文以 RAFT 协议为例对一致性协议的基本原理作简要说明^①。

RAFT 协议定义了一个复制组中若干节点的

交互行为。在稳定状态下,复制组中有一个主节点和若干个从节点。每个节点中有一致性模块、日志和状态机 3 个主要组件。状态机是一个抽象概念,其在分布式键值存储中是本地引擎的键值对集合。

图 3 为 RAFT 协议处理增删改操作的流程。请求首先被发送至主节点(步骤 1),主节点的一致性模块为其分配一个系统生命周期递增的日志索引,并将请求转发至各从节点(步骤 2)。主节点和各从节点将请求持久化到日志中,并返回主节点的一致性模块日志写入成功(步骤 3)。当大部分节点返回成功后,主节点会将请求应用到状态机上(步骤 4),并返回请求处理成功(步骤 5)。主节点在后续和从节点的交互中,指示从节点异步应用对应日志(步骤 6)。

RAFT 协议通过保持各复制组中节点日志的一致性,达到各节点状态机保持一致的目的。通过重放相同的日志,各复制组得到相同的状态机。在实际系统中,状态机会定时进行持久化,保存为一个状态机快照,即快照操作。当进行快照操作时,需要记录快照索引,即状态机最新应用的日志索引。当故障重启时,一致性模块加载状态机快照,并顺序重放索引大于快照索引的日

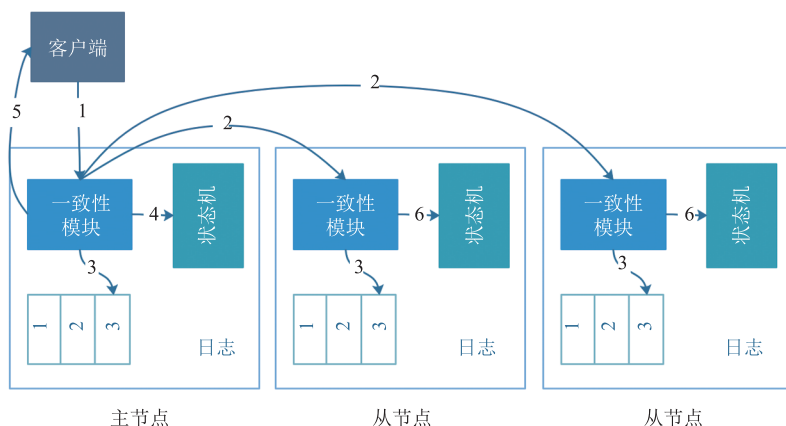


图 3 RAFT 协议处理增删改操作的流程

Fig. 3 Work flow of RAFT consensus algorithm

注^①: 本文虽然以 RAFT 作为一致性协议的代表进行说明,但本文中的技术同样适用于其他的基于日志复制的一致性协议。

志, 修复出最新的状态机, 这一过程叫作增量修复。使用快照和大于快照索引的日志, 便可修复出最新的状态机, 所以小于等于最新快照索引的日志均可以被安全截断。

当节点长期离线后再上线, 或迁移到新的存储服务器时, 主节点的日志可能无法满足其修复(或新建)状态机的需求。如, 某节点最新应用的日志索引为 100, 然而主节点维护的最旧日志索引为 110。那么, 必须先从主节点同步其状态机快照至本地, 然后从主节点同步并顺序重放索引大于快照索引的日志来修复状态机。这一过程叫作全量修复。

此外, 分布式键值存储中, 状态机已经持久化存储在本地引擎中, 所以快照操作并不需要额外持久化状态机, 只需要记录快照索引即可, 这种快照操作称作隐式快照。

4 系统设计

本节将对 PheonixLSM 的系统设计进行介绍。针对前述的日志和 WAL 双写, 以及全量修复时空间放大两个问题, 提出双写消除和过期数据实时清理两项优化。

4.1 双写消除

本小节说明如何使用 RAFT 日志进行状态机的同步, 同时在本地引擎中, 起到重建由进程重启导致丢失 Memtables 的作用。在满足该前提的情况下, PheonixLSM 不需要 WAL 保护, 所以可通过关闭 WAL 消除双写。

关闭 WAL 之后, 本地引擎由于失去保护, 重启会导致所有的 Memtables 丢失且无法重建, 只留下 SST 文件。本文归纳出使用日志修复 Memtables 的充分条件: 在任何时刻, 已持久化的 SST 文件都能构成一个状态机快照。在满足该充分条件的情况下, 关掉 WAL 的本地引擎在任意时刻重启后都存储一个状态机快照。根据一致

性协议的性质可知, 只要大于该快照索引的日志没有被删除, 那么在该快照的基础上, 就可以通过回放日志恢复状态机。

为了关闭 WAL, 需要解决以下两个问题:

