Key-Value 型 NoSQL 本地存储系统研究

马文龙[1),2)] 朱妤晴[1)] 蒋德钧[1)] 熊 劲[1)] 张立新[1)] 孟 潇[3)] 包云岗[1)]

[1)](中国科学院计算技术研究所先进计算机系统研究中心 北京 100190)

[2)](中国科学院大学 北京 100049)

[3)](工业和信息化部信息中心 北京 100846)

摘 要 NoSQL 系统因其高性能、高可扩展性的优势在大数据管理中得到广泛应用,而 key-value(KV)模型则是 NoSQL 系统中使用最广泛的一种存储模型. KV 型本地存储系统对于以机械磁盘为持久化存储的情形,存在许多性能优化技术,但这些优化技术面对当前的硬件发展新趋势,如多核处理器、大内存和低延迟闪存、非易失性内存 NVM(Non-Volatile Memory)等,难以充分发挥新硬件的优势,如数据索引、并发控制、事务日志管理等技术在多核架构下存在多核扩展性问题,又如数据存储策略不适应闪存 SSD(Solid State Drive)的新存储特性而产生了 IO 利用率低效的问题. 针对多核处理器、大内存和闪存、NVM 等硬件发展新趋势,文中面向当前的大数据应用背景,综述了 KV 型本地存储系统在索引技术、并发控制、事务日志管理和数据放置等核心模块上的最新优化技术和系统研究成果. 从处理器、内存和持久化存储的角度概括了 KV 型本地存储系统当前存在的最优技术,总结了当前研究尚未解决的技术挑战,并对 KV 型本地存储系统在 CPU 缓存高效性、事务日志扩展性和高可用性等方面的研究进行了展望.

关键词 NoSQL;键值存储;多核扩展性;并发数据结构;日志结构合并型存储;SSD/NVM

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2018.01722

A Survey on Local Key-Value Store of NoSQL System

MA Wen-Long[1),2)] ZHU Yu-Qing[1)] JIANG De-Jun[1)] XIONG Jin[1)]

ZHANG Li-Xin[1)] MENG Xiao[3)] BAO Yun-Gang[1)]

[1)](*Center for Advanced Computer Systems, Institute of Computing Technology Chinese Academy of Sciences, Beijing* 100190)

[2)](*University of Chinese Academy of Sciences, Beijing* 100049)

[3)](*Information Center, Ministry of Industry and Information Technology, Beijing* 100846)

Abstract NoSQL system is widely used in big data management because it has the great features of high performance and high scalability. Among all data models of NoSQL systems, the key-value (KV) model is the most widely used one. A large number of methods for performance improvements have been proposed for local KV stores that use SATA disks as the persistent storage. However, these methods cannot fully exploit new features and advantages of the emerging hardware technology, such as multi-core processors, larger memory, flash disks and low-latency non-volatile memory (NVM) storage devices. For example, the software components of indexing, concurrency control and transaction log management are facing the scalability problem on multi-core platforms; the data persistence strategy has not yet fully exploited the characteristics of flash devices such as SSD, leading to the unnecessarily inefficient I/O utilization. Firstly, this survey reviews in turn the state-of-the-art research on indexing, concurrency control, transaction logging and data persistence methods for local KV stores, focusing on how these works solve the challenges posed by the new hardware technology and by the big data applications. The reviewed indexing methods

收稿日期:2016-07-27;在线出版日期:2017-05-31. 本课题得到国家"九七三"重点基础研究发展规划项目(2014CB340402)、国家自然科学基金(61303054,61420106013)、山东省自然科学基金(ZR2016FM41)资助.**马文龙**,男,1991 年生,博士研究生,中国计算机学会(CCF)会员,主要研究方向为内存数据库、并发数据结构. E-mail: mawenlong@ict.ac.cn. **朱妤晴**,女,1983 年生,博士,助理研究员,中国计算机学会(CCF)会员,主要研究方向为分布式系统和数据管理. **蒋德钧**,男,1982 年生,博士,副研究员,中国计算机学会(CCF)会员,主要研究方向为存储系统、内存架构、操作系统、分布式系统. **熊 劲**,女,1968 年生,博士,研究员,博士生导师,中国计算机学会(CCF)会员,主要研究领域为大数据存储管理、分布式文件系统、基于 SSD/NVM 的存储系统. **张立新**,男,1971 年生,博士,研究员,博士生导师,中国计算机学会(CCF)会员,主要研究领域为数据中心系统、内存系统、体系结构模拟器、并行计算、性能评估和负载分析等. **孟 潇**,男,1987 年生,工程师,主要研究方向为分布式系统、数据管理. **包云岗**,男,1980 年生,博士,研究员,博士生导师,中国计算机学会(CCF)会员,主要研究领域为计算机体系结构、操作系统、性能评估等.

include hash table, B+tree, LSM-tree and their variants. The surveyed concurrency control methods are locking, multi-version based optimistic concurrency control, lock-free concurrency control and data-partition based concurrency control schemes. For transaction logging, this survey focuses on the NVM-based methods and the analysis of their scalability property. With regard to data placement, this survey presents the state-of-the-art research results on in-memory data caching and data persistence. Among all combinations of the surveyed methods, LSM-based KV store with the best concurrency control and data placement methods is a promising storage solution for the emerging hardware technology. As a result, this survey summarizes and compares in detail the state-of-the-art LSM-based KV storage systems, with a focus on the systems of bLSM-tree, HyperLevelDB, RocksDB, LOCS, cLSM-tree, LSM-tree and WiscKey. Secondly, this survey points out the research challenges unsolved by the state-of-art research. Such challenges include the challenges that the multi-core architecture poses on the scalability of indexing and concurrency control, and the challenges that new storage media like SSD and NVM pose on the transaction log management and data placement. For the challenges, the survey concludes on the state-of-the-art technology choices for the design and optimization of local KV storage systems. The conclusions are (1) the best concurrency control methods for indexes are multi-version concurrency control and lock-free concurrency control, with example systems like Bw-tree and cLSM-tree, and (2) LSM stores best matches the characteristics of new storage media like SSD, with example systems like MICA, FAWN-DS, LSM-trie and WiscKey. Finally, this survey also presents prospects on the research of local KV store, involving the aspects of CPU cache high-efficiency, transaction log scalability, and high availability. These prospects can be interpreted as three core research questions: (1) how to improve KV stores performance by exploiting the high capacity of CPU caches; (2) how to guarantee the scalability of transaction logging in KV stores and fully utilize the multi cores of CPUs; (3) how to provide high-performance accesses for both large and small data in KV stores.

Keywords　NoSQL; key-value store; multi-core scalability; concurrent data structures; LSM-based storage; Solid State Drive/Non-Volatile Memory

1　引　言

当前具有 4V(Volume 海量数据、Velocity 高度并发、Variety 多种结构数据并存、Value 数据项之间较弱的相关性使数据价值变得隐藏)特征的大数据,对存储系统的性能和功能需求发生了颠覆性变化.传统的存储系统如文件系统、关系型数据库已无法满足大数据存储的诸多需求.随着以大数据为基础的新型互联网服务产业的兴起,这一局面产生了一些改变,被称为 NoSQL 的新型存储浪潮对新应用背景下大数据的存储系统的发展产生了重大影响,出现了诸多新型的存储系统.

NoSQL 是一种非关系型的、分布式的、不严格遵循 ACID 原则并且高可扩展的新型数据存储系统①,并分为 key-value 存储、文档数据库和图数据库,其中 key-value 存储备受关注,已成为 NoSQL 的代名词[1].在亚马逊的 Dynamo[2] 和谷歌的 BigTable[3] 以论文形式发表后,key-value 存储系统的研发和部署在许多互联网服务商中广泛开展起来,并有大量的开源产品出现,引起了工业界和学术界的强烈关注.这种新型存储系统在大数据应用场景允许的前提下,通过放弃严格的 ACID 事务语义和复杂的 SQL 或者 POSIX 接口标准,采用简单灵活的 key-value 数据模型来换取更好的性能和水平扩展能力[4].由于存储系统的功能和性能之间的权衡是由存储系统的使用者本身的业务需求决定的,故而此类系统的研发呈现出百花齐放的态势.key-value 存储根据数据的组织结构不同,又细分为 key-document 存储、key-column 存储和 key-value 存储.key-document 存储,代表存储系统如 MongoDB、CouchDB;key-column 存储,代表存储系统如谷歌的 BigTable[3],Apache 的 Hbase、Cassandra[5],雅虎的 PNUTS[6];key-value 存储,代表存储系统如 BerkeleyDB、Redis②、LevelDB③、Tokyo Cabinet.

key-value 存储是一种当前应用最广泛的 NoSQL 数据库系统,而单节点的 key-value 存储系统(KV

① NoSQL. http://nosql-database.org/2016,3,20
② Redis. http://redis.io 2016,3,20
③ LevelDB. https://code.google.com/p/leveldb 2016,3,20

型本地存储系统)是实现高扩展高性能分布式NoSQL存储系统的引擎,运行在集群中的各个节点.典型的NoSQL系统如亚马逊的Dynamo,它将KV型本地存储系统BerkeleyDB作为集群中各节点的存储引擎;谷歌的BigTable、Apache的Cassandra和Hbase、雅虎的PNUTS等将日志结构合并型(LSM型)存储作为它的本地存储引擎;以及如Redis、Memcached等以本身为存储引擎进行水平扩展的分布式系统.KV型本地存储系统的性能是整个分布式集群性能的关键,当前已涌现出众多关于它的研究工作并有了一些研究成果,主要表现在数据索引、并发控制、事务日志和数据放置四个方面.但也存在着很多问题和挑战待探索和解决,这是本文选择这一KV型本地存储系统作为综述研究对象的动机.当前KV型本地存储系统的架构因应用需求的不同而各异,但它们具有共同的核心组件,包括数据索引、事务管理、缓存管理、并发控制、数据放置策略和事务日志管理.大数据给KV型本地存储系统带来了机遇,使得它的应用需求不断扩大,但同时面对大数据Velocity和Volume的特性,该存储系统的各个组件也遇到性能挑战.然而,CPU和存储等硬件技术的不断成熟和发展,正好给KV型本地存储系统创造了很好的性能扩展环境,但是该系统各个组件的传统设计在新型硬件平台上,并不能充分发挥硬件的性能,在设计时需要注意和考虑相关的问题,让程序的运行更加适应硬件的特性.

在多核架构下KV型本地存储系统遇到了多核扩展的瓶颈,突出表现在并发控制和事务日志管理两大组件.摩尔定律揭示了集成电路中晶体管数量增长的规律,CPU技术发展到今天,由于晶体管的高度集成带来了过多的能耗,因而CPU的时钟频率无法得到持续提高.当前摩尔定律主要体现在CPU核数的不断增加,出现了多核CPU和多核多CPU的多核架构,如IBM POWER7®对称多处理器架构每个核心支持4个线程、每个芯片支持8个核心及每台服务器支持32个芯片插槽来实现高度并行性,总共有1024个并发硬件线程①.根据摩尔定律和最近的多核发展趋势,单个CPU上的核数以一定速率仍在保持增长.然而KV型本地存储系统的传统的架构设计并不能充分利用多核的并行计算能力.在并发控制方面,如何使用有效的并发控制机制减少或避免多核对共享内存数据的竞争,以充分发挥多核的硬件级性能成为关键.在多核多处理器环境下,避免多线程对内存的竞争显得尤为重要.在内存和缓存中,各种不同的核共享一个通用的数据区域,这需要在它们之间进行同步.当不同的核心同时访问同一个数据区域时,会发生内存争用.在不同的核之间同步数据,因总线通信、传统的基于锁的并发控制带来的线程阻塞与上下文切换导致很大的性能损失[7].因此,当前的研究焦点是优化传统的锁机制,减少加锁对多线程的阻塞和上下文切换开销,典型工作如雅虎的cLSM-tree[7]和微软的Bw-tree[8].在读并发控制方面,采用基于数据多版本的并发控制很好地支持了多线程并发读而不受写线程加锁的阻塞影响.在写并发控制方面,基于无锁的并发控制很好地支持了多线程并发写而不用加锁,使得系统具有很好的多核扩展性;在事务日志管理方面,当前KV型本地存储系统中的事务日志是基于写前日志(WAL)协议的,并且它是一个集中式的日志.这样的设计存在两个问题,一是由于WAL要求在更新内存数据之前必须先持久化日志,以在系统发生宕机等情况时保证数据的可靠性.虽然写WAL是顺序的追加操作,但对于块设备如SSD、HDD而言,频繁的小粒度顺序追加写在I/O上的延迟,仍直接影响系统的性能[9].二是在多核多任务高并发环境下,集中式的单日志因多线程对其日志缓存的竞争,成为系统瓶颈.根据已有的研究工作,多线程在集中式日志缓存上的竞争时间约占到CPU在整个事务日志上时间的46%[10].当前在日志扩展性方面的热点研究是,采用本身以字节力度寻址并且跟DRAM同等读写延时的新型非易失型内存NVM作为事务日志的缓存,在NVM上聚合事务日志,然后通过异步的方式将缓存日志写回块设备,如SSD等.同时在NVM上实现分布式事务日志,解决多核在集中式日志上的性能扩展性问题.

在新型存储介质下KV型本地存储系统传统的数据放置策略遇到了I/O利用率低下的问题.随着存储技术的不断成熟和发展,非易失性内存NVM、高速闪存等新型存储介质被应用在数据库系统环境中,尤其闪存SSD因其成本的不断降低得到了广泛应用.SSD擅于随机读和顺序读写,由于其基于写前擦除的垃圾回收机制和地址转换原理,随机写不利于其性能的发挥[11].在面对大数据Velocity的特性时,传统的数据放置策略已不能充分适应这种新型存储设备的性能特性.不论是传统的机械硬盘还是新型的存储介质SSD,针对这些存储介质的数据放置原则仍遵循I/O五分钟原则,这一原则是数据库存储系统设计的参考之一.I/O五分钟原则,即如果一条记录被频繁访问,就应该放到内存里,否则就应该放在硬盘上按需要再访问,这个临界点就是五分钟②.五分钟的评估标准是根据投入成本来判断

① https://software.intel.com/en-us/articles/intel-guide-for-developing-multithreaded-applications/2016,3,20

② http://queue.acm.org/detail.cfm?id=1413264 2016,4,2

的，根据当时的硬件发展水准，在内存中保持 1 KB 数据的成本相当于在硬盘中存储同样大小数据 400 秒的开销（接近五分钟）. 在闪存时代，五分钟法则仍然有效，只是将新型存储介质 SSD 当成较快的硬盘使用①. 在当前大数据负载下存在部分全内存的缓存数据库系统，但大多数数据库系统仍遵循 I/O 五分钟原则，将冷热数据进行分离存储. 然而对于大数据环境下的 KV 存储系统来说，由于负载数据量的成倍增加和内存数据更新的高并发使得内存跟 SSD 之间的数据交换次数大大增加，如何避免小粒度随机写，优化数据的读写方式以适应 SSD 的硬件特性，避免或减少系统的 I/O 瓶颈成为研究焦点. 当前的研究中采用日志结构存储，对内存和持久化设备的写操作以追加日志的方式进行. 一方面将小粒度的请求聚合为大粒度请求节省 I/O，另一方面将随机写转化为了顺序写，降低随机写操作导致的 SSD 频繁垃圾回收和地址转换带来的开销，典型的工作如 FAWN-DS[11]、SILT[12] 等.

索引是整个存储系统的核心，KV 型本地存储系统的并发控制和数据放置等组件的设计直接跟索引息息相关. 一方面在多核多处理器的架构下需要优化索引本身的结构以提升多核在索引上的并行执行能力，避免 CPU 瓶颈；另一方面大内存的硬件条件允许将更多的数据驻留在内存，加上新型的存储介质 SSD、NVM 等持久化存储支持，需要优化索引结构减少 I/O，避免 I/O 瓶颈.

本文依次从数据索引技术研究、并发控制研究、事务日志扩展性研究和数据放置策略研究四个维度，对 KV 型本地存储系统的研究现状进行分析综述和归纳总结. 同时从系统研究的视角横向梳理和总结了 LSM 型 KV 系统的研究进展，该类系统在新硬件平台上有着其出色的性能，成为了工业界的主流 NoSQL 存储系统，同时也成为学术界的研究热点.

2　系统架构

当前 KV 型本地存储系统的架构因负载需求而各异，但各系统有着共同的核心组件，包括数据索引引擎、事务管理、缓存管理、多线程任务访问共享数据的并发控制、数据在持久化设备上的放置策略以及保证数据可靠性的事务日志管理. 如图 1 所示.

作为 NoSQL 数据库系统，KV 型本地存储系统在事务方面并没有严格遵守关系型数据库中 ACID 属性，没有复杂的事务管理机制，而是对应用层提供简单的数据库编程接口，以便快速高效地访问.

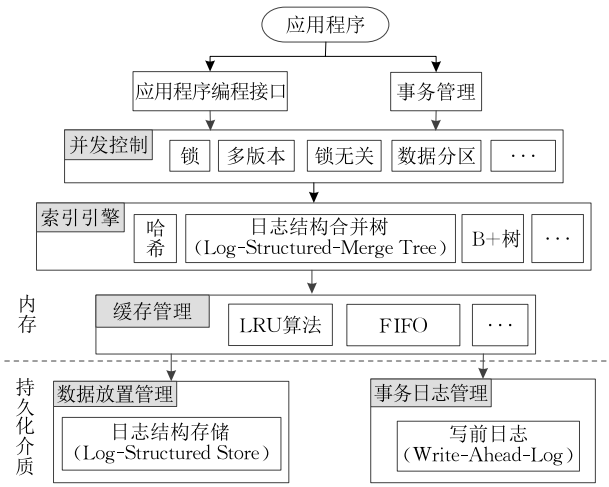


图 1　KV 型本地存储系统架构

BerkeleyDB 比较特殊，作为关系型数据库和 NoSQL 数据库之间的数据库系统，既有如关系型数据库中的完整 ACID 事务语义支持，又提供像 NoSQL 中的简单高效的数据库编程接口. 缓存管理也是 KV 型本地存储系统中必不可少的模块，不同系统采用的缓存管理方式基本相似，如 LRU 算法、FIFO 队列等. 对于 KV 型本地存储系统来说，事务管理和缓存管理不是当前研究所关注的内容. 当前研究的焦点主要包括高效的索引引擎、多核并发访问共享内存数据的并发控制技术、高吞吐率低延时的数据放置策略和事务日志扩展性四个方面. 表 1 显示了当前典型的 KV 型本地存储系统及其关键技术，由于各系统在事务日志管理方面均采用基于写前日志协议（WAL）的事务日志，因此在列表中未标出.

表 1　典型 KV 型本地存储系统及其关键技术

典型系统	索引技术	并发控制	数据放置策略
Memcached	哈希	锁	哈希表
Redis	哈希	单线程	哈希表、快照、日志文件
BerkeleyDB	哈希、B+tree	锁	普通文件
FAWN-DS	哈希	锁	日志结构存储
SILT	哈希	锁	日志结构存储、哈希表存储、顺序哈希存储
LSM 型系统	LSM-tree	多版本、锁	顺序字符串表

LSM 型系统由于其独特的并发控制技术和数据放置策略，在多核和新型存储介质上有着出色的性能表现. LSM 型系统在索引技术上采用 LSM-tree 索引，数据放置方面采用日志结构存储，使用多版本的读并发控制和基于锁的写并发控制. 表 2 展示了 LSM-tree 型 KV 型本地存储系统及其采用的关键技术.

① http://cacm.acm.org/magazines/2009/7/32091-the-five-minute-rule-20-years-later/fulltext#R15 2016,4,2

表 2 LSM 型 KV 型本地存储系统及其关键技术

典型系统	解决的问题	关键技术点
LevelDB	基于 LSM-tree 索引的 KV 型本地存储系统开源实现	内存表使用跳表;单线程更新内存表;单线程合并顺序字符串表
bLSM-tree	合并顺序字符串表的读放大	使用布隆过滤器;优化合并算法
RocksDB	LevelDB 单线程的合并过程阻塞写请求,合并成为性能瓶颈	多个只读内存表;多线程进行合并;使用布隆过滤器减少读放大
LOCS	单线程的合并过程阻塞写请求,合并成为性能瓶颈;单纯的 LSM+SSD 仍不能有效解决 I/O 上的瓶颈	多个只读内存表;多线程合并;高效的多队列请求调度器;使用多通道 Open-Channel SSD
流水线合并	合并过程的顺序串行执行,在多核架构下存在扩展性瓶颈;在新型存储介质上具有较低的 I/O 利用率	将顺序合并过程分为 CPU 计算和执行 I/O 两个阶段,对每个阶段的处理流程流水线并行化
HyperLevelDB	随着 RAM 容量剧增及 Flash 存储的出现,系统瓶颈转向多线程对内存表的写竞争	跳表上使用更细粒度的锁,支持多线程并发的写
cLSM-tree	跳表上细粒度的加锁开销在当前多核环境下仍较大;加锁导致的线程阻塞和上下文切换开销成本较高	使用无锁 Lock-free 的并发跳表;通过增加控制合并过程的同步指针来消除内存表上的全局锁
WiscKey	较大粒度的 value 请求导致合并过程带来更大的读写放大	分离 value 数据到单独的日志中存储,value 记录地址

接下来先从宏观上介绍这些基本组件,后续章节依次对各个组件的研究现状进行阐述,并结合实际的系统以系统研究的视角详细综述 KV 型本地存储系统的研究现状,最后做出总结和展望.