(1) 如何保证在任意时刻, 已持久化的 SST 文件都能构成一个状态机快照; (2) 在满足(1)的情况下, 如何获得上述状态机快照的快照索引。从该快照索引对应的下一条 RAFT Log 开始回放, 就能够重构出状态机。

为解决上述问题, PheonixLSM 改造了增删改请求的应用流程。原来的应用流程: 将每个增删改请求产生的键值对提交到本地引擎; 改造后的应用流程: 除增删改请求产生的键值对之外, 还需同时写入一个特殊的哨兵键值对。哨兵键值对在一个复制组中有且仅有一个, 其 value 是当前最新被应用的 RAFT 日志索引, 能够确定重启时本地引擎中状态机的快照索引。图 4 为 PheonixLSM 中哨兵键值对插入 Memtable 并持久化至 SST 文件的流程, 由于关闭了 WAL, 插入 Memtable 时只需要内存操作。

若仅依靠哨兵键值对, 则无法保证重启后已持久化的 SST 文件能够构成一个状态机快照。所以 PheonixLSM 还对增删改和回刷操作做了以下约束:

(1) 一个请求的所有键值对(包括普通键值对、墓碑键值对和哨兵键值对), 必须更新至同一 Memtable 中;

(2) Memtables 必须按照其转为只读的时间顺序顺次进行回刷。

为了使 PheonixLSM 能够指导状态机日志的截断操作, 新增了 GetPersisted 接口, 其可跨过 Memtables, 直接从 L_0 开始查找已持久化的键值对。通过 GetPersisted 接口查找哨兵键值对, 返回的结果即是 SST 文件构成的状态机快照的快照索引。如图 4 所示, GetPersisted 接口查找到的哨兵键值对的值为 700。这一返回结果表明,

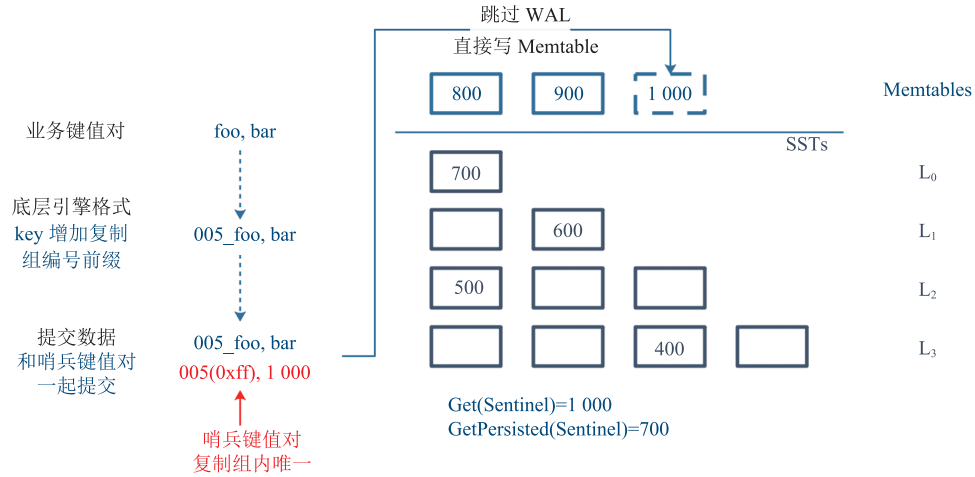


图 4 PhoenixLSM 中哨兵键值对插入 Memtable 并持久化至 SST 文件的流程

Fig. 4 Sentinel key-value pair inserting into Memtable and persisted to SST files

在没有新回刷发生的情况下发生故障重启, 那么本地引擎中存储的是快照索引为 700 的状态机快照。因此, 日志索引小于等于 700 的日志, 此时都可以被安全地删除。

本文通过实验证明了上述双写消除方案的有效性。假设本地引擎采取了前述的写操作和回刷操作约束, 使用 GetPersisted 接口获得哨兵值为 idx , 可以得出下述引理和定理。

引理 1 所有满足日志索引 $index \leq idx$ 的请求数据都已经持久化, 所有日志索引 $index > idx$ 的请求数据重启前在 Memtables 中, 没有回刷, 已经在重启中丢失。

证明: 首先使用反证法证明 $index \leq idx$ 的所有请求数据都已持久化。假如存在某日志索引 $index' < idx$ 的请求没有持久化, 那么, 日志 $index'$ 和 idx 的请求数据一定写入了不同的 Memtables 中, 否则会被同时持久化^②; 假设日志 $index'$ 和 idx 的请求数据分别被写入了 Memtables M_i 和 M_j , 那么由于 $index' < idx$, 一致性协议保证请求

数据按日志索引从小到大的顺序写入本地引擎, 因此, 日志 $index'$ 的数据会早于 idx 的数据被写入。由于任何时刻, 本地引擎都只有唯一的可读写 Memtable, 那么 M_i 早于 M_j 转为只读。索引日志 $index'$ 的请求数据没有持久化, 而 idx 的已被持久化, 说明 M_i 晚于 M_j 回刷, 违反了前述的回刷顺序约束。因此, 假设不成立, 所有日志索引 $index \leq idx$ 的请求数据都已持久化。

类似地, 也可以证明所有日志索引 $index > idx$ 的日志尚没有持久化。

定理 1 在满足写和回刷约束的前提下, 如果发生重启, 重启后的本地引擎中存储的是快照日志索引为 idx 的状态机快照。

证明: 根据引理 1, 当发生故障重启时, 所有日志索引 $index \leq idx$ 的请求数据均已被持久化, 而 $index > idx$ 的请求数据在重启前, 只是存储在 Memtables 中没有回刷, 在重启过程中已经丢失。按照一致性协议的定义, 所有日志索引 $index \leq idx$ 的请求数据, 在本地引擎中, 已经全

注^②: Memtable 回刷的原子性: Memtable 持久化成一个 L₀ 层 SST 文件的过程并不是原子的, 此时断电可能导致只有 Memtable 中部分的数据刷入该 SST 中。然而, Memtable 回刷的过程是由 (1) Memtable 数据写入 SST 文件, (2) 写入新的 LSM 树视图文件 (各 SST 文件的文件名、key 区间, 以及 SST 文件的层次关系等信息), (3) 替换旧 LSM 树视图文件 3 个步骤完成的。上述断电发生在步骤 (1), 重启之后, LSM 树的视图未被更新, 被写入一半的 SST 文件不在旧视图中, 会在初始化过程中被直接删除。而步骤 (3) 的替换视图文件是原子操作, 因此, 回刷操作中发生故障, 一个 Memtable 中的数据, 或者全部丢失, 或者全部已经持久化, 显示在新的视图文件中, 不会出现一个 Memtable 部分数据持久化, 部分数据丢失的场景。

部被顺序持久化, 并且没有多余的请求数据被写入。所以, 本地引擎中存储的是快照索引为 idx 的状态机快照。

4.2 过期数据实时清理

删除旧状态机是分布式键值存储的常见操作。节点长期离线后再上线触发全量修复, 以及负载均衡过程中, 复制组节点迁入新服务器时, 都需要先删除旧状态机, 以防污染新状态机数据。为便于操作, 分布式键值存储通常将不同复制组的 key 转换为不同前缀的底层格式。在本文的系统实现中, 业务逻辑的 key 都使用其所属的复制组编号作为前缀, 然后再存入本地引擎(如图 4 所示)。这种转换使得相同复制组中的 key 在底层引擎中具有相同的前缀, 存储在连续空间中, 可以通过范围删除清理某复制组的状态机。

原生本地引擎在删除旧状态机时, 插入了一个表示范围删除的墓碑键值对, 然后, 将从主节点处复制来的新状态机快照插入。如图 5 所示, 由于墓碑键值对通过逐层合并至最低层才能完全回收空间, 所以在新状态机插入后的较长时间内, 新旧状态机数据在本地引擎中共存, 将会造成存储空间的浪费。PheonixLSM 针对这一问题, 提出将各复制组的数据大致隔离在不同的 SST 文件中, 从而使得删除状态机的操作可以通过同步删除相关的 SST 文件完成, 实时释放空间。

PheonixLSM 通过改造 LSM 树的合并逻辑, 来达成上述复制组数据的隔离。LSM 树原有的

合并逻辑, 将 L_i 和 L_{i+1} 层中的候选 SST 文件的键值对分别按顺序读取出来, 并将两层的键值对进行合并后写入新的 L_{i+1} 层的 SST 文件。

PheonixLSM 修改后的合并流程中, 会根据键值对 key 的前缀判断其所属的复制组, 当处理的复制组发生改变时(如, 刚刚处理的 key 为 002_foo1, 而正在处理的 key 为 003_foo2, 说明复制组 002 的数据已经被写完, 开始写复制组 003 的数据), PheonixLSM 会将当前 SST 文件关闭, 并将当前键值对写入新的 SST 文件。从而保证, 在 L_i 及更低层的任何 SST 文件中, 都只包含属于某一个复制组的键值对。

图 6 是使用 PheonixLSM 进行本地引擎的旧状态机删除, 以及新状态机快照的安装流程。删除旧状态机时, 首先强制回刷 Memtables, 并在回刷后将 L_0 的 SST 文件合并至 L_1 。此时, 所有属于该状态机的键值对都存储在某个 SST 文件集中, 且该集中的 SST 文件不包含其他复制组的状态机数据。删除操作可直接删除复制组所有相关的 SST 文件。