2.1 索引技术

存储系统对外呈现出一定的模型,即按照数据组织方式对外部的呈现方式主要分为三种,即文件、关系型、键值型,它们是存储系统的外壳[13]. 而索引技术是存储系统的存储引擎,直接决定了存储系统能够提供的性能和功能. 索引是对数据库中一列或多列的值进行高效组织并满足特定查找算法的一种数据结构,在数据库存储系统中,这种具有索引功能的数据结构由数据库系统单独维护,这些数据结构以特定的方式引用或指向实际数据,这样就可以在这些数据结构上实现高级查找算法. 在 KV 型存储系统中,key 列关键字担负起了这个索引的使命,通过唯一的 key 就能映射查找到固定的数据项 value.

当前索引技术的研究主要为哈希、B+tree 和日志结构合并树 Log-Structured Merge Tree[14](以下简称 LSM-tree). 哈希索引技术的优化工作包括布谷哈希[15]、布隆过滤器[16]等,主要用于优化哈希冲突及冲突带来的读放大,典型的研究工作包括缓存哈希[17]、Redis、FAWN-DS[11] 及 SILT[12] 等. B+tree 索引技术的研究主要集中在优化节点的数据结构以减少 I/O 次数,典型的工作如 LA-tree[18]、Bw-tree[8]. B+tree 索引是实际的工业数据库系统中的标准索引引擎. LSM-tree 索引技术能够在内存中聚合小粒度的写请求数据,并以追加的方式将内存中聚合的大粒度数据写到持久化存储介质,以适应持久化设备的存储特性,提高 I/O 带宽. 当前对 LSM-tree 索引的研究集中在两个方面,一是如何减少其内部组件之间的合并导致的读写放大,如工作 RocksDB①、bLSM-tree[19]. 同时优化存储设备的存储特性以增加 I/O 带宽,为合并过程在硬件上提供更高额 I/O 带宽,典型的工作如 LSM-tree on Open-Channel SSD(简称 LOCS)[20]. 另一方面,优化 LSM-tree 索引的并发控制机制,多核环境下实现高扩展性,典型的工作如 HyperLevelDB、cLSM-tree[7].

2.2 并发控制

在以前的研究中重点关注的是读并发控制,而在当前的系统研究中,随着 CPU 技术的不断成熟,为应用层提供了像 Compare-and-Swap(简称 CAS)[21-22] 这样的 CPU 硬件锁支持的原子操作,使得对于多个写线程来说,无锁 lock-free 的写并发控制成为现实. 写并发控制在当前多核环境下显得尤为重要,它成为了当前研究的热点,像 Bw-tree[8]、cLSM-tree[7] 是典型的优化传统基于锁的写并发控制的研究工作,多线程无锁的高并发处理写请求,获得较好的多核扩展性.

2.3 事务日志扩展性

数据写内存之前先写日志,以保证数据的可靠性. 一般的,大多数系统采用的技术是 Write-Ahead-log② 即写前日志协议[23-24]. 那么 WAL 在当前系统中存在的问题是,在多线程高并发更新的环境下,无论数据放置技术对磁盘的写入是多么聚合,WAL 仍是小粒度落盘,并且多个线程对单个集中式的日志缓存共享写时存在竞争同步,这限制着事务日志的扩展性. 当前在数据库系统[9-10,25-28] 和文件系统[29-31] 方面已有初步的研究工作,核心思想是使用以字节粒度寻址且数据非易失的低延迟存储介质 NVM 来缓存 WAL,形成 DRAM、NVM 和 SSD 混合存储的多层日志架构. 由 NVM 作为中间层来转储 WAL,并跟 SSD 之间进行异步交换,减小了高并发写负载下 WAL 的扩展性瓶颈.

2.4 数据放置策略

数据放置技术,即针对存储介质进行的数据读写方式,使数据形成一定组织结构存储在介质上. 当

① A persistent key-value store for fast storage environments. http://rocksdb.org/2016,3,25

② Write-Ahead-log. https://www.sqlite.org/wal.html 2016,3,26

前在多核多处理器和大内存的硬件环境下，一个存储系统的性能瓶颈转向低吞吐率的硬盘等外部存储设备的 I/O 上. 而持久化存储设备有着自己的性能特性，小粒度的随机写不利于其带宽的高效利用. 于是如何设计高效的数据放置策略，充分发挥存储设备的硬件性能成为了系统设计的关键.

当前在数据库系统中，主要有两种数据放置策略. 一是将数据全部存储在内存中以获取极高的性能，如内存数据库 Memcached、Redis 等；另一种是改进存储设备包括传统的机械磁盘、新型的 Flash 存储 SSD 以及非易失内存 NVM 等上的数据组织形式，通过将低效率的随机写操作转化为顺序写，小粒度的请求聚合为大粒度请求，以提高 I/O 利用率.

对于全内存数据放置而言，根据 I/O 五分钟原则，面对当前 NoSQL 的云海量数据存储，由于较高的内存硬件成本和保证数据非易失所需要的功耗代价，加上被访问数据的冷热特性，将全部数据实时地放置在内存并不适宜[32]. 当前由于 SSD 闪存在性能和硬件成本上介于 DRAM 内存和机械磁盘之间，为了提高系统的数据处理性能，出现了基于 DRAM 内存和闪存 SSD 混合存储的 KV 存储系统[11-12,33-37]. 它们将 DRAM 内存作为存储系统中小部分元数据的缓存，而高速闪存 SSD 作为数据的存储介质. 因不同的存储介质有着不同物理存储特性，因此研究适合其特性并充分发挥其硬件性能的数据放置策略成为关键. 当前 HDD 和 SSD 在数据放置方面有着相似的特性[8]，频繁地小粒度随机写不利于二者性能发挥. 当前对该问题解决方案是 Log-Structured Store 即日志结构存储. 采用这种放置策略的工作包括以 LSM-tree 为索引引擎的系列系统研究，如 LevelDB 及以它为基础研发的系统如 RocksDB、HyperLevelDB①、LOCS[20]、bLSM-tree[19] 和 cLSM-tree[7] 等. 以 B+tree 为索引的研究 Bw-tree[8] 和以哈希为索引的 SILT[12] 等.

综上所述，当前对 KV 型本地存储系统的研究主要分为两个维度. 其一是，针对多核大内存环境，优化索引数据结构，获得高效快速的操作性能；优化针对不同索引数据结构的并发控制，实现多核的高扩展性. 同时，研究多核并发访问内存共享数据时，如何减少多 CPU 的缓存失效. 其二是，针对新型的存储设备如 SSD、NVM 等，如何优化数据放置策略，以充分利用低延迟、高吞吐率的硬件级性能；针对非易失性内存 NVM 的事务日志扩展性研究，减小整个系统在事务日志上的性能瓶颈.

索引是 KV 型本地存储系统的核心，并发控制和数据放置策略跟索引密切相关. 因此，下面结合实际的研究系统先介绍基本的索引技术及典型优化研究，作为叙述后续内容的基础.

3　索引技术研究

作为存储系统的发动机，不同的索引技术具有不同的特性，如索引数据结构占用内存的大小，查找记录 key 的效率即算法复杂度，并发访问的控制机制及效率等等. 下面介绍三种最典型且广泛应用于数据库存储系统中的索引技术及其相关研究.

3.1　哈希索引技术

3.1.1　传统哈希

哈希索引引擎是哈希表的持久化实现，如图 2 是一个典型的基于哈希索引的系统架构.

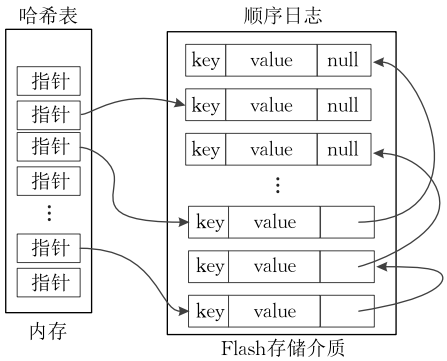


图 2　SkimpyStash 架构[35]

哈希表是根据键值 key 直接进行访问实际数据 value 的数据结构. 它通过映射函数把键值 key 映射到哈希表中一个位置，来获得实际数据 value 在存储介质 Flash 上哈希链中的位置，然后依次查找该哈希链，跟哈希链中的 key 值相比较，查找相应的 value 数据，从而直接访问记录以加快查找的速度. 因此，哈希型索引主要由内存中的哈希表和持久化存储介质上的哈希链构成. 在插入新的 KV 时，将该 KV 对插入到哈希链并更新哈希表，增加对该 KV 项地址的索引.

哈希型索引虽具有非常高的查询效率，但它也有自己的缺陷：(1) 支持插入、删除、修改以及随机读取操作，但不支持顺序扫描；(2) 随着哈希表的体积增大，内存占用开销越大；(3) 哈希冲突随着数据项增多而增加，哈希冲突导致请求必须扫描磁盘中的哈希链查找对应的值，从而加大了系统的 I/O 开销. 基于这些不足，对哈希型索引的主要优化工作是减少哈希冲突如布谷哈希[15] 和最小化读放大效应如布隆过滤器[16] 技术.

3.1.2　布谷哈希

为了减少哈希冲突，布谷哈希[15] 使用两个哈希函数 h_1 和 h_2，将每个 key 映射到两个候选位置，key

① Hyperleveldb. https://github.com/rescrv/HyperLevelDB 2016.4.2

必然存在候选位置，当插入一个 key 时，存在任意空的候选位置就插入成功. 否则，需要踢掉一个位置的 key，被踢掉的 key 同样通过 h_1 和 h_2 函数被放置到它另外一个候选位置上. 这个过程可能需要重复多次才能完成对所有 key 的安排，当重复次数过多意味着哈希表接近为满，这是一种再哈希的解决方法. 基于布谷哈希解决哈希冲突的研究系统典型的如 SILT[12] 等.

SILT 是一种针对 Flash 存储介质设计的非常高效的 KV 系统，SILT 的设计遵循以下目标：(1) 较低的读放大. 对于单个读请求，只触发 1+ε 次 Flash 读操作. 其中 ε 是很小且可配置的值；(2) SILT 能够避免随机写，支持顺序写，写放大是可控的；(3) 使用较少的内存索引. 日志结构存储不可避免地需要内存索引结构，SILT 尽量减少使用内存，SILT 将内存索引数据进行了压缩；(4) 降低内存索引的计算开销. SILT 内存索引的计算开销必须很低，使得 SILT 能够充分利用固态盘的吞吐率. SILT 的计算开销主要包括哈希表计算、排序、解压缩等；(5) 高效利用 Flash 空间. 为了提高查询和插入性能，SILT 的一些数据布局很稀疏，但是总体的空间利用率必须很高.

SILT 使用了一种混合的存储结构，包括日志结构存储，哈希存储以及顺序存储，如图 3，请求数据在 SILT 中会经历多个存储结构处理的过程，SILT 在内存占用、读写性能保持方面做了整体的平衡.

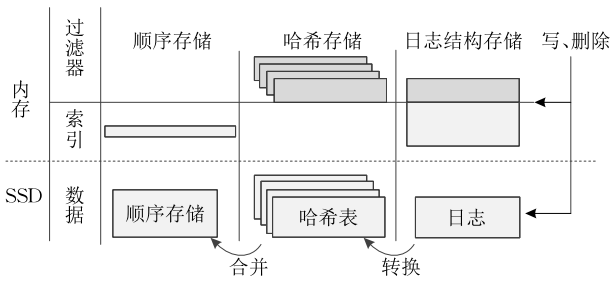


图 3 SILT 的基本架构[12]

SILT 对插入和更新请求的处理过程是：首先存储日志，把请求的 KV 数据以日志的方式追加写入 SSD，然后将 KV 数据在 SSD 日志中的偏移地址记录到哈希表，如图 4 所示. 这是一种基于哈希索引存储模型的应用，如前文中介绍过的 SkimpyStash[35].

既然是基于哈希的索引，那么必然存在哈希冲突的问题. 因此，SILT 采用了前文中介绍过的布谷哈希技术来解决这一问题，同时避免了布谷哈希的缺陷，当哈希表接近满员时，SILT 将日志存储的数据全部转化为哈希存储，进一步减少哈希冲突导致的顺序扫描日志存储的开销.

具体的操作过程如图 5 所示，在日志存储中按顺序追加插入的 K_1、K_2、K_3、K_4 项，根据哈希表中

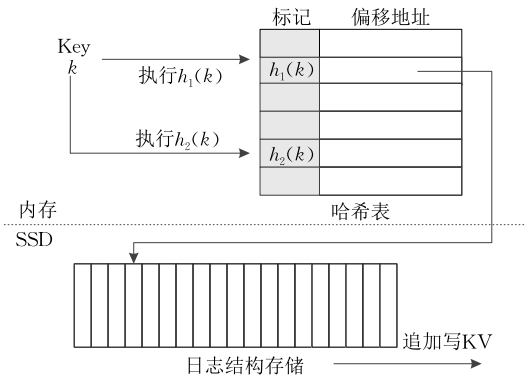


图 4 日志存储详细结构[12]

索引下标的顺序转化为哈希存储，这样哈希存储的内存中的哈希顺序跟 SSD 中的哈希存储顺序保持一致. 因此可以通过哈希直接计算获得 K 数据项在哈希存储中的位置，哈希表中只需要记录有效位标记即可索引到具体数据项而不需要额外记录数据项的具体偏移地址，跟日志存储结构相比减少了哈希表的内存占用.

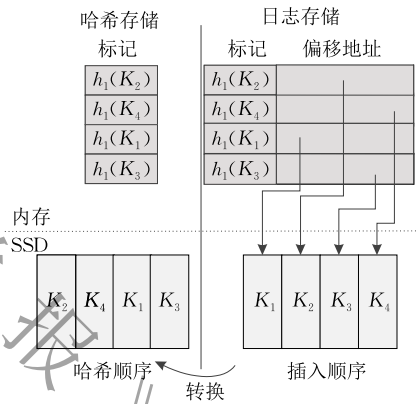


图 5 日志存储转哈希存储的过程[12]

为了进一步减少内存使用以及更好的冷热数据隔离存储，SILT 将系统中数据进行压缩形成顺序存储. 顺序存储的特点是只读不可写，因此 SILT 采用 Trie 前缀树进行数据的合并压缩. 其原理如图 6，通过扫描 key 的每比特位，根据 key 的比特位前缀形成一颗剪枝树，这样减少了冗余数据的存储开销，在面对海量数据存储时将哈希表的内存占用降到最小，同时减少了哈希冲突的可能.

哈希存储转化为顺序存储的合并操作由后台的线程定期执行完成. 如图 6 所示，假设一共有 8 个 key，每个 key 长为 5 比特位，则只需比较非阴影部分的比特位，就可以区分所有 key，而阴影部分是不必要的. 因此所有 key 被表示成上方的前缀树，如 00010 被表示为 000. 需要查询 10010 时，自顶向下，找到 100 时达到叶子结点，因为在其前面有 3 个叶子结点，key 在固态盘的索引偏移是 3(从 0 开始). 读取偏移为 3 的 key 进一步确认是否匹配. 不论 key

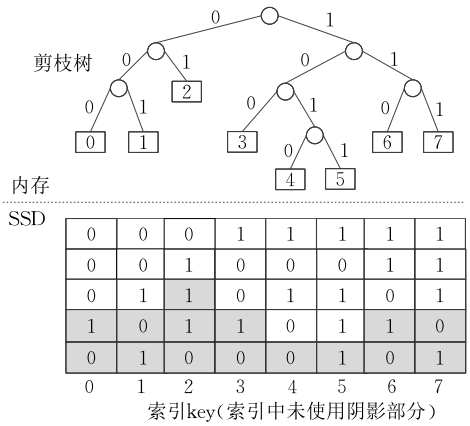


图 6　顺序存储结构[12]

是否在顺序存储中，都必然需要一次 Flash 读操作.

因此，SILT 通过整合多个不同 KV 存储系统的技术优势，在内存占用、读写放大带来的 SSD 空间占用和 I/O 开销方面做了有效的平衡. 这在当前大内存、高速存储设备 Flash、NVM 环境下仍然适用. 但 SILT 也有不足，一方面性能受后台繁杂的转换和合并操作影响，另一方面基于哈希的索引无法避免全部的哈希冲突. 对哈希索引的另外一项优化研究是布隆过滤器[16].

3.1.3　布隆过滤器

布隆过滤器是减少哈希冲突的另一项研究. 布隆过滤器技术可以在允许一定的错误率的情况下，用于判断一个元素是否属于一个集合. 但它的缺点是可能会将一个不属于集合的元素误判为属于这个集合. 布隆过滤器用于哈希索引中过滤不存在的 KV 数据，避免无用的大量读磁盘. 当插入一个元素 key 时，不直接计算该 key 的哈希值而是通过一组确定的哈希函数将该 key 映射到一个位编码上并对编码进行置位并存储. 在查找该元素 key 时，通过对 key 计算一组确定的哈希函数，通过比较编码位确认对应的 key 项是否存在. 如果存在则读取相应的记录若不存在则放弃，避免了传统哈希中哈希冲突导致的无用读磁盘哈希链操作. 这种方法跟传统的哈希相比存在一定的错误率但是能够节省大量的存储空间和 I/O 开销. 使用布隆过滤器优化哈希索引的典型研究系统如缓存哈希[17]、ChunkStash[36] 和 FlashStore[37].

缓存哈希是一种针对 Flash 设计的存储系统. 它的每个缓存哈希由多个超级表组成，每个超级表主要包含 3 部分，一个写缓存，一个实例表以及一系列布隆过滤器. 写缓存用于将小粒度的写入聚合成较大粒度的写，以充分发挥 SSD 的性能. 当把缓存中的内容写回到 SSD 时，形成一个实例，不同的实例以链表的方式存在于 SSD 中. 为了避免查找不存在的 key 带来的开销，每个实例对应一个布隆过滤器用于记录该实例中的元素. 为了提高系统的并发性能，缓存哈希将哈希的值域划分成多个超级表，各个超级表之间相互独立以达到提高可扩展性的目的.

FlashStore[37] 是 Debnath 等人提出的 KV 存储系统. 它用 SSD 作为内存和硬盘之间的缓存，这也是 I/O 五分钟原则在当今闪存时代的一个应用实例. FlashStore[37] 的系统结构如图 7 所示，为了减少小粒度写入对性能的影响，在内存中设计了一个缓存区，当缓存区写满时以较大粒度写回到 SSD. 在内存中通过哈希的方式对数据进行索引，采用布谷哈希的方式以减少哈希冲突所带来的影响. 为了减少内存的消耗，在内存中只存储 key 的标记不存储完整的 key，这样可能会产生无效的 Flash 读操作，但可以减少索引 key 对内存的消耗. 查找数据时为了减少查找不存在的 key 带来的开销，在内存中引入布隆过滤器[16] 来记录系统中的所有元素. 当需要进行空间回收时，系统采用类似于时钟算法的方式进行替换，即当访问一个 key 时，将其对应的最近位向量置为 1，回收进程通过检查内存中的最近位向量是否为 1 来决定这个 key 应该留在缓存中还是写回到磁盘.

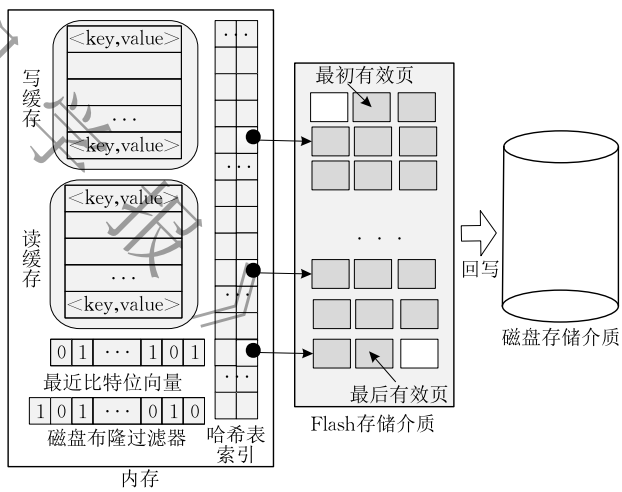


图 7　FlashStore 系统结构[37]

ChunkStash[36] 也是由 Debnath 等人所提出的针对 SSD 设计的 KV 系统. 系统的结构和思想跟 FlashStore[37] 类似，主要的区别在于 SSD 不再是硬盘与内存之间的缓存，而是作为持久化介质.