PheonixLSM 在删除状态机时(图 6(b)~(c)), 遍历 LSM 树视图中的 L_1 层及以下各 SST 文件, 视图中保存了每个 SST 文件的 key 区间。当某 SST 文件包含前缀为目标复制组编号的 key 时, 说明该 SST 文件中保存的都是目标复制组中的数据, 就可以进行删除操作(图 6(c))。删除旧状态机之后空间被完全回收, 新状态机快照在此时进

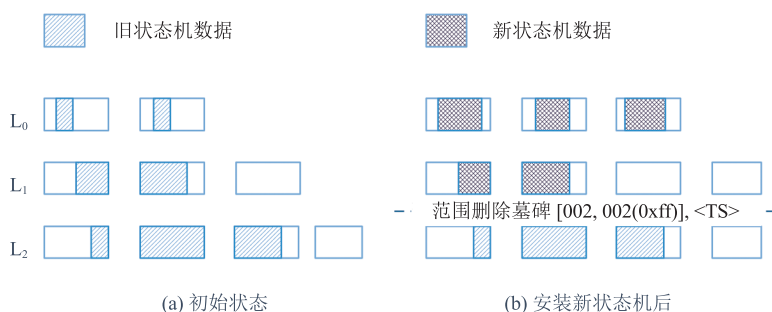


图 5 原生本地引擎的全量修复流程和其造成的空间放大

Fig. 5 Full-node repair with original local engine and the spatial amplification it caused

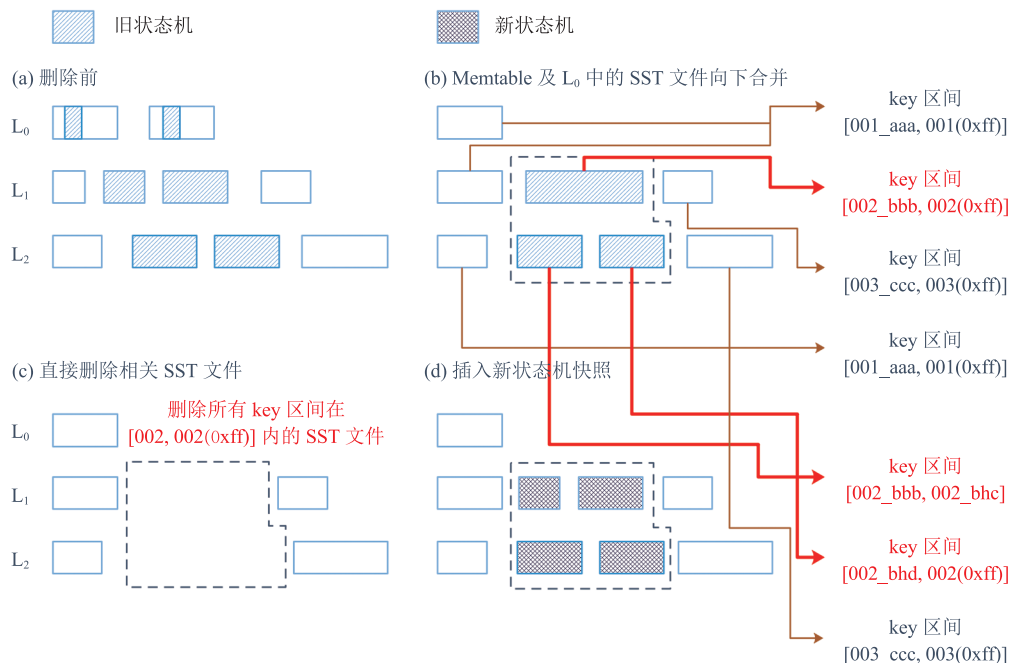


图 6 使用 PhoenixLSM 进行全量修复的旧状态机删除和新状态机快照的安装流程

Fig. 6 Full-node repair process with PhoenixLSM, including deleting old state machine and installing new one

行安装(图 6(d))。

一致性模块的修复操作, 需要先将状态机快照安装完毕, 才能重放日志, 因此, 在删除旧状态机并插入新状态机快照的过程中, 本地引擎不会收到该复制组请求。删除流程可以确保在强制回刷合并至 L_1 之后, 复制组的所有键值对都已经在 L_1 或更低层。PhoenixLSM 的删除操作, 能将旧的状态机完全删除, 且其他复制组不受影响。

5 原型实现和实验结果

5.1 原型实现和部署

PhoenixLSM 的原型基于 RocksDB 5.18.0 实现, 对 RocksDB 的代码库仅有约 200 行代码的改动。具体的改造方法见第 4 节。

在实际部署中, PhoenixLSM 作为自研分布式键值存储系统 PhoenixKV 的本地引擎。如图 7 所示, PhoenixKV 有 3 个主要组件: 数据服务器 KVAgent、元数据服务器 MetaServer 和客户端。

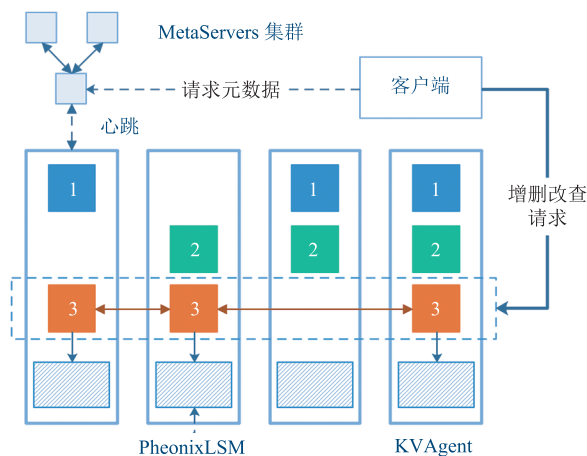


图 7 PhoenixKV 架构

Fig. 7 PhoenixKV architecture

PhoenixKV 将 key 空间映射到若干个使用 RAFT 协议维护一致性的复制组。复制组的节点分布在不同的 KVAgent 上, 以提供容错性。KVAgent 负责本地引擎和复制组节点的初始化, 本地资源的分配和清理等, 并定期心跳向 MetaServer 汇报健康状况。MetaServer 根据集群健康和负载状况, 通过心跳回复指示

对应 KVAgent 执行节点迁移。KVAgent 通过 GetPersisted 接口从 PheonixLSM 获取最新持久化的日志索引, 指导日志截断。MetaServer 集群也使用 RAFT 协议来维护关键元数据的一致性。客户端先向 MetaServer 请求路由信息, 包括复制组所在的 KVAgent 列表, 以及 key 到复制组的映射规则, 之后根据路由信息将请求发至对应的 KVAgent 处理。

5.2 实验环境与工具

本实验是在一个三主机的集群环境上进行的, 每台主机配备一颗 14 核 2.40 GHz 的 Intel Xeon E52680 v4CPU, 128 MiB 内存, 以及两块 480 GiB 的 Intel S4600 SATA SSD。3 台主机通过 10 Gbps 的网络连接。每台主机运行两个 KVAgent 以及一个 MetaServer。每个 KVAgent 管理其独占的 SSD, 故单 SSD 故障只会影响一个 KVAgent。PheonixKV 系统中有 256 个三副本复制组, 其节点分布在不同主机的 KVAgent 上, 提供主机级别的故障域。

本实验使用 RocksDB 自带的 db_bench 测试本地引擎, 并使用自主研发的 PheonixKVTest 测试工具测试 PheonixKV 性能。PheonixKVTest 使用指定并发数执行键值对的插入、删除、读取和校验操作, 同时可监控性能, 实时输出当前的时延和吞吐数据。PheonixKV 的目标是承载新型视频监控系统产生的对象存储元数据, 该业务模型会高并发写入约为 256 字节的键值对, 读请求较少。PheonixKVTest 模拟这一业务产生的元数据流量, 但是也可以自行定义读写比例, 以及读写键值对的大小和概率分布等。参考 MemC3^[9], 使用不同读写比例的业务流量来测试 PheonixLSM 在不同流量模型下的性能表现。

5.3 实验结果及分析讨论

本地引擎测试。使用 db_bench 的随机读写模式, 对比 PheonixLSM 和 RocksDB 的读写时延和尾时延(99 分位时延)。每轮实验使用 16 线

程, 共下发 1 000 万条随机读写请求。请求的 key 和 value 为 128 字节的字符串, key 的选择符合平均分布。调整读写请求的比例, 图 8 是不同本地引擎的读写时延对比结果。由图 8 可知, PheonixLSM 和关掉 WAL 的 RocksDB 各项时延和尾时延差异均在 3% 之内, 说明 PheonixLSM 的改造并没有带来性能损耗。当读写时延不同时, 与 RocksDB 相比, PheonixLSM 的写时延降低 94.1%~95.2%, 说明关闭 WAL 能带来较大的性能提升。

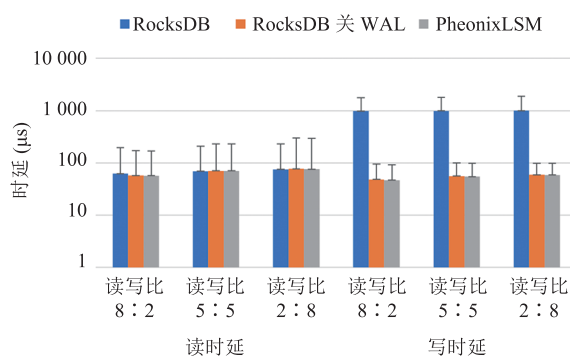


图 8 不同本地引擎的读写时延(平均时延和 99 分位时延)对比

Fig. 8 Read/write latency (average and 99-percentile) comparison of local engines