3.2　B+tree 索引技术

树是一种经典的索引数据结构，因其独特的组织结构被广泛使用，B+tree 是 B-tree 的变体，是一种多路搜索树[38].

3.2.1　基本的 B+tree 索引

B+tree 索引是在文件系统和数据库等存储系统中使用很广泛的索引，作为树型数据结构，树的根节点到存储实际数据的叶子节点之间的深度是决定

这种索引性能的核心因素.跟 AVL 树、红黑树[39]类似,B+tree 作为平衡二叉树,在插入请求的数据项时能够实现自动的调整树的深度,让树中各子树的深度保持相当水平,树的深度保持在 $O(\log N)$.

B+tree 索引系统是 B+tree 的持久化实现,不仅支持单条记录的插入、删除、更新及读取操作,还支持多条记录的顺序扫描.B+tree 索引具有以下关键特征:(1)实际的数据全部保存在叶子节点,其余节点保存叶子节点中 key 的索引信息,并且树的所有节点中 key 是顺序存储的;节点的大小一般跟存储介质上的存储单元大小对应,这样不但能够达到预读请求数据的效果而且使得树页面的出度更大;树的高度相对较小,使得 B+tree 在遇到海量数据请求时仍能保持较高且稳定的查询效率;(2)B+tree 的根节点常驻内存,查找数据时从根节点开始依次向子节点进行二分查找,每次查找 key 的 I/O 次数跟树的高度相关.为了节省 I/O,B+tree 维护着一个根节点指针提供随机查找,维护一个指向最小 key 所在叶子节点的指针来提供快速的顺序范围查询;(3)所有的叶子结点中包含了全部的 key 信息,且 B+tree 的各叶子节点拥有指向右相邻叶子节点的指针,这样避免了查找相邻叶子节点中的数据又必须从树的根节点开始查找带来的 I/O 开销提高了范围查询的效率.

但是 B+tree 也有自己的不足.当存在大量的小粒度随机写时,因反复的数据定位和更新导致 I/O 开销较大;不适合请求 KV 中 key 的重复率较高的情况;在写密集的负载下,海量高速率的数据更新会导致节点的频繁分裂与合并,这种行为在树的多层之间迭代传递带来的开销成为系统瓶颈[8].当前出现众多对 B+tree 的优化研究,包括针对新型存储介质 SSD、NVM 的 B+tree 索引优化[18,40-43],在多核环境下对 B+tree 索引的并发控制优化以实现多核高扩展性[8,44-47].在这里介绍具有代表性的研究 LA-tree[18]和 Bw-tree[8].

3.2.2 LA-tree

LA-tree[18]是对传统 B+tree 的一个优化工作,为了减少 B+tree 页面更新操作产生的额外 I/O,为树的节点设计了一个本地缓存.LA-tree 的结构如图 8 所示.

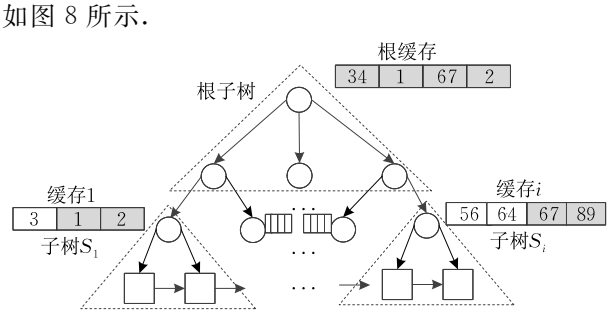


图 8 LA-tree 结构[18]

核心思想是在 B+tree 的中间节点设计一个缓存,对中间节点的更新操作暂时不将它传递到真正的叶子节点,缓存区会一直缓存节点的更新直到缓存区满员,然后将缓存区中的请求向叶子节点方向的下一个中间节点缓存区中层层传递,直到到达叶子节点,最后将缓存区清空.这种缓存的设计使得对节点的更新操作以批处理的方式进行,在 B+tree 中大大减少了页面节点的频繁更新带来的频繁读取磁盘的 I/O 开销,将多次更新聚合为一次使得写性能大大提升.LA-tree 的这种将传统 B+tree 中小粒度更新转化为大粒度更新的设计减少了随机更新的频率,更加有利于 SSD 等新型存储介质的硬件性能发挥.

但这种增加额外节点缓存的设计,缓存的大小成为权衡性能的关键.如果缓存的容量过大,一方面消耗的内存增加,另一方面过大的缓存会影响系统的读性能;如果缓存过小,并不会起到节点数据缓存的作用.因此,FA-Tree 的工作中设计了一套缓存大小在运行时根据负载进行动态调整的策略.

3.2.3 Bw-tree

Bw-tree[8],是微软游戏数据库 Microsoft SQL Server Hekaton 的索引引擎,它是针对多核环境对 B+tree 做的优化,支持多线程无锁地并发更新和访问各节点.Bw-tree 的架构如图 9 所示.

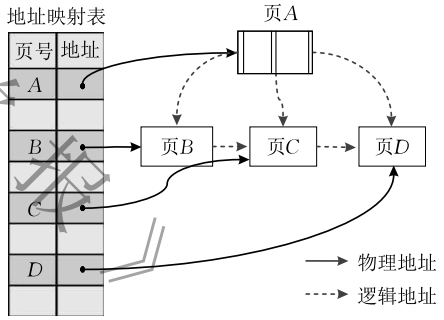


图 9 Bw-tree 结构[8]

Bw-tree 的核心思想是将 B+tree 中的每个节点抽象为虚拟节点,将由一个地址映射表来保存虚节点地址到实际物理节点地址的映射.Bw-tree 的设计具有三个突出的特点:(1)虚拟父子节点之间,虚拟兄弟节点之间均使用逻辑指针连接.逻辑地址是一个目标节点的页号,可以根据地址映射表将其转换成对应的物理地址;(2)虚拟节点的大小是动态变化的,各节点大小各异;(3)对节点的更新采用异地增量的方式进行,将对整棵树的小粒度随机更新聚集到较少的缓存行中,避免了高并发随机地原地更新树的各个页节点导致的 CPU 缓存行的频繁替换,提高了 CPU 缓存命中率.同时减小了 B+tree 中叶子节点的频繁分裂向根节点层层传递的开销.在 Bw-tree 工作中,关于对 B+tree 的并发控制机制优化的更多内容请参见 4.3 节.

另外，如 NV-Tree[41] 等工作在 B+tree 的中间兄弟节点之间增加横向指针，以提高查找性能. 同时结合新型存储介质 NVM 优化节点合并分裂的开销.

3.3　LSM-tree 技术

Log-Structured 一词来源于经典论文《The Design and Implementation of a Log-Structured File System》[48]. 将内存中的缓存数据以记录日志的形式追加写入到存储介质上，这种思想的初衷为了减少机械磁盘悬臂的频繁机械运动，将小粒度的随机写聚合成大粒度的顺序追加写，提高 I/O 利用率从而提升系统. 这在当前 Flash 为存储介质的系统中仍适用.

日志结构合并树 LSM-tree[14] 的动机跟 Log-Structured 文件系统类似. LSM-tree 通过使用一个基于内存的组件 C_0 和一至多个基于磁盘的 $(C_1, C_2, \cdots, C_K)$ 组件算法，对索引变更进行延迟及批量处理，并通过归并的方式高效地将更新迁移到磁盘.

LSM-tree 的核心思想就是将对数据的修改增量保持在内存中，达到指定的大小限制后将这些修改操作批量顺序地写入磁盘来达到最优的写性能，因为这会大大降低磁盘的寻道次数，一次磁盘 I/O 可以写入多个索引块. 但是，在提升写性能的同时会牺牲读性能.

3.3.1　LSM-tree 原理

数据的写入. LSM-tree 不需要每次有数据更新就必须将数据写入到磁盘中，而可以先将最新的数据驻留在内存，当内存中的数据超过一定阈值时，再将内存中的数据归并排序并合并追加到磁盘队尾，如图 10 所示.

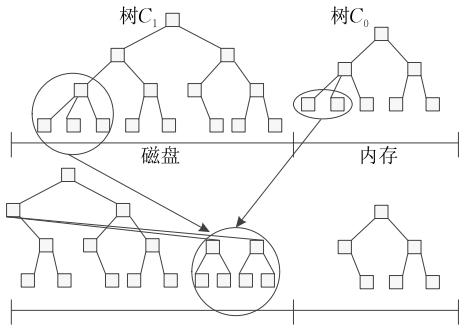


图 10　LSM-tree 结构[14]

合并操作会从左至右遍历内存中树的叶子节点与磁盘中树的叶子节点进行合并，当被合并的数据量达到指定大小时，会将合并后的数据持久化到磁盘，同时更新父亲节点到叶子节点的指针.

但是这样的设计存在的问题是，如果数据量过于庞大，相应地磁盘中的树会越来越大，导致的后果是合并的速度越加变慢，从而导致写操作被合并过程阻塞，写性能受到影响.

LSM-tree 对以上问题的解决方案是对整个数据集进行分层，将数据集以树为组件进行划分，每层中设置不同个数的树组件. 层次越低，数组件的个数越多数据集越大. 假设每层中只设置一个树组件，内存中的树组件为 C_0，磁盘中的树组件按层次依次为 $C_1, C_2, C_3, \cdots, C_{k-1}, C_k$，那么合并的顺序是 $(C_0, C_1), (C_1, C_2), \cdots, (C_{k-1}, C_k)$. 这样合并速度较快，对写操作的阻塞大大减少.

因此 LSM-tree 有以下的性能特性：(1) 如前所述写入操作首先在内存中进行，在内存树中聚集多个写操作的数据并且这棵内存树不会很大，这样后续的合并过程速度将很快；(2) 合并操作会顺序地将内存中聚集的数据一次性写入磁盘，这样将小粒度的随机写转化为大粒度的顺序写，提高了 I/O 利用率；(3) LSM-tree 减少了写操作的 I/O 的次数，对于更新操作，直接在内存中完成不占用磁盘 I/O，而 B+tree 索引结构每进行一次数据更新至少需要两次磁盘 I/O.

数据的读取. LSM-tree 的读取操作需要在内存以及各个层级磁盘文件中按照从新到老依次查找，代价较高. 但是，LSM-tree 对于读性能的保障也有自己的设计特点：(1) LSM-tree 在写入时不断地在做内存数据跟磁盘数据的合并来对磁盘中已有的记录进行整理，从而删除一些无效的记录，减少数据规模和层级的文件数量. 这样的合并操作可以达到优化读性能的目的；(2) 对缓存和内存的访问速度远快于磁盘，而读性能提升主要依赖于 CPU 缓存和内存缓存的命中率. 在低缓存命中率的情况下如果产生读取磁盘的操作将大大影响读性能；(3) 写入操作是在预定义的内存缓存中完成，在没有合并之前不占用磁盘的 I/O，这有利于为读操作提供更多的 I/O 机会提升读取效率.

对 LSM-tree 读性能的一般性优化方式一方面是布隆过滤器，通过使用带随机概率的位图快速确认指定的数据是否在要查找的数据集中，于是不用为了查找一个不存在的数据，对磁盘中数据集进行二分查找. 因而效率得到了提升，但付出的是空间代价；另一方面是合并优化，小树合并为大树. 因为小树对读性能来说有很大的 I/O 影响，所以有个后台线程不断地将小树合并到大树上，这样大部分的旧数据查询也可以直接使用 $O(\log N)$ 的查询找到，不需要再进行 $(M/m) * \log N$ 的查询.

LSM-tree 通过采用批量转储技术来避免磁盘随机写入，不论对于机械磁盘还是闪存，是一种高效的数据索引选择，尤其是对写性能有较高需求的系统. LSM-tree 索引被广泛应用于互联网本地数据库和分布式数据库的后台存储索引引擎，本地系统如谷歌的 LevelDB，Facebook 的 RocksDB，

HyperDex[49]的 HyperLevelDB,分布式系统如谷歌的 BigTable[3],Apache 的 Hbase、Cassandra[5],雅虎的 PNUTS[6]等.在后续章节中详细介绍基于 LSM-tree 索引的本地系统.

由于当前大多数基于 LSM-tree 的 KV 型本地存储系统是基于开源系统 LevelDB 进行研究设计的,因此 LevelDB 是其他 KV 型本地存储系统的研究基础.在此先介绍 LevelDB 的系统架构和基本实现机制.

3.3.2 LevelDB 核心架构

LevelDB 是由谷歌公司的 Jeff Dean 和 Sanjay Ghemawa 设计实现并开源的高性能 KV 型本地存储系统.LevelDB 的设计思想和实现技术跟谷歌的著名分布式数据库系统 BigTable[3]单个 Tablet 节点上的存储系统类似.

LevelDB 是一个典型的基于 LSM-tree 索引技术实现并优化的轻量级 KV 系统.在 LevelDB 系统诞生之后,对 LSM-tree 优化的相关工作基本都是基于它来实现的.接下来简要介绍其架构及它对 LSM-tree 的优化,后续章节中详细介绍其他工作对 LevelDB 的具体优化.

先来介绍 LevelDB 系统架构,如图 11 所示.

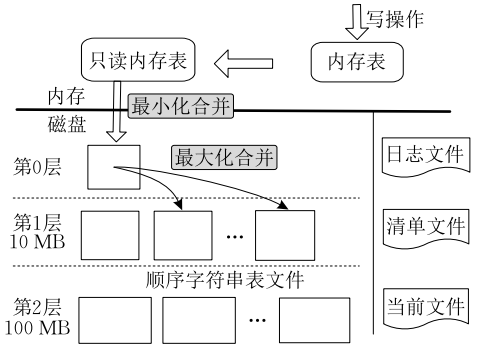


图 11 LevelDB 系统架构

LevelDB 由内存和磁盘上的文件混合构成,其核心的六个文件模块是:内存表、只读内存表、日志文件、清单文件、当前文件和顺序字符串表文件.在内存中,内存表负责存储新插入的键值对.当内存表被写满时系统将其转换成只读内存表,并建立新的内存表来接收写请求.通过 LevelDB 的最小化合并过程,只读内存表被写入磁盘的第 0 层,形成顺序字符串表文件.

在磁盘中,日志文件存储最近的数据更新以做故障恢复,采用追加的方式将每次的更新数据写到日志文件的末尾,每个日志文件对应当前的内存表,更新操作先写入日志文件然后更新内存表.当内存表被写入顺序字符串表文件后,相应的日志文件会被删除;清单文件存储当前数据库元数据,如每层包含哪些顺序字符串表文件,每个文件中 key 的范围以及其他的元数据信息如日志文件、压缩指针等,清单文件又称为描述符文件.当前文件是一个简单的文本文件,记录当前使用的清单文件名,通常通过判断当前文件是否存在来判断数据库是否已经创建;顺序字符串表文件用来存储数据和索引,该文件分为 7 层,第 0 层文件由只读内存表合并生成,称之为最小化合并;第 1 到 6 层文件由顺序字符串表文件合并生成,称之为最大化合并.

LevelDB 当然还包括内存缓存模块及磁盘上的一些临时文件等,这里只介绍最核心的设计.LevelDB 更多的是对 LSM-tree[14]的系统实现,但同时它对 LSM-tree 内存模块中的数据结构进行了优化,核心是基于跳表[50]数据结构的内存表实现.

跳表是一种 B+tree 的替代数据结构,但是又跟 B+tree、红黑树不同.跳表对节点高度平衡的实现是基于一种随机化算法的,这样跳表中的插入和删除的操作是比较简单的同时保证了 $O(\log N)$ 的查询时间复杂度.跳表的核心思想是,通过归并排序后链表中的数据是顺序的,并且每一个节点除了存储指向前后节点的指针外,还存储了指向前面第二个节点的指针,那么在查找一个节点时,仅仅需要遍历 $N/2$ 个节点即可.其本质是一种通过空间来换取时间的一个算法,通过在每个节点中增加了额外的指针来增加查询跳跃步长,从而提升查找的效率.

LevelDB 在设计上也呈现一些特点,后续针对 LSM-tree 的研究工作跟这些特点密切相关.这些特性包括:(1)并发控制方面.对于读线程来说,基于多版本的并发控制机制使得读线程并不被阻塞,而对于内存表中数据的写入,设置了内存表的全局锁和写线程队列,只允许单线程写;(2)只读内存表被合并到磁盘.内存中只有一个只读内存表,被后台单线程合并到磁盘,合并的速率直接影响写线程更新内存表的速率;(3)磁盘上不同层级多个顺序字符串表的合并过程由后台单线程执行,合并的速率也直接跟写线程更新内存表的速率相关.

虽然从形式上来看它跟 LSM-tree 有较大区别,但它的数据结构实际上相当于一个多组件的 LSM-tree.对于合并操作来说,是合并顺序排列 key 的重叠区间且去除冗余并做持久化存储,跟 LSM-tree 中的滚动合并的过程完全相同,同样是将缓冲的随机更新操作转化为对带宽利用率更高的顺序写入操作,这种读写模式的优化对于 SSD 存储介质来说是非常适宜的.跟 LSM-tree 类似的一种索引结构研究是 FD-tree[51],在此对此工作做简要介绍.

3.3.3 FD-tree

FD-tree[51]是一种 LSM-tree 的实现方式,同时

通过 Fractional Cascading[52]技术来提高其读性能. FD-tree 由一系列顺序排列，连续存放的数据记录组成，每一个序列等价于 LSM-tree 中一个树型组件的叶子节点. 与 LSM-tree 中不同的是除了数据记录以外，FD-tree 还在序列中插入了被称为“防护”的节点. 通过“防护”节点的定位作用，在进行读操作时如果前一个树型组件没有查找到需要的数据记录，则对下一个树型组件的检索不需要从头开始，“防护”节点将指向合适的页面尽可能减少读取的数据量. 对于这些“防护”节点，可以在两层次的序列进行合并操作时生成. 具体而言，“防护”节点将指向下一层次序列中 key 值不高于它本身的最后一个页面，于是当本层次序列的查找操作终止于该“防护”节点的情况下，对下一层次的查找从其指向的页面开始即可，而无需从头开始.

FD-tree 利用 Fractional Cascading 技术，将 LSM-tree 的读操作算法复杂度降低至与 B+tree 同等的水平. 另外对 FD-tree 并发控制的优化[53]实现了多核的高扩展性，提升了读写性能.

除外，类似于 B+tree 索引又跟 LSM-tree 索引技术相似的另一项研究是 COLO-tree[54]即一个缓存忘却算法树，该算法在不需要明确知道存储 I/O 中数据传输粒度的情况下可以使用同一套逻辑进行高效处理，能够减少 I/O 代价，具有较好的性能表现，对应的实际研究系统如 Tokudb[55-56].

综上所述，对索引技术的研究是在三大基础索引引擎上进行展开的，而这三大基础索引各有优势同时也有自己的不足，如表 3 所示.

表 3　三种经典索引引擎

索引	优势	缺点	研究现状
哈希	点查询高效，时间复杂度为 $O(1)$	哈希冲突导致读放大；不支持范围查询功能	布谷哈希减少哈希冲突；布隆过滤器减少读放大
B+tree	支持范围查询；保证了相对较好且稳定的点查询性能，时间复杂度为 $O(\log N)$	存在大量小粒度随机写，I/O 开销较大；节点的频繁合并和分裂具有较高的性能开销；页面粒度的锁，制约着系统并发能力，多核环境下影响系统扩展性	LA-tree 设计额外的页面节点缓存减少随机 I/O；Bw-tree 设计大小可变的弹性页面节点，以降低节点合并和分裂的频率，同时使用多版本和无锁的并发控制使系统具有高扩展性
LSM-tree	通过将随机写转化为顺序写，降低了 I/O 开销；基于多版本的读并发控制，在多核环境下支持多线程的并发读而无需阻塞等待写线程释放锁；跳表提供了跟 B+tree 同等的查询时间复杂度，$O(\log N)$	频繁的数据合并具有较大的读放大；范围查询性能差；读性能不稳定；读性能依赖于内存缓存、持久化存储介质上多层顺序字符串表的命中率，还依赖于合并过程；内存表的全局写锁制约着多核扩展性	FD-tree 使用 Fractional Cascading 技术优化随机读；bLSM-tree 使用布隆过滤器减少读放大；LOCS 结合新型存储介质 SSD 设计了多线程的合程；cLSM-tree 使用无锁并发控制代替内存表中的全局锁，实现写性能的高扩展性

在设计数据库存储系统时，并没有理想的索引技术选择，而是根据应用负载的功能和性能需求来选择不同的索引引擎，同时优化基础索引以满足更高的性能需求.