双写消除。本文考察双写消除为 PheonixKV 带来的性能提升。本实验使用 1/4/16/64 的并发数, 分别向 PheonixKV 集群插入 1 000 万条 key 和 value, 其均为 128 字节的随机键值对。图 9 为使用 RocksDB 和 PheonixLSM 作底层引擎时, PheonixKV 的吞吐对比结果。由图 9 可知, 当并发数不同时, 与 RocksDB 相比, PheonixLSM 的吞吐提升 32.0%~90.5%, 其中, 当并发数为 16 时, 吞吐提升最大。增加并发数, RocksDB 和 PheonixLSM 的单位计算资源吞吐均出现下降的情况, 这是由于 CPU 需要处理更多的线程调度工作。在不同并发数下, PheonixLSM 能带来 8.0%~21.7% 的单位计算资源吞吐提升, 且在并发数为 64 时的吞吐提升最大。

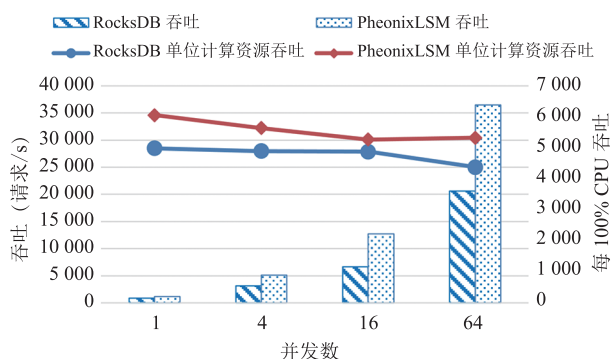


图9 使用 RocksDB 和 PhoenixLSM 作底层引擎时 PhoenixKV 的吞吐对比

Fig. 9 Throughput comparison of PhoenixKV with RocksDB and PhoenixLSM, respectively

将并发数固定为 64，以请求的读写比例为自变量，对比不同读写比例下 PhoenixKV 使用两种底层引擎的性能。向 PhoenixKV 下发 1 000 万条请求，吞吐和读写时延(包括平均时延和 99 分位时延)的对比结果如图 10 所示。对于同一种引擎，读写比对于读写时延影响较小；当使用不同引擎时，读时延差异较小，均在 8% 以内。当使用 PhoenixLSM 时，PhoenixKV 在不同读写比例下的平均写时延和 99 分位写时延分别降低 31.3%~35.5% 和 28.0%~37.8%。由图 10 所示，PhoenixLSM 将吞吐提升了 26.5%~50.3%。由于 PhoenixLSM 主要改善写性能，当读写比为 2:8 时，写请求占比最大，吞吐提升最多。

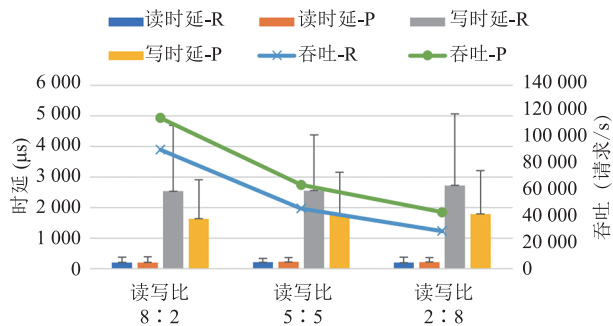


图10 不同读写比业务流量下的吞吐和读写时延(平均时延和 99 分位时延)对比

Fig. 10 Throughput and latency (average and 99-percentile) comparison of different read/write ratio

本文验证 PhoenixLSM 的状态机实时删除优化，对空间放大问题带来的改善。首先在系统中写入 5 亿键值对，并结束一个 KVAgent 进程，当继续写入 1 000 万键值对后，重新开启该 KVAgent 进程。在修复过程中，以每秒一次的频率对该 KVAgent 的磁盘空间使用情况进行采样。图 11 为全量修复过程中的空间放大对比结果。当 RocksDB 作本地引擎时，大量数据插入引发的合并操作，产生了较大的额外空间占用，即使在修复结束后，由于各层 SST 文件的分布还不稳定，后台仍在进行合并操作，所以额外空间的占用还在增加。此外，后台的合并任务也影响了正常的数据写入，导致修复时间变长。以修复后的稳态空间占用为基准，当 RocksDB 为本地引擎时，空间放大最多为 65.6%；而当 PhoenixLSM 为本地引擎时，空间放大最多仅为 6.4%。由图 11 可知，PhoenixLSM 将修复用时从 4 398 s 降低至 1 217 s，节约了 72.3% 的修复用时。