4　并发控制研究

4.1　基于锁的系统

锁是在数据库系统中常用的并发控制机制，典型的 KV 系统如 BerkeleyDB. BerkeleyDB① 是一个高性能嵌入数据库，可以保存任意类型的键/值对，而且可以为一个键保存多个数据. 同时 BerkeleyDB 也具有关系型数据库的特性，保证了事务的 ACID 语义，支持 SQL 查询. 但在很多场景下 BerkeleyDB 作为一个轻量级的底层 KV 存储引擎组件对外提供简单的 KV 接口使用，如亚马逊的分布式存储系统 Dynamo[2]. BerkeleyDB 作为一个介于关系数据库和内存数据库之间的数据库，它独特的设计让它在功能和性能上成为众多应用的选择，一方面使用简单的函数调用接口完成所有的数据库操作，代替数据库系统中经常用到的 SQL 语言，避免了对结构化查询语言进行解析和处理所需的开销；另一方面因为应用程序和 BerkeleyDB 运行在相同的进程空间，数据操作时可以避免繁琐的进程间通信，因此耗费在通信上的开销将降低. 同时支持数千的并发线程同时操作数据库，但在多核环境下数据库的性能跟并发控制机制直接相关[8].

从并发控制的角度来讲，针对不同的索引数据结构采用不同的并发控制机制. BerkeleyDB 选择以 B+tree 为索引. 如前文介绍它是一个平衡树，关键字顺序存储并且其结构能随数据的插入和删除进行动态调整. 对于 B+tree 索引数据结构而言，传统的采用基于锁的并发控制机制，在 BerkeleyDB 系统中也不例外. BerkeleyDB 在 B+tree 上基于页面锁的并发控制机制和使用传统空闲队列管理页面空间. 基于页面锁的并发控制机制，在高并发负载下一方面严重阻塞线程. 另一方面频繁地加锁解锁增加了线程上下文切换的开销[7]，严重制约着系统的多线程并发能力. 在 B+tree 上针对锁开销的优化的并发控制技术是基于多版本和无锁的并发控制机制，典型工作如 Bw-tree[8]、Btrfs[57]等. 另外，基于空闲

① Berkeley DB. http://en.wikipedia.org/wiki/Berkeley_DB 2016.4.11

队列的页面管理机制存在大量原地更新,这不利于多线程的高并发,同时大量的小粒度随机更新造成频繁 CPU 缓存行的替换和多次访问磁盘的高 I/O 开销.

然而 LSM 系列系统在写并发控制上同样采用了全局锁,但读并发控制典型地采用了多版本的并发控制,这是对锁机制的一个优化.多版本的读并发控制大大提高了多线程的读并发能力,读性能有了很好的扩展.

4.2 基于多版本并发控制的系统

如上所述,基于锁的并发控制在多任务高并发执行时存在频繁地线程阻塞和线程上下文切换,严重制约着系统的扩展性.基于多版本并发控制技术的系统研究在不断深化,典型的工作如 LSM 型 KV 系统的研究[7,19-20].在 LSM 型系统中,每一个内部 key 由用户 key 和全局递增且唯一的序列号组成,并且内部 key 无重复.对数据库的更新操作只有写,更新、删除操作均是写操作并且不存在原地更新.对同一条记录的多次更新形成多个版本记录,多版本的并发控制机制消除了锁给读线程带来的频繁阻塞和上下文切换开销,在写线程对数据更新时允许多个读线程并发读,大大提升了系统的读性能.同时这种异地的更新有利于保护 CPU 缓存行地址,提高了多核 CPU 缓存的命中率.对小粒度的 KV 对象来说,这进一步优化了读性能.

另一种类似多版本的读并发控制技术是 CoW 即 Copy-on-Write 技术.基本思想是,在对数据对象执行读操作的时候,内存数据不发生任何变动直接执行读操作,而在对对象执行写操作时,将对象复制一份到新的内存地址,对新副本进行更新.此时数据索引的指针仍指向旧的对象,这样写线程更新的同时允许了多个线程进行非阻塞的并发读.当新对象的更新完成时将索引指针改为指向新的对象,旧的对象成为了一个历史版本,可以定期做快照进行备份或定期做垃圾回收进行删除.这种 CoW 技术广泛应用于文件系统,典型的工作如 BtrFs[57].在数据库系统中 CoW 的高效在于可以减少写共享资源引起的读阻塞,能够很好地支持多线程的读并发且可以保证数据的一致性.这样达到了读写分离,是一种在当前多核的硬件环境下提高系统并发性能的选择,典型的工作如 HyPer[58],利用 CoW 技术使 OLAP 跟 OLTP 共享虚拟内存地址空间并维护一致性,避免了加锁实现了高并发.在 Dorje[59] 工作中也尝试采用了 CoW 的并发控制,在读多写少的负载下比 BerkeleyDB 等系统具有更好的性能表现,但在写密集负载下因它本身的内存拷贝和缓存占用,写性能并不如 BerkeleyDB 等系统.因此 CoW 的并发控制机制是在读多写少的负载情景下的一种理想的读并发控制选择.

4.3 基于无锁并发控制的系统

多版本并发控制是针对读优化的并发控制机制,对于写操作来说,传统的系统中仍采用基于锁的并发控制.如上所述,锁的并发控制限制着多核的并发,严重制约着系统的扩展性,不能充分使用系统资源.但是当前随着 CPU 技术的不断成熟,出现了 CPU 硬件级指令支持的可供应用层使用的原子操作,如 CAS,使得多线程能够对共享数据进行并发的原子更新而无需加锁即 Lock-free 成为现实,使得系统具有很高的并发能力,从而优化了写性能.

CAS[21-22],即 Compare and Swap(void * p, Any expectedValue, Any newValue)操作,它涉及到三个操作数:内存值、预期值、新值.当且仅当预期值和内存值相等时才将内存旧值更新为新值.这样处理的逻辑是,线程首先检查某块内存的值是否跟之前读取时的一样,如不一样则表示期间此内存值已经被其它线程更新过,舍弃本次操作.否则说明期间没有其它线程对此内存值进行过操作,可以把新值设置给此块内存.CAS 操作是通过 CPU 的 CMPXCHG 指令来完成的,利用了 CPU 的硬件锁来实现对共享资源的串行访问,CAS 操作不像加锁,不需要进入内核不需要切换线程,是一种乐观的并发控制机制.

在当下处理器多核及大内存环境下基于 Lock-free 即无锁或锁无关并发控制的系统研究成为热点[7-8,46,60-61],典型工作如 Bw-tree[8]、cLSM-tree[7].

Bw-tree 是针对多核、大内存和 Flash 存储介质的硬件环境设计的.为了更好地使用系统的硬件资源,基于 B+tree 索引的 KV 系统在设计时主要面临以下挑战:(1)多核 CPU 提高了多线程的并发度,但随着并发的增加,B+tree 中基于页面节点的加锁对多线程造成频繁阻塞,限制了系统扩展性.对应的,Bw-tree 采用无锁技术,一方面解决了多线程对共享资源的并发读访问,另一方面允许多线程原子的并发更新共享资源数据.但是在多核环境下,高并发带来的问题是来自多个核的线程访问共享数据时会导致 CPU 缓存不一致问题从而降低 CPU 缓存命中率,尤其在多芯片或 NUMA 架构下维护缓存一致性需要付出更多的通信代价;(2)页粒度的原地更新导致高 CPU 缓存失效.当多线程频繁地随机原地更新不同页面,造成之前 CPU 缓存行中内容的频繁替换,导致较低的 CPU 缓存命中率.对应的,Bw-tree 采用了异地增量更新,一方面允许了多线程并发读访问,另一方面避免了页的原地更新,当多线程频繁地随机更新不同页面时保护了之前缓存行的地址,提高多核处理器缓存命中率;(3)当下 SSD 被广泛用作持久化存储介质,SSD 虽擅长于随机读和顺序读写,但是对随机写 SSD 需要写前先擦除.小粒度随机写不利于 SSD 性能发挥.因此

Bw-tree 采用日志结构存储，对写入的数据在内存做一次聚合，然后以追加的方式写到 SSD. 将小粒度的随机写转化成大粒度的顺序写，减轻了 SSD 的 FTL 的垃圾回收开销.

Bw-tree 是典型的 B+tree，提供 $O(\log N)$ 的访问性能. 它有三个特殊的优化设计：(1) 使用页号来识别一个页，通过查询映射表将页号转化为逻辑地址来访问页；(2) 页的大小是灵活的，没有限制页的大小，并且兄弟节点之间通过水平向右的指针依次连接；(3) 页的增长通过增量记录来实现，更新不会是原地更新，而是通过创建一个增量记录来描述更新，并放置在当前页之前.

更新一个节点页的过程如图 12 所示：(1) 创建一个增量记录，指向当前页；(2) 获取该页在映射表上的项；(3) 将增量记录的逻辑地址作为页的新逻辑地址，使用 CAS 更新映射表中对应项的地址.

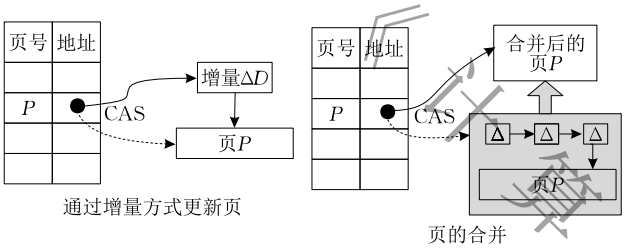
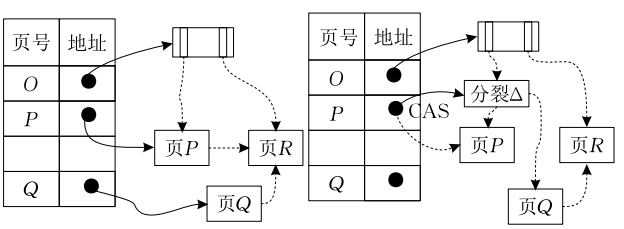


图 12　使用 CAS 更新映射表[8]

所有的增量都包含全局的序列号，即使用多版本的读并发控制. 随着增量链变长，查询性能会降低，为了避免这种情况，定时执行页合并，将增量记录和基本页重组生成新的页，页合并在增量链长度超过阈值时触发. 页合并时线程需要做以下操作：(1) 创建一个新的页，然后把最新的记录版本写入到页中；(2) 使用 CAS 插入新的页. 若成功则请求回收老的页，若失败释放新的页. 失败并不会重试，后面的线程会继续执行直到完成.

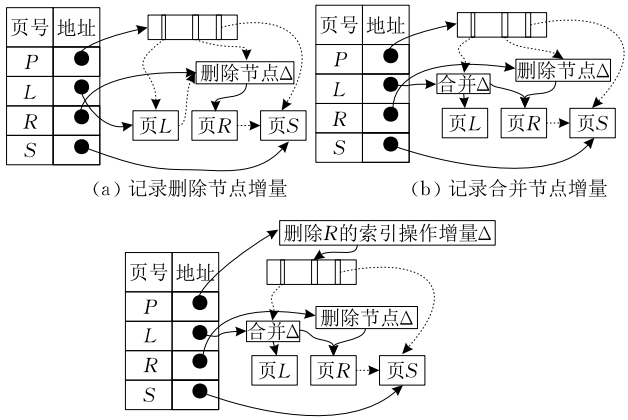
Bw-tree 是典型的对 B+tree 索引的优化工作，因此它也有节点分裂和合并的特性. 节点分裂是由一个后台线程在页的大小超过系统设置的阈值时进行分裂. 如图 13 所示，整个过程分为 2 个阶段，先在叶子上做分裂，然后更新父节点指针，一直向上递归直到完成分裂，分裂的具体过程跟 B+tree 的分裂一样，在此不做赘述.



(a) 先创建兄弟节点Q　　(b) 接着插入分裂增量

图 13　Bw-tree 节点分裂[8]

如图 14 所示，节点合并由 3 个步骤组成：(1) 删除节点. 如要合并 R 先在 R 上标记删除，添加一个增量记录，表示 R 已经被删除. 一个线程读取 R 时如果碰到删除节点的增量记录，就会将该增量应用到左边的兄弟节点上；(2) 合并孩子节点. 在 L 上添加一个节点合并增量物理的指向 R，这样就表示 R 中的数据是在节点 L 中. R 的存储状态被转换成了 L，只有当 L 被合并的时候 R 才被回收. 当对 L 查询时，变成对树的查询，可以从 L 开始访问 L 跟 R 的 key；(3) 更新父节点. 通过使用删除增量删除父节点上关于 R 的索引信息和 L 的 key 信息.



(a) 记录删除节点增量　　(b) 记录合并节点增量

(c) 记录删除被合并节点的索引操作增量

图 14　Bw-tree 节点合并过程[8]

Bw-tree 使用额外的内存缓存来刷新内存与 SSD 之间的页. 映射表给 Bw-tree 提供抽象的逻辑页，Bw-tree 通过页号来构建一个 B+tree. 如图 9，在映射表中是内存地址或 SSD 的偏移. 如是 SSD 偏移那么就从 SSD 读入到内存中，所有涉及到内存的操作都是通过 CAS 来完成. 为了跟踪 SSD 中页的版本和位置，使用刷新增量来进行记录. 同时记录了已被刷新的修改，之后的刷新只对应增量. 若刷新页成功，则刷新增量就会包含新的刷新偏移，页的状态会被设置为已刷新.

Bw-tree 在数据放置方面采用了日志结构存储，页的批量顺序写入大大减少了 I/O 次数. 当更新页的时候，只需要更新增量部分，通过增加刷新缓存的页数量来减少 I/O 的消耗. 但是这样会带来读开销，因为页中的数据不是连续被存放的.

在并发控制方面，Bw-tree 是一个基于 Lock-free 写并发控制，基于多版本的读并发控制的高扩展性系统，在多核的硬件环境下是一种全新的系统设计.

cLSM-tree[7] 是另一项典型的基于 Lock-free 并发控制的研究工作，对它的详细叙述请参见 7.2.2 节.

4.4　基于数据分区并发控制的系统

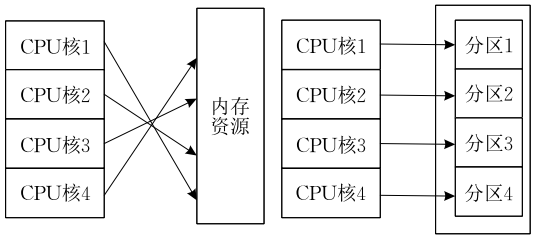
在当前多核环境下，系统的并发控制直接跟系统的性能相关. KV 系统要充分发挥当前处理器的

高速处理能力,就需要良好的线程同步机制.除基于锁、多版本并发控制、lock-free 外的一种特殊的并发控制机制是对共享内存资源做分区,将不同的 CPU 核心跟指定的分区进行绑定,以减少或消除共享内存的锁同步带来的开销,同时通过使用已有技术 RDMA、DPDK 实现低延迟的访问[62-66],最新的研究工作典型的如 MICA[67].

MICA 的全称为 Memory-store with Intelligent Concurrent Access. MICA 是一个全内存 KV 存储系统,因其独特的并发控制机制、网络处理机制和数据结构设计,具有较高的性能表现. MICA 并没有实际的应用系统,只是针对小粒度 KV 研究的缓存系统原型,没有可靠性保证机制.

MICA 的动机是在多核环境下减少多线程对 KV 数据结构的并发扩展性限制,减少或消除内核对任务线程的阻塞,提高系统的扩展性. Bw-tree[8]、cLSM-tree[7]采用了 Lock-free 的技术实现了多线程写的并发控制,基于多版本的读并发控制. 而 MICA 采用了对多核共享的内存进行分区的方式,使得多核对各自的目标资源区域进行并行的访问. 同时为了更加有效地并行访问分区的数据,MICA 使用 Intel 的 DPDK① 技术,通过网卡设备避免 Socket I/O 直接将客户端的请求映射到特定的 CPU 核心. 这样的设计,一方面避免了 Socket 数据传输,实现了网络数据包在内核中的零拷贝,消除了内核空间和用户空间在网络栈上的高开销. 对于内存 KV 系统来说,客户端和服务器之间低延迟的网络通讯是至关重要的. 另一方面,将客户端的请求直接映射到特定的 CPU 核心,实现了 CREW(Concurrent Read Exclusive Write)即读共享写独占的并发控制,达到共享锁的效果. 同时也实现了 EREW(Exclusive Read Exclusive Write)即独占读和写,达到了独占锁的效果,只是跟独占锁不同的是,EREW 允许了多线程并行的独占各自的共享数据,不阻塞多个执行线程而实现了真正意义的并发.

MICA 主要由三部分构成分别是并行访问、请求分发和 KV 索引数据结构. 第一部分是客户端对服务器的并发访问,对服务器内存中的数据结构进行并行访问的模型如图 15 所示.



(a) 多核共享访问 (b) 多核独占访问

图 15 并行访问模型

图 15(a)有更好的负载均衡,而限制了 CPU 的扩展性,如线程同步、跨 NUMA 节点的延迟和缓存失效;图 15(b)有更好的 CPU 扩展性,但是在倾斜的负载下有较低的性能. 第二部分是 MICA 服务器中的请求分发机制. MICA 使用 Intel 的 DPDK 技术绕过了 Socket 的 I/O 过程,DPDK 这种特殊的技术支持应用层使用软件的方式去控制网卡,最小化数据传输的开销. 在 NUMA 系统中,每个 CPU 核心跟 NIC 通过 PCIe 总线连接,MICA 中每个 CPU 核和 NIC 仅能访问存储在它们相应的 NUMA 域中的缓存数据. 基于客户端的分区对请求的 key 进行分发,不同的 key 将分发到各自的分区核上进行处理.

DPDK 是 Intel 提供的提升数据面报文快速处理速率的应用程序开发包,它主要利用以下几个方面的支持特点来优化报文处理过程,从而加快报文处理速率:(1)使用大页缓存支持来提高内存访问效率;(2)利用 UIO 支持,提供应用空间中驱动程序的支持,即网卡驱动是运行在用户空间的,减少报文在用户空间和应用空间的多次拷贝;(3)利用 LINUX 亲和性支持,把控制面线程及各个数据面线程绑定到不同的 CPU 核,节省了线程在各个 CPU 核上的来回调度;(4)提供内存池和无锁环形缓存管理,加快内存访问效率. 第三部分是对 KV 数据结构的具体操作,跟普通的哈希数据结构类似,除了哈希表外设计一个存储数据的循环队列,作为追加写 KV 数据的日志. 同时 MICA 设计了自己的 KV 内存管理数据,包括内存的分配回收与索引和缓存的管理.

MICA 对缓存和存储单独管理,每一个分区缓存由循环队列日志和有损的并发哈希索引表构成. 如图 16 所示,每一个日志项由 key 的初始化大小值、key 的哈希值、KV 的长度以及该日志项的过期时间构成. 有损哈希是因为当向该循环队列日志中插入新的 key 时,该队列根据 FIFO 原则将剔除最老的数据项即日志头部的 key,对缓存中数据项进行"持久化"保存,在内存中形成哈希链.

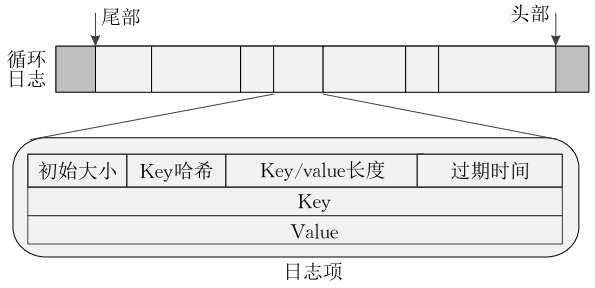


图 16 MICA 结构[67]

① Intel. Intel Data Plane Development Kit (Intel DPDK). http://www.intel.com/go/dpdk 2016,4,12

基于分区的并发控制机制中，控制好负载 key 对分区的倾斜度成为关键. 在 MICA 中仍存在问题，各个分区中的频繁地原地更新会增加缓存行替换频率，异地增量更新是一个好的解决方案.

总之，在多处理器多核环境下，传统的锁并发控制在多线程高并发下导致线程的严重阻塞和较大的线程上下文切换开销，影响多核扩展性. 在索引数据结构上选择最佳的并发控制机制能最大程度发挥多核的硬件性能. 表 4 总结了典型的并发控制策略及相关的系统研究.

表 4　关键的并发控制机制

并发控制机制	优点	缺点	系统研究现状
锁	低更新错误；适用范围广	频繁的线程阻塞和昂贵的上下文切换开销	对数据分区以减少竞争，如 MICA
多版本	多线程读的高并发；读线程不受写线程独占写锁的阻塞	占用更多的存储空间；引入额外的垃圾回收操作开销	基于多版本的系统研究典型的如 LSM 型系统，获得读性能高扩展，高效的数据恢复
Lock-free	支持多写线程高并发. 不被阻塞也不发生线程上下文切换	比锁更高的更新错误率；适用范围有限，支持更新的最大内存长度为 16 字节	Lock-free 写并发控制的系统研究，典型地包括 cLSM-tree 和 Bw-tree，在当前多核环境下具有出色的写性能
数据分区	减少多线程对单个共享内存数据区的竞争，减少了锁开销	负载请求在分区上的倾斜直接影响并发效率；分区内部仍存在多线程竞争	典型的研究系统如 MICA，它使用数据分区减少多核的竞争，同时使用 DPDK 技术实现并行访问模型

在设计数据库存储系统时，基于多版本的读并发控制和基于 Lock-free 的写并发控制成为理想的并发控制机制，在多核环境下提供最佳的性能.