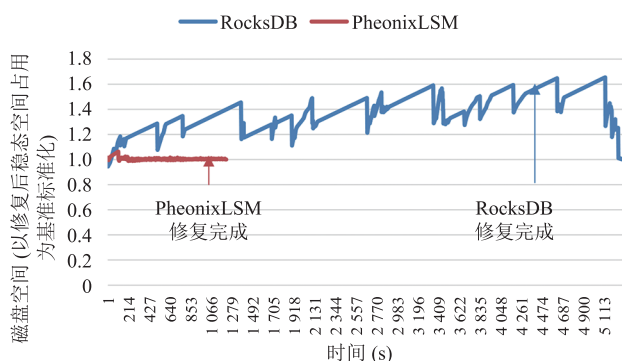


图11 全量修复过程中的空间放大对比

Fig. 11 Comparison of spatial amplification during full-node repair

PhoenixLSM 修改合并逻辑，潜在地增加 SST 文件数目。图 12 对比了随着 PhoenixKV 插入键值对数目的变化，不同底层引擎中 SST 文件数目及占用存储空间的变化。当键值对数目较少时，PhoenixLSM 导致 SST 文件数目增加较多；当键值对为 1 000 万个时，RocksDB 仅有 20 个 SST 文件，而 PhoenixLSM 有 210

个。然而, 当数据规模继续扩大时, SST 文件的数目差异会减少, 当键值对为 10 亿个时, RocksDB 和 PheonixLSM 分别有 2 161 和 2 176 个 SST 文件, 数目相差很小。综合而言, PheonixLSM 带来的 SST 文件数目增加, 对于底层的文件系统来说, 不会明显增加处理压力。当键值对数目逐渐增加时, 与原生 RocksDB 相比, PheonixLSM 占用的存储空间均没有增加, 请求处理的性能也没有因此发生下降(见图 8)。

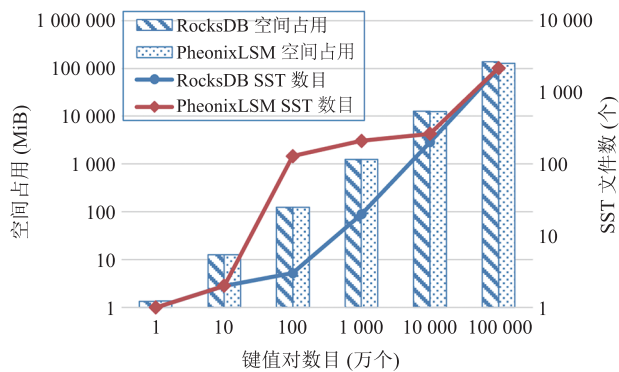


图 12 RocksDB 和 PheonixLSM 的 SST 文件数目和空间占用对比

Fig. 12 Number of SST files and space usage for RocksDB and PheonixLSM

6 讨论与分析

作为存储系统的核心组件, 分布式键值存储的性能和承载规模至关重要。然而, 本地引擎并没有很好地适配分布式键值存储的业务特征。本文提出了针对分布式业务优化的底层引擎 PheonixLSM, 解决了一致性协议日志和本地引擎 WAL 双写问题, 以及全量修复过程中的额外空间放大问题。据相关调查表明, PheonixLSM 是第一个针对分布式键值存储业务改造的底层引擎^⑨, 其有效解决了分布式键值存储的痛点,

能够将吞吐提升 90% 以上; 还可将全量修复的空间放大由 65.6% 降低至 6.4%, 效果明显; 此外, 还有效提升了分布式键值存储的业务和修复性能, 以及承载规模。在后续的工作中, 还将进一步结合分布式框架的业务模型, 针对性优化底层引擎, 从引入新型索引数据结构、底层文件组织, 以及发挥新型硬件特征等角度, 进一步提升分布式键值存储的性能和承载规模。

分布式键值存储的性能和承载规模, 取决于底层引擎的能力, 以及上层分布式路由模块分发流量、均衡负载和存储的能力。本文重点关注前者, 后续基于 PheonixKV 的工作中, 如何确保性能随集群规模扩大线性提升, 各底层引擎性能均能得到发挥也将是重点研究方向。

7 结 论

本文描述了 PheonixLSM 的设计与实现。PheonixLSM 是一个基于 LSM 树的本地键值引擎, 面向分布式键值存储, 进行了针对性的优化。PheonixLSM 通过改造 LSM 树的增删改和回刷流程以及 SST 文件的组织形式, 配合上层分布式逻辑, 解决了分布式键值存储系统中一致性协议的双写引发的性能问题, 以及全量修复时的额外空间放大问题。本文在 RocksDB 的源代码上进行改造实现了 PheonixLSM, 并在三主机的分布式集群上进行了实验。实验结果证明, PheonixLSM 带来了性能改进, 以及全量修复过程中空间放大问题的有效消除。与原生 RocksDB 的分布式键值存储相比, PheonixLSM 的分布式键值存储, 吞吐提升 90% 以上, 单位计算资源吞吐提升 20% 以上。全量修复过程中的空间放大由 65.6% 下降至 6.4%, 并减少了 72.3% 的修复时间。