5　事务日志扩展性研究

当前通过使用新型存储介质 NVM 作缓存，进行多层数据恢复日志的热点研究主要在两个方面：一方面，使用 NVM 以高速比特寻址并且非易失的性能特性，较长时间缓存 WAL，减少跟持久化存储介质之间的 I/O，减少写密集负载下 WAL 的 I/O 瓶颈，如 Hwang 等人提出的基于 NVM 的两层日志[9]；另一方面，在多核环境下除了存在事务日志的 I/O 瓶颈外，WAL 单一的集中式日志，阻塞了多线程的并发更新，出现日志扩展性瓶颈问题. 典型的，多伦多大学的 Wang 等人提出了采用分布式日志[10]，使得多线程并发更新 WAL 达到较好的扩展性，解决了整个系统在日志上的瓶颈.

5.1　NVM

对于存储系统来说，系统瓶颈一般分为 CPU 瓶颈和 I/O 瓶颈. 随着当前多核技术的不断成熟，存储系统的瓶颈主要集中在 I/O 上，但新型存储介质的出现将改变这一局面，如 NVM. Non-volatile memory(NVM)即非易失性内存存储器，它有以下关键特性：(1) 和块设备存储器如 HDD 等不同，它以字节粒度寻址；(2) 具有很高的读写性能，例如阻变式存储器(RRAM)具有比 DRAM 更低的写延迟，约 3 ns，而 DRAM 的写延迟达到约 10 ns①；相变存储器(PCM)[68]具有跟 DRAM 同样低的延迟[69]；(3) 数据非易失；(4) 具有良好的扩展性和低功耗.

由于 NVM 本身以高速字节粒度寻址并且非易失，在设计存储系统的事务日志时，它成为解决 WAL 产生的频繁小粒度 I/O 请求的最佳存储介质或缓存选择，尤其在多核高并发访问时能有效解决单一中心日志导致的扩展性问题.

5.2　日志扩展性

随着多核、大内存及新型存储介质如 NVM 等硬件技术的不断成熟，数据库系统的运行平台逐渐被转移到配置这些高性能硬件的服务器平台上. 跟文件系统类似，在 KV 存储系统中，对内存中 WAL 日志缓存的多核高并发更新，会产生 WAL 的频繁小粒度更新. 日志的扩展性和写入方式同样成为影响整个系统性能的因素.

在当前众多数据库系统中，数据恢复策略仍是通过单一中心化的 WAL 写前日志来完成的，由内存中的日志记录缓存和磁盘上的持久化日志存储构成. 传统的单一集中式日志如图 17(a)，一方面，这种单一集中式的缓存日志在当前大规模多核并行硬件环境下，多线程对其日志缓存的竞争成为主要的性能瓶颈. 多线程在集中式内存缓存日志上的 CPU 竞争开销达到 46%[10]；另一方面当前基于 DRAM 的日志缓存，需要频繁地刷新缓存并同步日志到磁盘，频繁的小粒度 I/O 在写密集负载下成为系统瓶颈. 解决该问题的典型研究是 Wang 等人提出的分布式日志，如图 17(b)所示.

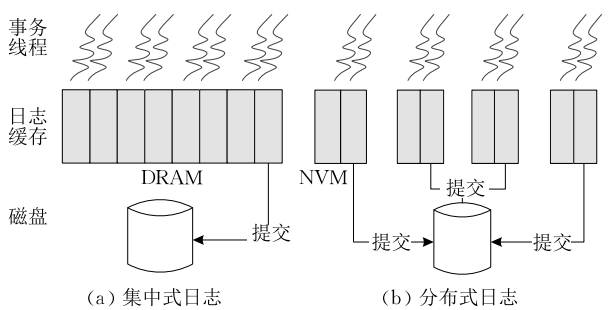


图 17　集中式日志和分布式日志

① ITRS. http://www.itrs.net 2011,6,15

分布式日志的核心思想是,在优化 I/O 方面利用 NVM 作缓存替代 DRAM. 由于它具有跟 DRAM 相似的性能并且数据非易失,这正好能够解决集中式日志存在的频繁刷新日志缓存的问题. 在解决 CPU 瓶颈方面,将集中式日志进行分区,分割为多个独立的 NVM 日志记录缓存,并通过内存接口跟 DRAM 一样使用 NVM. 这样不仅减少了多线程对集中式单一日志的竞争,而且可以将 NVM 中的日志记录进行异步写回到磁盘. 整个事务日志的单线程处理过程是:日志记录首先在 CPU 缓存中,一旦 CPU 缓存提交给 NVM,则完成了日志记录的高速持久化,写日志请求立马返回. 此时并不需要立即将 NVM 中的日志数据写回到磁盘,而是将 NVM 做中间缓存聚合数据到更大粒度,然后异步写到磁盘. 这消除了以磁盘为日志存储介质的传统事务日志管理中等待慢长 I/O 的开销. 多个线程同时操作多个日志缓存,同时也达到了多核扩展的效果.

分布式日志的核心问题是如何对日志缓存进行分区才能解决多线程事务之间的复杂依赖及恢复顺序的正确性[10]. 分布式日志的设计存在三个挑战:(1)日志划分方式. 一般的分为按页级别划分和按事务级别划分,页级别的划分允许事务线程写任何 NUMA 节点的日志缓存,这样会带来跨节点访问日志缓存的开销;事务线程级别的划分跟页级别的划分相反,事务线程只写本地 NUMA 节点被分配的日志缓存,消除了日志缓存的远端访问,提升了日志性能;(2)保证日志的恢复顺序正确性. 在集中式日志中日志序列号(*LSN*)唯一识别一条日记录,但在分布式日志中,*LSNs* 并不唯一,每一个日志缓存中均拥有自己的一个 *LSNs*,保证多个不同日志缓存中的 *LSNs* 的顺序全局一致是关键. 在事务线程级别的日志划分中,由于事务线程仅写日志记录到固定的日志页中,但多个分布式日志页被多个事务线程并发地乱序写入,此时事务中的页 *LSNs* 并不全局唯一,*LSNs* 只保证单个日志中的顺序,此时会出现不同日志中有相同 *LSN* 的情况. 为了让日志记录全局唯一,当前工作提出了基于一种逻辑时钟[70]的全局顺序号 *GSN*,在每个日志页、事务和日志中都维护一个 *GSN*,根据事务线程对日志页的加锁类型和写日志记录来产生 *GSN*,具体情况见表 5.

表 5 全局顺序号产生规则表[10]

全局顺序号(*GSN*)	日志页缓存	事务	日志
排他锁	Max(页中 *GSN*,事务中 *GSN*)+1		/
共享锁	/	Max(页中 *GSN*,事务中 *GSN*)	/
写日志	Max(页中*GSN*,事务中*GSN*,日志中*GSN*)+1		

在排他模式下,如果事务线程写日志页缓存,则此时需要设置日志缓存和事务中的全局 *GSN* 为日志页中的 *GSN* 跟事务自己的 *GSN* 的最大值加一,在共享模式下事务的全局 *GSN* 设置为事务 *GSN* 跟页缓存日志 *GSN* 的最大值;在插入日志记录到日志时需要设置日志的 *GSN* 为事务 *GSN*、页缓存 *GSN*、日志 *GSN* 中的最大值加一,日志 *GSN* 被存储到日志中. 日志记录中的 *GSN* 保证了分布式日志全局的写入顺序. 日志记录除了保存 *GSN* 外,还保存了 *LSN*,用来保存日志记录在当前日志中的偏移位置信息;(3)CPU 缓存的易失性问题. 当下处理器为了提升性能,CPU 缓存越来越大. 当应用程序通过内存接口访问 NVM 时,日志记录最先被缓存在 CPU 缓存,日志记录在没有被持久化到 NVM 之前面临着日志记录丢失的风险. 解决方案是在 CPU 缓存使用非易失的缓存,如 FeRAM,同时扩展 Pelley 等人提出的 NVRAM 组提交协议[71],构成分布式的轻量级消极组提交协议. 其原理如图 18 所示.

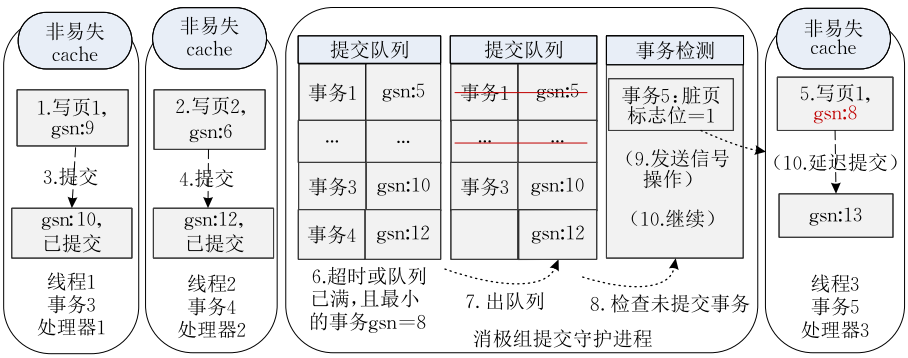


图 18 消极组提交协议原理[10]

每一个事务线程运行在各自的物理处理器上,协议有以下几步构成:(1)在日志页被提交之前事务 3 更新日志页 1,事务 4 更新日志页 2,即上图 18 中的第 1 步和第 2 步;(2)在第 3 步和第 4 步时,两个事务线程将清除脏页标记位并且检测内存页边界,将 *GSN* 为 10 和 12 的记录提交到提交队列,此时事务 5 写一条日志记录到日志页 1 并设置脏页标记位,等待更新页 1;(3)消极组提交协议守护进程

读取所有线程的 *GSN*，得到正在等待提交并且最小的 *GSN*，图 18 中为事务 5 中的 8；然后守护进程将所有 *GSN* 小于 8 的事务从提交队列出队列进行提交，然后给应用程序返回日志已被提交. 在图 18 中第 7 步后，由于事务 3 和 4 有 *GSN* 大于 8 的事务记录，因此仍在提交队列中；(4) 如果有任何掉队线程没有更新自己的 *GSN*，则提交队列会因找不到最小 *GSN* 而出现依赖混乱达到渐渐满员. 此时，守护进程将向所有有脏页标记位线程发送杀死线程信号，这些线程一旦收到信号，将重新校正自己的 *GSN* 而且清除脏页标记位. 如图 18 中第 8 步到第 10 步. 由于信号传输是异步的，杀死线程的系统调用在信号被传送之前返回. 因此，守护进程返回图 18 中第 6 步，开始重新检查延迟事务.

另外，部分相关工作直接使用新型非易失型内存替换传统的磁盘存储介质，来存储日志达到加速日志效率，典型的工作如 FlashLogging[72] 和 PCM-Logging[73]. 而另一部分相关工作在 NVM 基础上优化了多核对日志缓存的竞争，但并没有真正改变集中式日志的本质，典型的如 Aether[29]，因为多线程访问的日志数据在 CPU 缓存中共享，并存在较长的刷新等待时间，所以它允许提前释放加在内存日志缓存上的锁，来减少多线程对内存日志缓存的竞争，但这不适用于 NUMA 环境.

6　数据放置策略研究

6.1　基于全内存数据放置的系统

在 KV 系统未出现之前，面对半结构化和非结构化的新型大数据负载时关系数据库因其本身复杂的事务处理、并发控制机制，性能低下. 此时，解决这种低效的方案是采用更大的内存缓存来获取更高的访问性能，用来做关系型数据库前端缓存的基于全内存数据放置的系统应用而生，如 Memcached① 和 Redis 等.

将数据完全放置在内存与放置在外部存储设备相比，在随机访问上有着巨大的性能优势. 另外内存中的数据在数据结构组织上更加灵活，摆脱了块设备存储特性的束缚，使得数据处理非常高效. 内存放置数据一般直接使用高效的哈希索引来组织内存数据，形成庞大的内存哈希表，具有很高的查询效率. 典型系统如 Memcached、Redis.

Memcached 的基本应用模式如图 19 所示，写请求的数据需要存储到 RDBMS 中. 读请求的处理过程是，如果是首次访问时则读取 RDBMS，同时将 RDBMS 中的数据缓存到 Memcached 中. 后续访问则直接读取 Memcached，若命中则直接返回，否则按照首次访问的流程来处理.

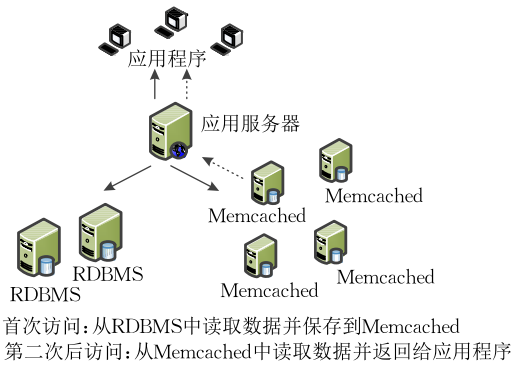


图 19　Memcached 基本应用模式

对于基于全内存的数据存储系统来说，虚拟内存机制虽然可以解决冷热数据分隔的问题，但为了保证系统可靠性依然需要借助持久化设备来实现. 但内存的数据结构仅仅通过简单处理一般难以满足块存储设备的要求. 现今较为流行的 Redis 系统是一个典型的例子，该系统主要为内存数据存储的场景设计，基本应用模式跟 Memcached 类似. 但它提供了一种自包含的虚拟内存机制②，同时也支持数据持久化存储.

Redis 是一个基于内存、可日志结构持久化的高性能数据存储系统. 跟 Memcached 相比，相同点是为了保证效率数据都缓存在内存中，不同点是为了保证数据的高可靠性 Redis 会周期性地把更新的数据写入磁盘或者把修改操作写入追加的记录文件. Redis 除了内存缓存存储以外支持两种持久化放置策略，快照和增量追加日志. 快照，即定期将完整的内存数据持久化到磁盘文件中. 当 Redis 需要做快照持久化时，它会创建一个子进程，子进程将内存数据写到磁盘上一个临时只读文件中，然后将旧的临时只读文件删除，达到写时复制的效果. 快照临时文件的优势在于能够很好地进行数据的灾难性恢复，缺点在于做快照时间的设定直接跟数据的丢失率相关. 另外，在数据集较大或 CPU 性能不够好的环境下，做快照不断创建子进程的过程耗时较长，严重时长达一微秒甚至是一秒③；增量追加日志是记录对数据的修改操作. Redis 将数据缓存在内存中，然后定期通过异步方式保存到磁盘上，这称为“半持久化模式”；也可以把每一次数据变化都写入到一个增量追加日志文件里面，称为“全持久化模式”，Redis 的全持久化模式实际是靠同步时间间隔和日志来完成的. 增量日志的优势在于，提供了多种持久化策略，可以根据负载的需求设定相应的持久化模式，将数据的丢失率降到最低. 而缺点是，在相同数

① Memcached. http://memcached.org 2016,4,8
② Antirez. Redis Virtual Memory: the story and the code. http://antirez.com/post/redis-virtual-memory-story.html 2016,3,28
③ Redis. http://redis.io/topics/persistence 2016,7,10

据集下由于额外的元数据等信息,增量日志文件的大小比快照临时文件要更大.选择何种持久化策略取决于应用对数据安全的依赖程度,根据不同的安全级别选择其中一种或两种策略并用.

全内存的数据放置虽然提供了很高的读性能,但对于写性能来说,存在着很多挑战,包括传统的并发控制和内存缓存的管理方式等[74].

6.2 基于持久化存储的系统

持久化 KV 存储系统在当前数据结构种类纷繁复杂的大数据应用中扮演着非常重要的角色[75],包括网页应用[3]、电子商务[2]、图片存储[76]、云存储[77]、社交网络[78-79]等.而持久化存储设备主要为传统的机械磁盘 HDD 和新型 Flash 存储设备 SSD.在数据的放置方式即数据的读写方式方面二者有相似的性能特点,对于 HDD 和 SSD 来说小粒度的随机写都不利于其性能发挥[80-83].随机写对 HDD 来说,由于 HDD 固有的机械寻到机制,性能大打折扣.对于 SSD 来说,根据已有的研究工作 Soft-defined Flash(简称 SDF)[84],SSD 由 Flash 芯片阵列构成,小粒度的读写请求不利于发挥其内部的并行能力,而且由于 SSD 页粒度写块粒度垃圾回收并且写前回收的特性,频繁的小粒度随机写会导致 SSD 内部数据碎片化加剧,加重了 FTL 层垃圾回收的负担,这将严重影响 SSD 的性能发挥.因此,即使在使用 SSD 的今天,I/O 在持久化存储系统中仍成为主要瓶颈[85-86].但二者对数据写入方式有共同的需求,那就是增大写粒度,减少随机写.那么当前对于持久化设备的数据放置技术研究主要为 Log-structured 技术.Log-structured 即日志结构存储技术,这一技术源自经典的研究工作 Log-Structured File System[48](简称 LFS),这一工作最早由 Ousterhout 和 Douglis 在 1981 年提出,并做了具体的实现.其后有许多不同类型但具有相似核心设计思路的文件系统也被泛称为 LFS.近年来,这种类型的文件系统由于其读写负载比较适合 Flash 类型存储设备,故其中如 JFFS[87] 和 YAFFS① 等系统被不少智能手机的 Flash 存储设备所采用.

传统文件系统的数据组织比较注重数据的空间局部性比如经典的 ext2/3 文件系统[88],数据和元数据都是分别放置在特定区域.对于 LFS 来说,其设计思路核心在于将硬盘空间以类似日志的形式做写入操作,也就是将原本写入操作中频繁的元数据或者数据原地更新操作转化为日志式的顺序写操作.LFS 的主要性能优点有:(1)利用顺序操作替代随机操作,减少大量的磁盘寻道操作,提高了带宽利用率,进而提高了文件系统写入操作的性能;(2)提供了一种文件系统不间断快照的能力,持久自动备份文件系统的操作.这一点类似于版本文件系统的设计功能;(3)提供了一种高效的错误恢复功能.相比于传统文件系统需要大量时间恢复的元数据不一致等严重错误,LFS 可以快速从崩溃前的检查点恢复,而且这种快速高效的恢复能力与管理数据的总量是没有关系的.

在 KV 型存储系统中 LSM-tree 有着跟 LFS 同样的设计初衷,均是典型的采用日志结构存储技术的设计.当前基于日志机构存储技术的研究工作有基于 LSM-tree 的实际研究系统包括从本地系统如 LevelDB、RocksDB、cLSM-tree[7]、LogBase[89]、LSM-trie[90] 等到分布式系统 BigTable[3]、Cassandra[5]、PNUTS[6]等.另外采用哈希索引技术的 FAWN-DS[11]、采用 B+tree 索引的 BerkeleyDB、NVMKV[91] 等研究工作在针对 SSD 的数据放置设计时也采用日志结构存储技术.在此介绍典型的日志型存储工作 FAWN-DS.

FAWN-DS 采用基于哈希的索引引擎和基于日志结构存储的 SSD 数据放置技术.其架构如图 20 所示.

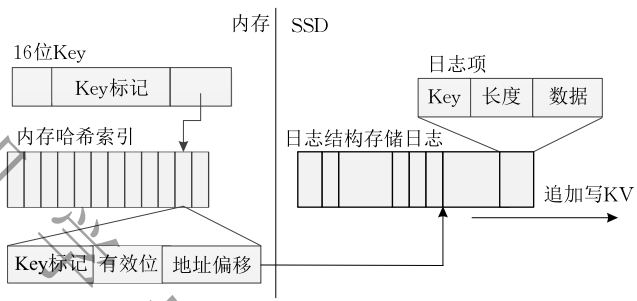


图 20 FAWN-DS 架构[11]

FAWN-DS 为了节省内存不直接将 key 的全部内容存储到哈希表,而是存储一个 key 的标记位,类似于内存中前缀匹配,如图 20 所示.key 的长度是固定的 160 位即 20 B,位于右端的高 i 位作为索引位,紧接着是长度为 15 位的 key 标记位,其余空间为 key 的内容.内存哈希表中的每个项的长度为 6 个字节,由三部分构成即 15 位的 key 标记位、一位有效位和 4 字节长度的指针.4 个字节的指针用来记录 key 对应的数据 value 在 SSD 上的偏移地址.

当根据 key 读取一条 value 记录时,先根据 key 的高 i 位确定哈希桶的位置并读取相应的哈希表中的项,然后通过比较两个 key 的标记位来确认 key 是否匹配,若匹配则根据哈希项中保存的指针位读取 SSD 上的相应数据.若匹配错误或不完全,则需要扫描 SSD 上哈希链进一步比较确认.

当存储 KV 数据项时需要做两个操作,一是追加写入数据到日志结构文件中,二是插入写入数据

① YAFFS. https://en.wikipedia.org/wiki/YAFFS 2016,4,8

的偏移值到相应的内存哈希表条目中，并将有效位置位. 跟其他采用日志结构存储数据放置策略的系统类似，FAWN-DS 的设计中也不存在更新数据时做旧数据的删除工作，甚至在删除某个数据条目时也只是写入一个删除标记也就是将删除操作转化为一个插入操作并没有回收空间. 这种设计都是为了将传统数据放置机制中的随机写入更新操作转化为顺序的日志结构追加写操作. FAWN-DS 是一种符合 SSD 读写特性的存储方式，将大量的随机更新转化为日志结构的顺序更新，减少了 SSD 的 FTL 层的开销. 当前的 KV 系统的研究中，对数据放置技术的研究兼顾了 HDD 跟 SSD 的特性，数据放置技术对二者是通用的.

非易失性内存 NVM 是一种以字节粒度寻址，有着跟 DRAM 同等大小的读写延时而且数据非易失的存储设备，当前主要用于事务日志的优化存储. 基于集中式日志的数据放置策略不利于发挥 NVM 和多核的并行处理能力，分布式日志缓存在多核架构下具有很高的扩展性. 针对新型存储介质 NVM 的数据放置策略优化的详细内容请参见 5.2 节.

7　LSM 型 KV 系统研究

以上从系统研究的视角出发，重点从数据放置技术研究和并发控制技术研究两个维度简述了 KV 型本地存储系统的研究现状. 其中 LSM 型系统因其特殊的索引技术、数据放置技术和读写并发控制机制，在集多核、大内存及 SSD 存储设备一身的高性能服务器环境下成为学术界的研究热点，如雅虎的 Golan-Gueta 等人提出的 cLSM-tree[7]、北京大学的 Wang 和 Sun 等人提出的 LOCS[20]、威斯康森大学的 Lu 等人提出的 WiscKey[75] 等. 同时，也成为了当下工业界所使用的主流 KV 存储系统，本地系统如谷歌的 LevelDB，Facebook 的 RocksDB，HyperDex[49] 的 HyperLevelDB，分布式系统如谷歌的 BigTable[3]，Apache 的 Hbase①、Cassandra[5]，雅虎的 PNUTS[6] 等. 因为 LSM 型本地存储系统是实现分布式系统的核心存储引擎，其系统性能优化研究成为当下热点. 下面在如前叙述的研究工作基础上综合的叙述 LSM 型 key-value 本地存储系统的研究现状. LSM 型本地存储系统的研究主要在两个方面，一是对顺序字符串表的合并过程进行优化，减少读写放大导致的系统在 I/O 上的瓶颈；二是在当前多核的硬件背景下，对 LSM-tree 索引的并发控制的优化研究，实现高扩展性.

7.1　合并优化，减少读写放大

如前所述的 LevelDB，LSM 型系统的最大缺点是顺序字符串表之间的频繁合并导致的读写放大，尤其当请求数据的粒度增大时对系统的性能影响显著增加[75]. 一方面频繁地合并会产生额外读取磁盘的 I/O，另一方面合并过程会阻塞写线程对内存表的写或减缓写速率，在合并的负荷较大时会停止写线程使其等待合并完成，这样的设计直接影响系统的写性能. 那么当前涌现出很多对 LSM 型系统的合并过程进行优化的研究工作，如雅虎的 bLSM-tree[19]、中国科学院大学的流水线合并[92]、Facebook 的 RocksDB、北京大学的 LOCS[20] 及威斯康森大学的 WiscKey[75] 等.

图 21 是 WiscKey 工作对 LevelDB 读写放大的一个基本测试：key 为 16 B，value 为 1 K，请求 key 为随机均匀分布[93]；YCSB[94] 分别加载 1 G、100 G 数据集进行准备测试；YCSB 在加载之后分别运行 100 000 条读操作测试读放大. 结果如下，随着数据集的不断增加，读写放大越加严重，在 100 G 数据集的情况下合并导致的读放大达到 327.

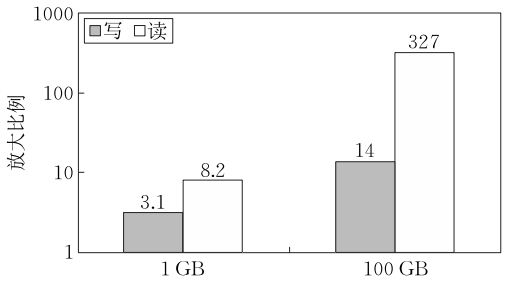


图 21　读写放大测试[75]

7.1.1　bLSM-tree

bLSM-tree[19] 优化 LevelDB 中的合并调度器，进一步减少读放大带来的性能影响. 一方面用布隆过滤器[18] 减少读取磁盘无用数据带来的开销，减小读放大. 另一方面 bLSM-tree 减少了层级数，只有三层，位于内存中的 C_0 和磁盘中的 C_1，C_2. bLSM-tree 跟 LevelDB 不同，采用 B-tree 的内存表和磁盘数据结构，这样的设计适合小对象数据的读写.

7.1.2　RocksDB

RocksDB 的目标是优化合过程并减少写阻塞. 在 LevelDB 中，在只读内存表被单线程合并到磁盘的过程中，如果新的内存表又被写满了，此时新满的内存表只能等待合并过程完成才能变为只读内存表，然后建立新的内存表来接收写操作请求. 在等待的过程中写操作被阻塞，换言之，被写的内存数据对合并过程来说供过于求，性能瓶颈出现在单线程的合并.

RocksDB 的优化工作主要从以下两个方面进行的. 首先通过增加多个只读内存表，同时利用多线程处理合并过程，以解决写内存表的速度与合并字串顺序表的速度之间差距造成的性能瓶颈；其次是优化合并过程. RocksDB 增加了合并时的布隆过滤

① Hbase：Bigtable-like structured storage for hadoop HDFS. http://wiki.apache.org/hadoop/Hbase 2016,3,22

器[16],在读取被压缩的顺序字符串表时,对一些不再符合条件的 KV 进行丢弃,这样减少无效的磁盘数据读取.

RocksDB 实现了两个布隆过滤器,一个是在读数据块之前使用布隆过滤器过滤不包含 key 的数据块,节省 I/O 带宽.另一个是在查询内存表时动态生成一个布隆过滤器,对内存中的 key 进行过滤,提高查找效率,但付出的是空间代价.RocksDB 支持在 key 的一部分上设置布隆过滤器,很好的支持了范围查询.

RocksDB 优化 LevelDB 的单线程合并为多线程,解决了合并过程阻塞写内存表的问题.此时,系统的性能瓶颈主要在于单通道的磁盘 I/O,无论多线程对内存表的合并多快,但磁盘的 I/O 带宽是有限的.下面的工作,基于新型存储介质 SSD 对这个问题进行了优化.

7.1.3 LOCS

LOCS[20],主要工作一方面是优化合并减少写阻塞.LevelDB 不支持多线程合并,这样会出现合并过程阻塞写内存表的情景;另一方面是结合合并过程对 I/O 带宽进行的优化.传统的 SSD 限制了读的并发能力,读性能低.对于 Open-Channel SSD 来说,当前的 LevelDB 并不感知 SSD 的多通道 I/O,所以它的合并操作是由单线程依次读取磁盘中多个不同层级的顺序字符串表文件进行的,不能并发读.

LOCS 是基于一种优化过的多通道 Open-Channel SSD 来改进 LevelDB,优化读写性能.对于写,在内存中具有一个内存表和多个只读内存表.多个只读内存表由 LevelDB 的 I/O 请求调度器分配给多个合并线程,多个合并线程由 SSD 调度器发往不同的 SSD 通道.这样做到了单线程写内存,多线程合并磁盘的效果.对于读,跟写的过程相似,由多个读线程并发的从不同的 SSD 通道读取多个不同层级的顺序字符串表文件到内存中进行合并.LOCS 的系统架构如图 22 所示.

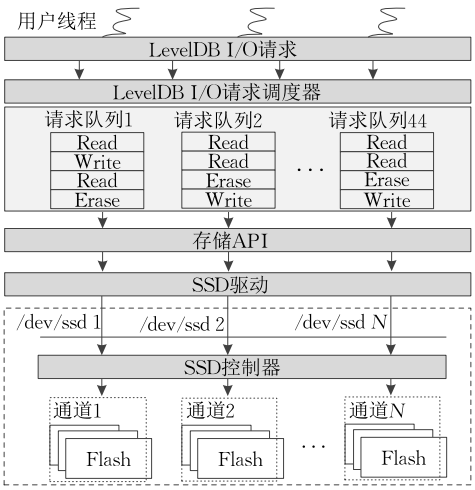


图 22 LOCS 架构[20]

核心组件主要由 LevelDB I/O 请求调度器和 Open-Channel SSD 构成.改进后的 LevelDB 在内存中设计了多个只读内存表,数量可以配置.如何将众多的只读内存表发往不同的线程队列是一个关键问题,需要保证各队列的负载均衡.LOCS 设计了两种调度策略:(1) Round-Robin 即循环接收请求.对于读写,写的延迟大于读,如果其中个别队列中聚集了大量的写请求,那么导致各队列的不平衡,个别线程会闲置影响整体性能;(2) Least Weighted-Queue-Length 即最少权重队列长度.根据读写请求延迟,给读写请求赋予不同的权重值,请求的调度根据队列的权重长度来进行,这样各线程队列之间达到一个相对平衡.

Open-Channel SSD[84] 如图 23 所示,SSD 将其内部的多通道特性暴露给应用,每个通道可以被应用多线程并发地访问.对于单个 SSD,磁盘 I/O 带宽将大大增加.这是 SSD 的软件实现.

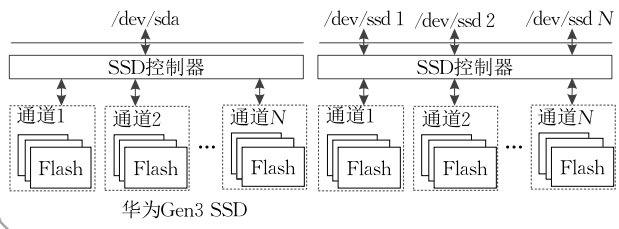


图 23 Open-Channel SSD[20]

LOCS 从多个只读内存表、请求调度器和多通道 SSD,到多个合并线程的设计和实现,优化了整个系统整体的请求处理流程和最大化、最小化合并过程.充分利用了 CPU 和 I/O 设备资源的硬件级并行处理能力,实现了系统整体的高吞吐率.

7.1.4 流水线合并

流水线合并[92],是由中国科学院大学的 Zhang 等人提出的一项优化 LSM-tree 中最大化合并过程的技术.它的动机跟 LOCS 一样,充分使用底层硬件资源的并行处理能力使系统具有高扩展性,获得较高的合并带宽和吞吐率.但跟 LOCS 不同的是,LOCS 对最大化最小化合并做的全局优化,一方面通过设计多个只读内存表,避免了在单个只读内存表的情况下合并过程阻塞写请求的情况;另一方面通过优化请求调度和利用多通道 SSD 以及使用多个合并线程,加速合并过程.但是,它并没有改变合并过程本身,多个线程仍是顺序地执行合并过程中的每一步骤.而流水线合并优化了合并过程本身,改变了合并过程的具体细节,多个合并线程流水线式执行合并过程.但流水线合并的工作聚焦在 LSM 中的最大化合并过程,不涉及最小化合并,也没有改变已有只读内存表的个数及请求调度,做的是局部的优化.

如图 24 所示,LSM-tree 中的合并过程由 S_1 ~

S_7 组成. S_1：从磁盘读取一个数据块和相关校验和到内存；S_2：计算校验和判断数据完整性；S_3：解压数据块，得到原始的 KV 对；S_4：合并两个数据块的数据并进行排序，形成新的数据块；S_5：压缩新形成的数据块；S_6：再次计算校验和，为将来的 S_2 做准备；S_7：将经过 S_5 得到的新数据块和 S_6 形成的校验和从内存写入到磁盘. 其中 S_1 和 S_7 有磁盘读写 I/O 发生，而 $S_2 \sim S_6$ 仅有 CPU 计算.

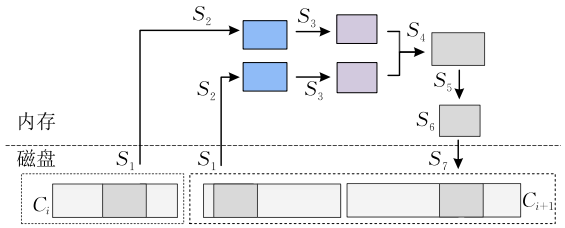


图 24　LSM-tree 的最大化合并过程

在 LSM-tree 中，以上的合并过程是顺序执行的，如图 25 所示.

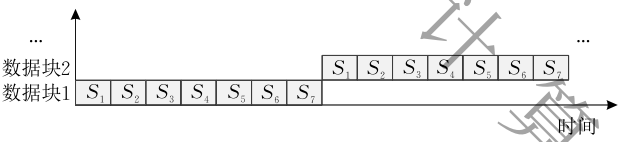
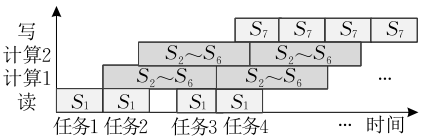


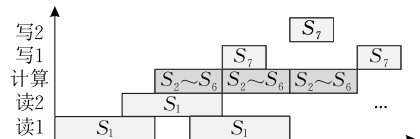
图 25　顺序合并过程

LSM-tree 的性能开销在于复杂的后台合并过程，由于合并过程是顺序执行的，在多核和新型存储介质的硬件环境下，合并性能无法得到扩展. 对于顺序合并过程，根据流水线合并工作的分析评测，当 HDD 作为持久化介质时，两个阶段 S_1 和 S_7 构成的 I/O 用时占用合并过程总时间的 62%；当 SSD 作为持久化存储介质时，$S_2 \sim S_6$ 阶段组成的 CPU 计算用时占用合并过程总时间的 60%以上[92]. 根据评测结果和理论分析，该工作将流水线合并的过程分为并行计算流水线合并和并行存储流水线合并.

并行计算是指，在使用 SSD 持久化存储介质时，如图 26(a)中所示，由 $S_2 \sim S_6$ 阶段构成的 CPU 计算比由 S_1 和 S_7 构成的磁盘读写存储消耗更多的时间，CPU 计算成为合并瓶颈. 因此对 $S_2 \sim S_6$ 的计算过程设计多个子线程进行并行处理. 并行存储如图 26(b)中所示，实现了并行计算后，由于 S_1 和 S_7 所占用的时间大于 $S_2 \sim S_6$ 的计算时间(如使用 HDD 作为持久化存储介质的情况下)，I/O 成为合并瓶颈. 因此对多个流水线中的 S_1 和 S_7 采用多块磁盘进行并行读写，实现并行存储. 流水线合并的主要挑战在于实现并行存储流水线. 由于不同顺序字符串表中数据 key 存在重叠，因此不同数据块之间会存在依赖，无法做到数据块间的并行. 但是在单个顺序字符串表中，由于数据项 key 是有序的，数据之间不存在依赖关系. 因此，并行存储流水线合并中流水线并行读写是基于单个顺序字符串表中的数据块进行的，这是实现并行存储流水线合并的前提.



(a) 并行计算流水线合并过程



(b) 并行存储流水线合并过程

图 26　并行流水线合并过程[92]

假设 b 是数据块的大小，t_{S_i} 是过程 S_i 的执行时间，那么顺序合并的带宽为

$$B_{\text{顺序合并}} = \frac{b}{\sum_{i=1}^{7} t_{S_i}} \tag{1}$$

即合并的总执行时间为步骤 $S_1 \sim S_7$ 的时间总和，这种情况下合并时间是最长的.

假设 m 代表磁盘个数，那么得出的存储并行流水线合并的带宽为

$$B_{\text{存储并行流水线}} = \frac{b}{\max\left\{\frac{t_{S_1}}{m}, \sum_{i=2}^{6} t_{S_i}, \frac{t_{S_7}}{m}\right\}} \tag{2}$$

根据式子(2)，如果 $m < \dfrac{\max\{t_{S_1}, t_{S_7}\}}{\sum_{i=2}^{6} t_{S_i}}$ 则说明合并过程中执行读写 I/O 的时间超过 CPU 计算的时间，I/O 成为合并瓶颈；否则合并过程受限于 CPU 计算. 此时，理想的合并加速比为 $\min\left\{m, \dfrac{\max\{t_{S_1}, t_{S_7}\}}{\sum_{i=2}^{6} t_{S_i}}\right\}$.

对于并行计算流水线而言，假设有 n 个线程参与合并计算，则其带宽为

$$B_{\text{计算并行流水线}} = \frac{b}{\max\left\{t_{S_1}, \frac{\sum_{i=2}^{6} t_{S_i}}{n}, t_{S_7}\right\}} \tag{3}$$

根据式子(3)，如果 $n < \dfrac{\sum_{i=2}^{6} t_{S_i}}{\max\{t_{S_1}, t_{S_7}\}}$ 则说明合并过程受限于 CPU 计算，需要增加合并线程数量. 然而如果设置过多的合并线程，合并带宽并不会提升，合并瓶颈转向磁盘 I/O，这跟实际的评测结果一致[92]. 在这种情况下理想的合并加速比为 $\min\left\{n, \dfrac{\sum_{i=2}^{6} t_{S_i}}{\max\{t_{S_1}, t_{S_7}\}}\right\}$.

根据流水线合并[92]的评测结果,优化后的并行流水线合并与传统的顺序合并相比,合并带宽提升77%,系统 IOPS 提升 62%.

7.1.5 WiscKey

正如前面所述,LSM-tree 型 KV 系统固有的缺点是它的合并导致的读写放大. RocksDB 采用多个合并线程和布隆过滤器[16]来优化这种读写放大带来的 I/O 开销,bLSM-tree[19]采用布隆过滤器的同时优化了合并算法. 那么 WiscKey[75]的动机也是优化合并,减小合并产生的性能瓶颈.

WiscKey 的结构如图 27,它的核心思想是将 KV 中的实际数据 value 从 LSM 中分离到一个单独的日志文件中.

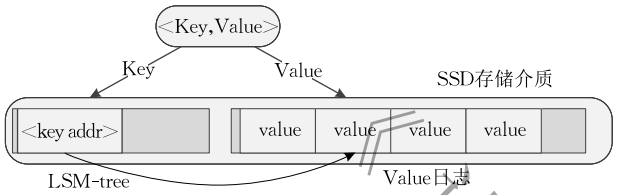


图 27 SSD 上的 WiscKey 数据层次[75]

在 LSM 系统中存储 key-addr,addr 只是一个存储日志文件中 key 对应的实际数据 value 的偏移地址的指针. 这样减少了 LSM 的数据量,无论实际数据 value 的粒度多大,读写放大维持在较低水平.

在 LevelDB 中 KV 是连续并且顺序存储的,这样在范围查询的时候从顺序字符串表中顺序地读 KV. 然而在 WiscKey 中,key 跟 value 是单独存储的,范围查询时需要随机地读 value,这对于范围查询来说很低效. WiscKey 利用 SSD 设备的并行 I/O 通道,从 value 日志文件预读范围查询的 values. WiscKey 设计了跟踪范围查询的模型,当跟踪检测到一段连续的 KV 对被请求时,WiscKey 开始从 LSM-tree 读出预取数量的连续 keys. 同时相应的 value 地址也从 LSM-tree 读出插入到一个队列,多个后台预取线程将范围查询对应的 values,并发地从 SSD 的 value 日志文件中读出.

对于垃圾回收来说,当一个 KV 对被删除或更新时 WiscKey 不会立即回收该数据的空间,而是在合并期间若有删除或覆盖更新的 KV 对时,则将该数据丢弃而且释放. value 日志中 value 的详细结构如图 28 所示.

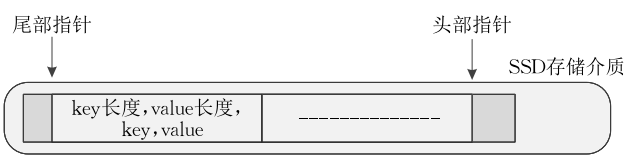


图 28 value 结构[75]

头部指针(value 日志的末尾)和尾部指针在内存中维护,在 LevelDB 中持久化保存. 当垃圾回收线程改变了尾部指针后,所有对 value 日志的写追加到日志文件的头部. 头部和尾部之间的有效数据将在合并过程中读数据时被搜索到. 从尾部开始读取大块 KV 对,当查找到的 KV 对是有效的,就将其追加到 value 日志的头部位置,然后释放该块 KV 对所占用的空间且更新对应的尾部指针. 为了避免在垃圾回收期间系统崩溃,WiscKey 在释放块内存之前先做一次持久化,将新追加的 value 和新尾部同步到 SSD. 跟 LevelDB 一样,后台线程做合并时触发垃圾回收.