注^⑨: 使用谷歌学术于 2021 年 5 月 26 日检索英文关键字 “distributed key-value storage; spatial amplification; extra synchronization” 以及中文关键字 “分布式键值存储; 空间放大; 双写” 的检索结果中, 均没有相关的工作。

参考文献

- [1] Lamport L. Paxos made simple [EB/OL]. (2001-11-01)[2021-05-21]. <https://lamport.azurewebsites.net/pubs/paxos-simple.pdf>.
- [2] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm [C] // Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, 2014: 305-320.
- [3] Google Inc. LevelDB [EB/OL]. (2021-02-24)[2021-05-21]. <https://github.com/google/leveldb/releases/tag/1.23>.
- [4] Facebook Inc. RocksDB [EB/OL]. (2021-05-06)[2021-05-21]. <https://github.com/facebook/rocksdb/tree/v6.20.3>.
- [5] PingCAP Inc. TiKV [EB/OL]. [2021-05-21]. <https://docs.pingcap.com/tidb/stable/tikv-overview>.
- [6] Lakshman A, Malik P. Cassandra: a decentralized structured storage system [J/OL]. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40. DOI: 10.1145/1773912.1773922.
- [7] Redis Labs. Redis [EB/OL]. [2021-05-21]. <https://redis.io/>.
- [8] Danga Interactive. Memcached [EB/OL]. [2021-05-21]. <https://memcached.org/>.
- [9] Fan B, Andersen DG, Kaminsky M. MemC3: compact and concurrent Memcache with dumber caching and smarter hashing [C] // Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, 2013: 371-384.
- [10] Wikipedia. B+ Tree [EB/OL]. (2021-05-16)[2021-05-21]. https://en.wikipedia.org/wiki/B%2B_tree.
- [11] Oracle Inc. MySQL Documentation on MyISAM Storage Engine [EB/OL]. [2021-05-21]. <https://dev.mysql.com/doc/refman/8.0/en/myisam-storage-engine.html>.
- [12] Oracle Inc. MySQL Documentation on InnoDB Storage Engine [EB/OL]. [2021-05-21]. <https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html>.
- [13] Dong S, Kryczka A, Jin Y, et al. Evolution of development priorities in key-value stores serving large-scale applications: the RocksDB experience [C] // Proceedings of the 19th USENIX Conference on File and Storage Technologies, 2021: 33-49.
- [14] Cao ZC, Dong SY, Vemuri S, et al. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook [C] // Proceedings of the 18th USENIX Conference on File and Storage Technologies, 2020: 209-223.
- [15] Raju P, Kadekodi R, Chidambaram V, et al. PebblesDB: building key-value stores using Fragmented Log-Structured Merge Trees [C/OL] // Proceedings of the 26th Symposium on Operating Systems Principles, 2017: 497-514. DOI: 10.1145/3132747.3132765.
- [16] Lu LY, Pillai TS, Gopalakrishnan H, et al. WiscKey: separating keys from values in SSD-conscious storage [J]. ACM Transactions on Storage, 2017, 13(1): 1-28.
- [17] Chen H, Ruan CY, Li C, et al. SpanDB: a fast, cost-effective LSM-tree based KV store on hybrid storage [C] // Proceedings of the 19th USENIX Conference on File and Storage Technologies, 2021: 17-32.
- [18] Zhang T, Wang JY, Cheng XT, et al. FPGA-accelerated compactions for LSM-based key-value store [C] // Proceedings of the 18th USENIX Conference on File and Storage Technologies, 2020: 225-237.
- [19] Li Y, Tian C, Guo F, et al. ElasticBF: elastic Bloom filter with hotness awareness for boosting read performance in large key-value stores [C] // Proceedings of the 2019 USENIX Annual Technical Conference, 2019: 739-752.
- [20] Wu FG, Yang MH, Zhang BQ, et al. AC-key: adaptive caching for LSM-based Key-Value Stores [C] // Proceedings of the 2020 USENIX Annual Technical Conference, 2020: 603-615.
- [21] Zhong WS, Chen C, Wu XB, et al. REMIX: efficient range query for LSM-trees [C] // Proceedings of the 19th USENIX Conference on File and Storage Technologies, 2021: 51-64.
- [22] Balmau O, Dinu F, Zwaenepoel W, et al. SILK: preventing latency spikes in log-structured merge key-value stores [C] // Proceedings of the 2019 USENIX Annual Technical Conference, 2019: 753-766.