如前所述,bLSM-tree 通过优化合并算法和增加布隆过滤器,减小合并频率及合并过程中的读写放大;RocksDB 采用多只读内存表的设计,同时使用多线程合并以加速合并过程的并行执行,减少合并过程对写内存表的阻塞;LOCS 从多只读内存表、请求调度和基于 Open-channel SSD 的多线程合并方面,充分利用 I/O 设备的并行能力提升合并速度;流水线合并优化改变合并过程本身,将原始的顺序合并改变为并行流水线合并,通过多线程执行合并任务和多磁盘提供存储带宽,提升了合并带宽;WiscKey 将 KV 中的 value 数据分离到单独的日志文件中存储,减少合并的数据量,以减少合并过程的性能开销. 其他一些工作,典型的如 PE[95]将数据项 key 的范围划分成多个子 key 范围,并只对热数据项 key 进行合并,减少合并数据量;GTSSL[96]采用了基于 SSD 作为高层顺序字符串表的读缓存,HDD 作为低层次顺序字符串表的存储介质的混合存储. 它使用重新插入和空间回收机制,将冷热数据进行分层存储,即将低层 HDD 中被最新访问的热数据项 KV 提升到更高的层级 SSD 中,并将合并后的数据驻留在高层,这样减少了总的层级数目. 同时结合布隆过滤器,减少了读放大. 另外,基于 SSD 为高层顺序字符串表的存储介质的设计,不仅提高了查询性能,而且加速了最小化合并过程.

以上工作解决了单线程的合并过程阻塞写内存表的问题,以及使用多线程合并顺序字符串表时磁盘 I/O 成为系统瓶颈的问题. 然而,对于 LevelDB 来说,内存表只支持单线程的写不支持并发写. 也就是说,即使合并过程和磁盘 I/O 很畅通无阻,如果内存中的数据供不应求,那么系统整体性能仍较低. 下面介绍多线程对内存表的并发更新,实现系统高扩展性的相关工作.

7.2 优化并发控制,实现高扩展性

传统的 LSM 型系统采用多版本的读并发控制,具有很高的读并发性能. 但是在写并发控制方面,仍采用对内存表的全局锁机制,多线程的任务只能依次进入写线程队列等待前面任务完成,始终只允许单个线程对内存表进行更新操作. 在当前多核

环境下，这种机制制约着系统的写并发能力，限制着系统扩展性. 在发生频繁合并导致读写放大时，多个写任务在写队列中排队等候的延迟进一步增加. 采用内存表的全局锁机制的研究工作有 LevelDB 和以 LevelDB 为基础的研究系统如 RocksDB、bLSM-tree 以及 WiscKey 等.

在写并发控制方面，对全局锁的优化研究工作主要包括两个方面. 一方面是消除全局锁，在内存表的跳表上使用更加细粒度的锁，使得多个写线程并发去执行写任务. 只有当多个写线程对相同节点进行更新时加锁的同步动作才会发生. 这样大大增加了多线程并发能力，减少了全局锁的频繁加锁解锁带来的线程上下文切换开销和线程阻塞. 典型的工作如 HyperLevelDB.

7.2.1　HyperLevelDB

LevelDB 中内存表的写不能并发，阻碍写性能. 如果合并过程效率很高如在 RocksDB、LOCS[20] 中，那么在内存表未写满之前不会变为只读内存表，也就是单线程的写内存表对合并到磁盘的操作来说数据供不应求. 对内存表的多线程并发写，成为系统的需求. HyperLevelDB 的优化方案是改进并行机制，允许多个写线程独立地插入自身要写的内容到日志和内存表里，同时为了维护写的顺序而使用同步机制.

LevelDB 固有的并发控制是，对于写操作是通过对内存表加全局锁来实现的，多个线程依次顺序的写内存表，任意时候只有一个线程在执行写. HyperLevelDB 优化了 LevelDB 内存表中的数据结构，在跳表上使用更细粒度的内部锁，保证多线程并发的写，提升写性能. 跳表允许多线程无锁的读，并且对写线程使用外部的同步机制. 读和写操作执行第一步均是遍历跳表以在表中查找期望的关键字，在没有拥有任何锁的情况下执行这个遍历. 其次对写操作来说，允许所有线程在不需要牺牲正确性的情况下，并行地插入数据到跳表中.

同时 HyperLevelDB 使用 mmap 和 munmap 系统调用维护用户缓冲空间，允许并发线程自动向日志文件添加内容. 日志文件同步底层缓冲的访问，并且给不同的写线程指定非重叠区，允许写线程并行地拷贝自己的数据. HyperLevelDB 也优化了数据合并机制，实现了一个降低写放大的合并调度器. 合并算法在合并时选择在两个层级中产生最小写放大的一组顺序字符串表. 一个后台合并线程监控每个层级的大小，并且对最适合合并的一组顺序字符串表进行压缩. 第二个后台线程在第一个线程无法承受更多负载的时候触发，它执行全局优化以选择可降低写放大的压缩目标，而且并不关心每个层级的大小.

对写并发控制第二个方面的优化研究是，基于无锁或锁无关的 lock-free 的写并发控制机制，使得多个线程并发地原子更新共享节点而无需加锁成为可能. HyperLevelDB 在内存表的跳表上采用更加细粒度的锁，跟传统的全局锁相比，减少了线程阻塞增加了多线程的并发. 但是在面对海量数据请求更新时，这种细粒度的加锁导致的线程上下文切换和阻塞开销仍非常高.

当前基于 Lock-free 写并发控制的研究在 LSM 型系统中典型的工作如雅虎的 cLSM-tree[7].

7.2.2　cLSM-tree

cLSM-tree[7] 是对内存中内存表的多线程并发写的另一种优化. 核心工作是对内存表中的数据结构采用无锁的并发数据结构跳表[50]，提高多个写线程的并发，实现高扩展性.

cLSM-tree 采用 libcds① 提供的无锁并发数据结构映射和跳表实现了内存表，这样内存表支持多线程无锁的并发读写访问. 在 LevelDB 中当内存表写满后，内存表被转变为只读内存表，然后做合并操作，做完合并操作后只读内存表从内存中丢弃，同时创建新的内存表来接收写请求. 但是在多线程并发写的情况下存在的问题是：并发操作下如何同步全局的三个指针即指向内存表的指针 P_m、指向只读内存表的指针 P'_m 及指向磁盘顺序字符串表的指针 P_d. cLSM-tree 优化了对这些指针的加锁方式，分为两个过程和两种性质的锁[7]. 如下过程 1，在写内存表的过程中对 P_m、P'_m、P_d 三个指针加共享锁，允许所有线程只读指针不允许修改，处于共享状态. 合并分为两个过程如下过程 3 和过程 4，对该三个指针加排他锁，只允许单个线程去更新内存表的指向，处于独占状态.

过程 1.　写过程 PUT(key k, value v).

1. Lock.*lockSharedMode*()
2. P_m.insert(k,v)
3. Lock.*unlock*()

过程 2.　读过程 GET(key k).

1. value $v \leftarrow$ find k in P_m, P'_m, P_d in this order
2. RETURN v

过程 3.　BEFOREMERGE 过程.

1. Lock.*lockExclusiveMode*()
2. $P'_m \leftarrow P_m$
3. $P_m \leftarrow$ new in-memory component
4. Lock.*unlock*()

过程 4.　AFTERMERGE(DiskComp Nd)过程.

1. Lock.*lockExclusiveMode*()
2. $P_d \leftarrow N_d$
3. $P'_m \leftarrow \perp$
4. Lock.*unlock*()

① libcds. Library of lock-free and fine-grained algorithms. http://libcds.sourceforge.net/2016,4,15

在过程 3 中,内存表被转变为只读内存表后,需要更新 P'_m 的值指向新生成的只读内存表;需要创建新的内存表,然后更新 P_m 指向新的内存表.在过程 4,只读内存表被合并到磁盘后,需要同时更新指向磁盘最新的顺序字符串表的 P_d 指针和指向只读内存表的指针 P'_m(此时为空).

Libcds,是一个包含无锁容器和安全内存回收算法的开源 C++库.该库包含一系列并发数据结构的实现(如队列、链表及集合等),也包含基于这些数据结构的 lock-free 和 fine-grained 算法.

8 研究展望

大数据的 4V 特性给 KV 型本地存储系统提出了更高的性能需求,而多核和存储等硬件技术的发展为它的性能扩展带来了机会,但同时也向它提出了严峻的挑战,主要表现为多核扩展性和针对新型存储介质数据放置.围绕这两个挑战,出现了众多关于 KV 型本地存储系统的研究工作并有了一些初步成果.本文针对大数据 4V 特性中的 Volume 和 Velocity 特性,从数据索引、并发控制、事务日志管理和数据放置管理四个方面对现有的研究成果进行了分析综述和归纳.从现有的研究成果来看,数据索引作为整个系统的引擎,关于它的研究一方面优化数据组织结构,减少索引本身的缺陷带来的性能开销,如布谷哈希等[15-17],另一方面是针对索引的多核扩展性研究;针对多核扩展性的研究也已经有了初步的成果,在并发控制方面,基于多版本的读并发控制 MVCC 和基于无锁 lock-free 的写并发控制表现出了出色的性能,具有多核的高扩展性[7-8],因而备受关注;在事务日志扩展性方面,基于新型存储介质 NVM 设计分布式事务日志的工作开始出现,这能够突破传统的基于磁盘的集中式日志导致的系统扩展瓶颈,但由于处于起步阶段,实际的系统仍不多见;在针对新型存储介质的数据放置策略方面,日志结构存储被众多研究系统所采用,充分利用了 SSD 的硬件性能,成为最佳的数据放置策略选择,尤其 LSM 型存储系统采用日志结构存储是它的典型设计.总体来看,尽管 KV 型本地存储系统的研究已经取得了重要进展,但围绕大数据 Velocity 的负载需求和多核的硬件架构环境,仍有诸多挑战待探索和解决.根据已有的相关研究成果,我们可以预见以下有关 KV 型本地存储系统的研究趋势:CPU 缓存高效性、事务日志扩展性和高可用性.

(1)CPU 缓存高效性.随着服务器配置的不断提升,CPU 缓存的容量越来越大,而对于对小粒度数据请求具有天生性能优势的 KV 存储系统来说,充分利用大容量的 CPU 缓存挖掘系统的读性能成为关键.虽然高扩展性的存储系统充分利用了多核处理器的并行能力,但这给 CPU 缓存的高效访问带来了挑战.基于 CPU 原子指令支持的无锁 lock-free 写并发控制支持多线程高并发更新[7],减少了传统锁并发控制导致的线程阻塞和上下文切换开销,但是多线程的高并发带来了多个 CPU 缓存的一致性问题.缓存一致性是表示处理器缓存中的数据项目值与系统内存中的数据项目值相同的一种状态.在对称多处理器系统中,每个 CPU 都有各自的本地缓存,内存系统必须保证缓存一致性.多线程交替无锁高并发地更新同一内存地址,导致多个 CPU 的缓存副本失效,发生了 CPU 缓存行之间的伪共享①,需要强制读取内存来维护缓存一致性.缓存行的频繁伪共享增加了 CPU 缓存行的替换次数即发生的频繁的缓存写回,导致大量的缓存失效,并且在 NUMA 架构下带来了更长的跨节点的通信延迟.

在多处理器多核环境下,存储系统的设计需要保护缓存行的地址,尤其对于小粒度的 KV 存储系统来说,缓存的命中率跟 KV 系统的性能直接相关[7].当前有两种解决方案,第一种是对多核共享内存数据进行分区,将 CPU 跟指定的分区进行绑定,避免CPU缓存之间的干扰,典型的工作如 MICA[67].这种方案的不足是,当请求的 key 对分区的倾斜度很严重时将影响系统的扩展性,造成多 CPU 资源的使用不平衡.所以负载请求对分区的倾斜控制算法成为关键.第二种是采用异地增量更新策略[8].无锁 lock-free 的写并发控制,实现了多线程无锁并发的访问共享的数据,如果采用页粒度的原地更新,那么多线程对共享数据结构的大规模随机更新会造成 CPU 缓存行的频繁替换,导致更加严重的缓存失效.然而,在异地增量更新的情况下,一个新的 CPU 缓存行就能够容纳多个小粒度更新请求,而不必频繁破坏已有的缓存行或踢出一少部分缓存行.在多核架构下,关于 CPU 缓存的高利用率的研究工作仍不多见,解决了多核扩展性挑战后,如何最大限度地降低 CPU 缓存一致性带来的缓存失效,高效利用当前的大容量缓存开采 KV 型本地存储系统的性能成为未来的研究趋势.

(2)事务日志扩展性.KV 型 NoSQL 系统在面对 Volume 和 Velocity 特性的大数据负载时,系统的可靠性保证成为重点.在多核和新型存储架构下,无锁的并发控制实现了索引数据结构的多核高扩展性,KV 系统的性能瓶颈逐渐转向基于块设备的事务日志 WAL[10,27].当前 KV 型存储系统采用 WAL 技术来做事务日志的管理,传统的 WAL 采用 HDD 或 SSD 为存储介质,而大内存和新型存储介质

① https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads/2016,9,28

NVM 的出现，WAL 的性能瓶颈转向 CPU 的处理能力[28]，在多核架构下 WAL 遇到了扩展性问题.

一方面当前 KV 系统采用集中式的单 WAL 日志来保证数据可靠性，WAL 需要在更新内存数据之前先做日志持久化，由于单个请求粒度较小，所以对 SSD 存在的大量频繁的小粒度写，具有较大的 I/O 延迟. 当前在多核环境下，系统的事务日志 I/O 延迟直接影响整个系统性能[9]. 基于以字节粒度寻址、低延迟并且数据非易失的 NVM 恰好能够解决事务日志的数据放置问题. 将 NVM 作 WAL 缓存、将 SSD 作 WAL 的持久化存储的多层 WAL 存储设计，提高日志效率成为研究焦点. 另一方面，当系统在面临高并发负载时，虽然 NVM 提供了跟 DRAM 相当的读写延迟且保证了数据非易失，提高日志效率，但在 NVM 上集中式的单日志由于多核对共享日志缓存的竞争而成为系统的扩展性瓶颈[9-10]. 基于新型存储介质 NVM 多核高扩展性的分布式事务日志，备受关注[27-28]. 当前在 KV 存储系统中对事务日志扩展性的研究处于起步阶段，关于它的研究将成为新的趋势.

(3) 高可用性. 当前大数据环境下，负载请求对象的大小层次不齐，对 KV 系统的考验越来越大. 一个 KV 系统在对象的粒度变化较大时仍能保持较理想的性能成为应用负载的性能需求[92]. 在这种情况下，KV 存储系统和文件系统相结合的存储架构成为研究热点，典型工作如 WiscKey[75]、Atlas[77]. 如对于 LSM 系统来说，当 value 的粒度较小时请求被 KV 系统来处理，KV 系统对小粒度请求具有天生优势. 当 value 粒度增加到指定量时，KV 中大粒度的请求 value 导致缓存占用增加，同时频繁的数据合并导致更大的读写放大. 此时，将 KV 中的大粒度 value 进行分离存储，KV 系统中存储 value 的元数据信息. 为了加快对大文件的读取速度，将外部的大 value 文件优化放置在新型存储介质 NVM 上. 因此对请求对象的大小具有弹性存储并保持较高性能的高可用 KV 系统成为研究趋势.

9　总　结

KV 型本地存储系统是 NoSQL 系统中应用最广泛的一种存储引擎，当前多核、大内存和新型持久化存储设备的硬件环境给这一存储引擎提供了很好的性能扩展机会，但同时大数据的应用负载也给它带来了许多挑战. 在这样的软硬件背景下，它成为了越来越多研究者所关注的焦点. 本文以系统研究的视角，对系统各组件的研究现状进行了详细综述和归纳总结. 本文从索引技术的角度阐述了哈希、B+tree 和 LSM-tree 三大索引技术的研究进展，并梳理了相关工作的优缺点，指出了仍存在的问题；从并发控制的视角叙述了传统基于锁的并发控制存在的多核扩展性问题及相关新技术；从事务日志扩展性的角度介绍了非易失型内存 NVM，阐述了传统集中式日志存在的问题及基于 NVM 的分布式日志的相关研究；从数据放置策略的维度叙述了大数据环境下全内存数据放置存在的问题，并重点介绍了针对新型存储介质 SSD 的数据放置策略.

在多核 CPU 和大内存环境下，KV 型 NoSQL 系统中越来越多的数据缓存在内存中，对缓存数据的高效索引以及对索引的并发控制，直接跟多核的性能扩展相关. 哈希索引具有高效的查询效率，基于分数分区的并发控制在哈希索引上具有较好的性能表现，但在单个分区内仍存在传统锁的扩展性限制. 哈希索引在功能上不支持范围查找. 虽然 B+tree 索引提供了很好的范围查找功能和较高的查询效率，但多线程对内存共享数据并发访问时，B+tree 中基于页面节点的锁并发控制机制严重制约着系统的扩展性；基于 LSM-tree 为索引的系统同样支持高效的范围查询，在并发控制方面，虽然在其核心数据结构跳表上采用了多版本的读并发控制，提高了多线程读的高并发，但在写性能方面仍存在全局锁的并发限制问题. HyperLevelDB 消除了 LevelDB 的全局锁，在跳表内部各节点上采用更细粒度的锁，支持多线程并发更新，这提升了多核对共享资源读写高并发，但在面对当前海量数据高速率更新时，加锁带来的线程阻塞和上下文切换开销仍较大. cLSM-tree[7] 在跳表上通过使用 CPU 硬件级指令支持的原子操作 CAS 进行无锁 lock-free 的并发访问，极大的提升了多核环境下系统的读写扩展性，充分发挥了多核多处理器的硬件级性能. 因此，从 CPU 和内存角度来说，KV 型 NoSQL 本地存储系统中索引技术的选择取决于不同索引技术对功能支持的取舍，但不论采用哈希、B+tree 还是 LSM-tree 索引，基于这些索引技术的最佳并发控制是基于多版本的读并发控制和基于无锁的写并发控制，当前典型的工作如基于 B+tree 索引的研究 Bw-tree[8]、基于 LSM-tree 索引的研究 cLSM-tree[7].

新型存储介质 SSD、NVM 对数据放置策略产生了重大影响，表现在事务日志的管理和数据的放置. 在事务日志的管理方面，多核架构下虽然通过采用新型的并发控制技术实现系统的多核高扩展性，但这给 I/O 带来了挑战. 如前所述，WAL 产生大量小粒度的 I/O，它需要保证每次的 KV 请求写内存之前先做日志持久化. 虽然 WAL 是顺序的 I/O，但是这种频繁的小粒度 I/O 导致 HDD 较大的机械寻道开销，SSD 的 Flash 转换层的转换和垃圾回收开销增加. 以字节粒度寻址，有着跟 DRAM 同等读写

延迟的新型存储介质非易失型内存 NVM 的出现，给当前基于磁盘的集中式 WAL 带来了扩展机会.将 NVM 作为事务日志的存储介质，同时设计分布式事务日志，实现了事务日志的多核高扩展性和高效的日志放置.因此，基于 NVM 的分布式事务日志管理策略成为当前系统设计的目标和选择[10,27].在数据放置方面，在基于传统机械磁盘为持久化存储介质的时代，I/O 往往是存储系统的性能瓶颈.而当前，一方面单位内存价格渐渐降低，大容量内存被部署于服务器环境，虽然大内存环境允许将更多的数据驻留在内存，但根据 I/O 五分钟原则，由于数据有冷热特性，数据并不适合长时间全内存存储，而需要持久化.随着闪存、非易失内存技术的不断成熟和发展，高性能持久化存储介质 SSD 被应用于数据库服务器环境，这种新型的存储介质虽然提供了很高的硬件级吞吐率和低延时性能，但它的性能发挥跟对它的读写方式直接相关.SSD 擅长于随机读和顺序写，但随机写会导致它内部的 FTL 层需要做大量的垃圾回收操作，带来额外的开销，而日志结构存储，能够较好的解决小粒度随机写问题.因此，基于日志结构存储的数据放置策略成为众多 KV 系统的选择，如 FAWN-DS[11] 和 MICA[67] 等，这也是 LSM-tree 型系统的设计初衷，如 LSM-trie[90] 和 WiscKey[75] 等.

综合当前对该系统各组件的研究成果和仍存在的问题，我们从日志扩展性、CPU 缓存高效性和高可用性三个维度，对该系统的研究趋势做了预见和展望.我们从 CPU、内存和持久化存储等角度，给出了当前最优的索引技术、数据放置策略、并发控制技术和事务日志管理策略.在多核、大内存和新型持久化存储设备的新硬件环境下，最优的技术能够实现较理想的性能扩展.但在大数据时代，面对大数据的 Volume 和 Velocity 特性，单项技术存在自己的局限性，仍然无法满足各种复杂的应用需求.不同技术之间渐渐出现融合的趋势，但这尚处于起步阶段.可以预见，在未来的研究中，多种数据库技术共存，使用多种存储介质高效异构存储，具有高扩展性和高可用性的通用 KV 型 NoSQL 本地存储系统将成为研究趋势.希望本文能够为 KV 型 NoSQL 本地存储系统的设计者和研究者在选择和优化系统关键技术方面给以启发.

参 考 文 献

[1] Shen De-Rong, Yu Ge, Wang Xi-Te, et al. Survey on NoSQL for management of big data. Journal of Software, 2013, 24(8): 1786-1803(in Chinese)
(申德荣，于戈，王习特等. 支持大数据管理的 NoSQL 系统研究综述. 软件学报，2013，24(8)：1786-1803)

[2] DeCandia G, Hastorun D, Jampani M. Dynamo: Amazon's highly available key-value store//Proceedings of the 21st ACM Symposium on Operating Systems Principles. Stevenson, USA, 2007: 205-220

[3] Chang F, Dean J, Ghemawat S. BigTable: A distributed storage system for structured data//Proceedings of the 7th Symposium on Operating System Design and Implementation. Seattle, USA, 2006: 205-218

[4] Stonebraker M, Cetintemel U. Technical perspective: One size fits all: An idea whose time has come and gone. Communications of the ACM-Surviving the Data Deluge, 2008, 51(12): 76-76

[5] Lakshman A, Malik P. Cassandra: A decentralized structured storage system. ACM SIGOPS Operating System Review, 2010, 44(2): 35-40

[6] Cooper B F, Ramakrishnan R, Srivastava U, et al. PNUTS: Yahoo!'s hosted data serving platform//Proceedings of the VLDB Endowment. Auckland, New Zealand, 2008: 1277-1288

[7] Golan-Gueta G, Bortnikov E, Hillel E, Keidar I. Scaling concurrent log-structured data stores//Proceedings of the 10th European Conference on Computer Systems. Bordeaux, France, 2015: 32:1-32:14

[8] Levandoski J J, Lomet D B, Sengupta S. The Bw-tree: A B-tree for new hardware platforms//Proceedings of the 29th International Conference on Data Engineering. Brisbane, Australia, 2013: 302-313

[9] Hwang Y, Gwak H, Shin D. Two-level logging with non-volatile byte-addressable memory in log-structured file systems//Proceedings of the 12th ACM International Conference on Computing Frontiers. Ischia, Italy, 2015: 38:1-38:2

[10] Wang Tianzheng, Johnson R. Scalable logging through emerging non-volatile memory//Proceedings of the VLDB Endowment. Hangzhou, China, 2014: 865-876

[11] Andersen D, Franklin J, Kaminsky M. FAWN: A fast array of wimpy nodes//Proceedings of the 22nd ACM Symposium on Operating Systems Principles. Big Sky, USA, 2011: 1-14

[12] Lim H, Fan B, Andersen D G, Kaminsky M. SILT: A memory-efficient, high-performance key-value store//Proceedings of the 23rd ACM Symposium on Operating Systems Principles. Cascais, Portugal, 2011: 1-13

[13] Yang Chuan-Hui. Large-Scale Distributed Storage System: The Principle Analytic and Architecture Action. Beijing: China Machine Press, 2013(in Chinese)
(杨传辉. 大规模分布式存储系统：原理解析与架构实战. 北京：机械工业出版社，2013)

[14] O'Neil P, Cheng E, Gawlick D, O'Neil E. The log-structured merge tree. Acta Informatica, 1996, 33(4): 351-385

[15] Pagh R, Rodler F. Cuckoo hashing. Journal of Algorithms, 2004, 51(2): 122-144

[16] Bloom B H. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970, 13(7): 422-426

[17] Anand A, Muthukrishnam C, Kappes S, et al. Cheap and large CAMs for high performance data-Intensive networked systems//Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation. San Jose, USA, 2010: 29-29

[18] Agrawal D, Ganesan D, Sitaraman R. Lazy-adaptive tree: An optimized index structure for flash devices//Proceedings of the VLDB Endowment. Lyon, France, 2009: 361-372

[19] Sears R, Ramakrishnan R. bLSM: A general purpose log structured merge tree//Proceedings of the 2012 ACM SIGMOD

International Conference on Management of Data. Scottsdale, USA, 2012: 217-228

[20] Wang P, Sun G, Jiang S, et al. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD//Proceedings of the 9th European Conference on Computer Systems. Amsterdam, Netherlands, 2014: 16: 1-16:14

[21] Herlihy M, Shavit N. The Art of Multiprocessor Programming. San Francisco, USA: Morgan Kaufmann Publishers Inc., 2008

[22] Duffy J. Concurrent Programming on Windows. New Jersey, USA: AddisonWesley, 2008

[23] Sears R, Brewer E. Segment-based recovery: Write-ahead logging revisited//Proceedings of the VLDB Endowment. Lyon, France, 2009: 490-501

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

[25] Fang R, et al. High performance database logging using storage class memory//Proceedings of the 2011 IEEE 27th International Conference on Data Engineering. Hannover, Germany, 2011: 1221-1231

[26] Huang J, Schwan K, Qureshi M K. NVRAM-aware logging in transaction systems//Proceedings of the VLDB Endowment. Hangzhou, China, 2014: 389-400

[27] Arulraj J, Pavlo A, Dulloor S R. Let's talk about storage & recovery methods for non-volatile memory database systems//Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. Melbourne, Australia, 2015: 707-722

[28] Kim W H, Kim J, Baek W, et al. NVWAL: Exploiting NVRAM in write-ahead logging//Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems. Istanbul, Turkey, 2016: 385-398

[29] Johnson R, Pandis I, Stoica R, et al. Aether: A scalable approach to logging//Proceedings of the VLDB Endowment. Singapore, 2010: 681-692

[30] Islam N S, Wasi-Ur-Rahman M, Lu X, et al. High performance design for HDFS with byte-addressability of NVM and RDMA//Proceedings of the 2016 International Conference on Supercomputing. Istanbul, Turkey, 2016: 8:1-8:14

[31] Xu J, Swanson S. Nova: A log-structured file system for hybrid volatile/non-volatile main memories//Proceedings of the 14th USENIX Conference on File and Storage Technologies. Santa Clara, USA, 2016: 323-338

[32] Graefe G. The five-minute rule twenty years later, and how flash memory changes the rules//Proceedings of the 3rd International Workshop on Data Management on New Hardware. Beijing, China, 2007: 6:1-6:9

[33] Sheehy J, Smith D. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data. Bellevue, USA: Basho Technologies, 2010

[34] Nath S, Kansal A. FlashDB: Dynamic self-tuning database for NAND flash//Proceedings of the 6th International Conference on Information Processing in Sensor Network. Cambridge, USA, 2007: 410-419

[35] Debnath B, Sengupta S, Li J. SkimpyStash: RAM space skimpy key-value store on flash//Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. Athens, Greece, 2011: 25-36

[36] Debnath B, Sengupta S, Li J. ChunkStash: Speeding up inline storage deduplication using flash memory//Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference. Boston, USA, 2010: 16-16

[37] Debnath B, Sengupta S, Li J. FlashStore: High throughput persistent key-value store//Proceedings of the VLDB Endowment. Singapore, 2010: 1414-1425

[38] Ramakrishnan R, Gehrke J. Database Management Systems. New York, USA: McGraw-Hill Higher Education, 2000

[39] Cormen T H, Leiserson C E, Rivest R L, Stein C. Introduction to Algorithms. Cambridge, UK: MIT Press, 1990

[40] Roh H, Park S, Kim S, et al. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives //Proceedings of the VLDB Endowment. Seattle, USA, 2011: 286-297

[41] Yang Jun, Wei Qingsong, Chen Cheng, et al. NV-Tree: Reducing consistency cost for NVM-based single level systems //Proceedings of the 13th USENIX Conference on File and Storage Technologies. Santa Clara, USA, 2015: 167-181

[42] Chen Shimin, Jin Qin, Persistent B+-trees in non-volatile main memory//Proceedings of the VLDB Endowment. Hawaii, USA, 2015: 786-797

[43] Shahvarani A, Jacobsen H A. A hybrid B+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms//Proceedings of the 2016 International Conference on Management of Data. San Francisco, USA, 2016: 1523-1538

[44] Sewall J, Chhugani J, Kim C, et al. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on manycore processors//Proceedings of the VLDB Endowment. Seattle, USA, 2011: 795-806

[45] Braginsky A, Petrank E. A lock-free B+ tree//Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures. Pennsylvania, USA, 2012: 58-67

[46] Levandoski J, Lomet D, Sengupta S, et al. Indexing on modern hardware: Hekaton and beyond//Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. Utah, USA, 2014: 717-720

[47] Oukid I, Lasperas J, Nica A, et al. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory//Proceedings of the 2016 International Conference on Management of Data. San Francisco, USA, 2016: 371-386

[48] Rosenblum M, Ousterhout J K. The design and implementation of a log-structured file system. ACM Transactions on Computer Systems, 1992, 10(1): 26-52

[49] Escriva R, Wong B, Sirer E G. HyperDex: A distributed, searchable key-value store//Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. Helsinki, Finland, 2012: 25-36

[50] Pugh W. Skip lists: A probabilistic alternative to balanced trees. Communications of the ACM, 1990, 22(6): 668-676

[51] Li Yinan, He Bingsheng, Yang Robin Jun, et al. Tree indexing on solid state drives//Proceedings of the VLDB Endowment. Singapore, 2010: 1195-1206

[52] Chazelle B, Guibas L J. Fractional cascading: I. A data structuring technique. Algorithmica, 1986, 1(2): 133-162

[53] Thonangi R, Babu S, Yung J. A practical concurrent index for solid-state drives//Proceedings of the 21st ACM International Conference on Information and Knowledge Management. Maui, USA, 2012: 1332-1341

[54] Bender M A, Fineman J T, Gilbert S, Kuszmaul B C. Concurrent cache-oblivious B-trees//Proceedings of the 17th

Annual ACM Symposium on Parallelism in Algorithms and Architectures. Vegas, USA, 2005: 228-237
[55] Bender M A, Farach-Colton M, Fineman J T, et al. Cache-oblivious streaming B-trees//Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures. San Diego, USA, 2007: 81-92
[56] Buchsbaum A L, Goldwasser M, Venkatasubramanian S, Westbrook J R. On external memory graph traversal//Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms. San Francisco, USA, 2000: 859-860
[57] Rodeh O, Bacik J, Mason C. BTRFS: The Linux B-tree filesystem. ACM Transactions on Storage, 2013, 9(3): 1-32
[58] Kemper A, Neumann T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots//Proceedings of the 2011 IEEE 27th International Conference on Data Engineering. Hannover, Germany, 2011: 195-206
[59] Liu Kai-Jie. The Design and Implementation of a Local Key-Value Store on SSD[M. S. dissertation]. Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 2012(in Chinese)
(刘凯捷. 基于 SSD 的本地 key-value 型存储系统的设计和实现[硕士学位论文]. 中国科学院计算技术研究所, 北京, 2012)
[60] Pandis I, Tözün P, Johnson R, Ailamaki A. PLP: Page latch-free shared-everything OLTP//Proceedings of the VLDB Endowment. Seattle, USA, 2011: 610-621
[61] Larson P-Å, Blanas S, Diaconu C, et al. High-performance concurrency control mechanisms for main-memory databases //Proceedings of the VLDB Endowment. Seattle, USA, 2011: 298-309
[62] Mitchell C, Geng Y, Li J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store//Proceedings of the 2013 Conference on USENIX Annual Technical Conference. San Jose, USA, 2013: 103-114
[63] Pavlo A, Curino C, Zdonik S. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems//Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. Scottsdale, USA, 2012: 61-72
[64] Metreveli Z, Zeldovich N, Kaashoek M F. CPHash: A cache-partitioned hash table//Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New Orleans, USA, 2012: 319-320
[65] Tu S, Zheng W, Kohler E, et al. Speedy transactions in multicore in-memory databases//Proceedings of the 24th ACM Symposium on Operating Systems Principles. Farmington, USA, 2013: 18-32
[66] Blott M, Karras K, Liu L, et al. Achieving 10 Gbps line-rate key-value stores with FPGAs//Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing. San Jose, USA, 2013: 1:1-1:6
[67] Lim H, Han Dongsu, Andersen D G. Kaminsky M. MICA: A holistic approach to fast in-memory key-value storage//Proceedings of the 11th Symposium on Networked Systems Design and Implementation. Seattle, USA, 2014: 429-444
[68] Caulfield A M, et al. Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing//Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. New Orleans, USA, 2010: 1-11
[69] Johnson R, et al. Scalability of write-ahead logging on multicore and multisocket hardware. The International Journal on Very Large Data Bases, 2012, 21(2): 239-263
[70] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, 21(7): 558-565
[71] Pelley S, Wenisch T F, Gold B T, Bridge B. Storage management in the NVRAM era//Proceedings of the VLDB Endowment. Hangzhou, China, 2014: 121-132
[72] Chen Shimin, FlashLogging: Exploiting flash devices for synchronous logging performance//Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data. Rhode Island, USA, 2009: 73-86
[73] Gao Shen, Xu Jianliang, et al. PCMLogging: Reducing transaction logging overhead with PCM//Proceedings of the 20th ACM International Conference on Information and Knowledge Management. Glasgow, UK, 2011: 2401-2404
[74] Stonebraker M, Hachem N, Helland P. The end of an architectural era//Proceedings of the 33rd International Conference on Very Large Data Bases. Vienna, Austria, 2007: 1150-1160
[75] Lu Lanyue, Pillai T S, Arpaci-Dusseau A C, Arpaci-Dusseau R H. WiscKey: Separating keys from values in SSD-conscious storage//Proceedings of the 14th USENIX Conference on File and Storage Technologies. Santa Clara, USA, 2016: 133-148
[76] Beaver D, Kumar S, Li H C, et al. Finding a needle in Haystack: Facebook's photo storage//Proceedings of the 9th Symposium on Operating Systems Design and Implementation. Vancouver, Canada, 2010: 47-60
[77] Lai Chunbo, Jiang Song, Yang Liqiong, et al. Atlas: Baidu's key-value storage system for cloud data//Proceedings of the 31st International Conference on Massive Storage Systems and Technology. Santa Clara, USA, 2015: 1:1-1:14
[78] Armstrong T G, Ponnekanti V, Borthakur D, Callaghan M. LinkBench: A database benchmark based on the facebook social graph//Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. New York, USA, 2013: 1185-1196
[79] Shetty P, Spillane R, Malpani R, et al. Building workload-independent storage with VT-Trees//Proceedings of the 11th USENIX Symposium on File and Storage Technologies. San Jose, USA, 2013: 17-30
[80] Kim H, Agrawal N, Ungureanu C. Revisiting storage for smartphones//Proceedings of the 10th USENIX Symposium on File and Storage Technologies. San Jose, USA, 2012: 17-17
[81] Min C, Kim K, Cho H, et al. SFS: Random write considered harmful in solid state drives//Proceedings of the 10th USENIX Symposium on File and Storage Technologies. San Jose, USA, 2012: 12-12
[82] Lee C, Sim D, Hwang J, Cho S. F2FS: A new file system for flash storage//Proceedings of the 13th USENIX Symposium on File and Storage Technologies. Santa Clara, USA, 2015: 273-286
[83] Arpaci-Dusseau R H, Arpaci-Dusseau A C. Operating Systems: Three Easy Pieces. 0.9 Edition. Madison, USA: Arpaci-Dusseau Books, 2014
[84] Ouyang J, Lin S, Jiang S, et al. SDF: Software-defined flash for web-scale internet storage systems//Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. Salt Lake City, USA, 2014: 471-484
[85] Tanenbaum A S, Bos H. Modern Operating Systems. 4th Edition. Upper Saddle River, USA: Prentice Hall Press, 2014
[86] Wu G, He X, Eckart B. An adaptive write buffer management scheme for flash-based SSDs. ACM Transactions on

Storage, 2012, 8(1): 1:1-1:24
[87] Woodhouse D. JFFS: The Journaling Flash File System. Ottawa: Red Hat Inc., 2011
[88] Card R, Ts'o T, Tweedie S. Design and implementation of the second extended filesystem//Proceedings of the 1st Dutch International Symposium on Linux. Amsterdam, Netherlands, 1994: 1:1-1:13
[89] Vo H T, Wang S, Agrawal D, et al. LogBase: A scalable log-structured database system in the cloud//Proceedings of the VLDB Endowment. Istanbul, Turkey, 2012: 1004-1015
[90] Wu Xingbo, Xu Yuehai, Shao Zili, Jiang Song. LSM-trie: An LSM-tree-based ultra-large key-value store for small data //Proceedings of the USENIX Annual Technical Conference. Santa Clara, USA, 2015: 71-82
[91] Marmol L, Sundararaman S, Talagala N, Rangaswami R. NVMKV: A scalable, lightweight, FTL-aware key-value store //Proceedings of the USENIX Annual Technical Conference. Santa Clara, USA, 2015: 207-219
[92] Zhang Z, Yue Y, He B, et al. Pipelined compaction for the LSM-tree//Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium. Phoenix, USA, 2014: 777-786
[93] Atikoglu B, Xu Y, Frachtenberg E, et al. Workload analysis of a large-scale key-value store//Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems. London, UK, 2012: 53-64
[94] Cooper B, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB//Proceedings of the 1st ACM Symposium on Cloud Computing. Indianapolis, USA, 2010: 143-154
[95] Jermaine C, Omiecinski E, Yee W G. The partitioned exponential file for database storage management. The International Journal on Very Large Data Bases, 2007, 16(4): 417-437
[96] Spillane R P, Shetty P J, Zadok E, et al. An efficient multi-tier tablet server storage architecture//Proceedings of the 2nd ACM Symposium on Cloud Computing. Cascais, Portugal, 2011: 1:1-1:14

MA Wen-Long, born in 1991, Ph. D. candidate. His research interests include in-memory database and concurrent data structures.

ZHU Yu-Qing, born in 1983, Ph. D., assistant professor. Her research interests include distributed systems and data management.

JIANG De-Jun, born in 1982, Ph. D., associate professor. His research interests include storage systems, memory architecture, operating systems, and distributed systems.

XIONG Jin, born in 1968, Ph. D., professor, Ph. D. supervisor. Her research interests include big data storage and management, distributed file systems, SSD- and NVM-based storage systems.

ZHANG Li-Xin, born in 1971, Ph. D., professor, Ph. D. supervisor. His research interests include data center computing systems, advanced cache/memory system, architectural simulators, parallel computing, performance evaluation, and workload characterization.

MENG Xiao, born in 1987, engineer. His research interests include distributed systems and data management.

BAO Yun-Gang, born in 1980, Ph. D., professor, Ph. D. supervisor. His research interests include high performance computer architecture, operating system and performance analysis.

Background

The emergence of new hardware with multi-core CPUs, larger memory and high-performance persistent storage devices like SSD provides a rare opportunity to exploit the performance for KV storage system. However, the architecture of KV storage system has to be reconsidered. The bottleneck of storage system generally consists of CPU bound and I/O bound. As for CPU bound, multi-core hardware supports more concurrent tasks, but the traditional lock-based concurrency control results in more latency due to the frequent multi-threaded blocking and context switch. On the other hand, modern CPU Cache owns larger capacity, which provides a chance to exploit performance of KV storage system for small granularity workloads. However, the multi-threaded access to the share data with massive concurrency may result in Cache incoherence between CPU Caches, this leads to more Cache invalidation especially in NUMA environment. As for I/O bound, although new storage devices such as SSD provide a lower latency and high throughput in hardware level, the ultimate performance of SSD is closely related to the way of writing data due to the SSD's special properties. For example, concurrent random write with small granularity can lead to huge amount of data fragmentation and exacerbate overhead of the FTL garbage collection. Therefore, in order to fully exploit the performance of KV storage system under new hardware and big data workload environment, the architecture of KV database system needs to be reconsidered to make its several components well-balanced.

This paper reviews the state-of-the-art research on indexing, concurrency control, logging and data persistence methods for local KV stores, focusing on how these works solve the challenges posed by the new hardware technology and by the big data applications. This paper is not limited to the review of certain algorithm or technology about local KV storage system, but the comprehensive survey of local KV store in view of system research. We also make a summary of this paper and prospect at the end.

This work is supported in part by the State Key Development Program for Basic Research of China (Grant No. 2014CB340402), the National Natural Science Foundation of China (Grant No. 61303054 and No. 61420106013) and the Shandong Provincial Natural Science Foundation (Grant No. ZR2016FM41).