计算机科学
COMPUTER SCIENCE

基于LSM树的键值存储系统技术研究综述

吕萌, 华文镝, 谢平

引用本文

吕萌, 华文镝, 谢平. 基于LSM树的键值存储系统技术研究综述[J]. 计算机科学, 2023, 50(8): 1-15.

LYU Meng, HUA Wendi, XIE Ping. Key Value Storage Technology Based on LSM-tree:A Survey[J]. Computer Science, 2023, 50(8): 1-15.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

基于LDPC读延迟的刷新和副本结合策略优化方案

Policy Optimization Scheme of Refresh and Duplication Combination Based on LDPC Read Delay

计算机科学, 2023, 50(7): 38-45. https://doi.org/10.11896/jsjkx.220900179

面向NoSQL数据库的JSON文档异常检测与语义消歧模型

Outlier Detection and Semantic Disambiguation of JSON Document for NoSQL Database

计算机科学, 2021, 48(2): 93-99. https://doi.org/10.11896/jsjkx.200900039

区块链系统的数据存储与查询技术综述

Survey of Data Storage and Query Techniques in Blockchain Systems

计算机科学, 2018, 45(12): 12-18. https://doi.org/10.11896 / j.issn.1002-137X.2018.12.002

一种面向下一代互联网的广域网智能存储系统

WAN Intelligent Storage System for Next Generation Internet

计算机科学, 2010, 37(10): 279-282.

对等网络中搜索策略的研究

计算机科学, 2003, 30(9): 94-96.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.220900178

基于 LSM 树的键值存储系统技术研究综述

吕　萌　华文镝　谢　平
1 青海师范大学计算机学院　西宁 810016
2 青海师范大学网络信息管理中心　西宁 810016
3 青海省物联网重点实验室　西宁 810008
4 省部共建藏语智能信息处理及应用国家重点实验室　西宁 810008
5 高原科学与可持续发展研究院　西宁 810016
(2256526997@qq.com)

摘　要　键值存储是数据库最简单的组织形式。在数据密集型的应用场景中,键值存储系统发挥着关键的作用。随着对及时数据分析需求的增加,良好的系统性能变得越来越重要。目前大多数键值存储系统的存储引擎都是日志结构合并树(Log-Structured Merge Tree,LSM 树)。因具有卓越的写性能,LSM 树被广泛应用于写密集型的场景和现代 NoSQL 系统的存储层。与传统的 B 树相比,LSM 树采用顺序写入的访问模式,并使用内存缓冲区来批处理新的写入线程,因此 LSM 树具有更大的写优势。然而,数据的重复读写和不必要的压缩操作导致了 LSM 树的读写放大问题,从而严重影响了系统的性能,尤其在数据密集型的应用场景。如今,研究人员做了大量工作来缓解这些问题,文中研究了影响 LSM 树性能的各个因素,搜集了大量提升基于 LSM 树的键值系统性能的文献,并对其加以整理和分类,讨论它们的优势和权衡,使读者可以了解基于 LSM 树的存储技术及其优化策略,最后调查了几个具有代表性的基于 LSM 树的键值存储技术并讨论了潜在的未来研究方向。

关键词:LSM 树;NoSQL;存储管理;键值系统;数据检索
中图法分类号　TP393

Key Value Storage Technology Based on LSM-tree:A Survey

LYU Meng,HUA Wendi and XIE Ping
1 School of Computer Science,Qinghai Normal University,Xining 810016,China
2 Network Information Management Center,Qinghai Normal University,Xining 810016,China
3 Key Laboratory of Internet of Things of Qinghai Province,Xining 810008,China
4 State Key Laboratory of Tibetan Intelligent Information Processing and Application,Xining 810008,China
5 Academy of Plateau Science and Sustainability,Xining 810016,China

Abstract　Key-value storage is the simplest form of database organization and it plays a key role in data-intensive application scenarios. With the increasing demand for timely data analysis,good system performance becomes more and more important. At present,the storage engine of most key-value storage systems is the log-structured merge tree(LSM-tree). Because of its excellent write performance,the LSM-tree is widely used in the write intensive scenes and the storage layer of modern NoSQL system. Compared to the traditional B-tree,LSM-tree adopts sequential write access mode. At the same time,it uses memory buffer to batch new write threads,so it has greater write advantages. Nevertheless,repeated reading and writing of data and unnecessary compression operations lead to the problem of read and write amplification of the LSM-tree. Finally,these problems seriously affect the performance of the system,especially in the data-intensive application scenarios. Nowadays,researches have make great efforts to solve the problems. Firstly,this paper investigates various factors that affect the performance of the LSM-tree,collects a lot of literature on improving the performance of LSM tree-based key-value systems,organizes and categorizes them. Then it discusses their advantages and tradeoffs to enable readers to understand LSM tree-based storage technologies and their optimization strategies. Finally,several representative LSM tree-based key-value storage technologies are surveyed and some potential future research directions are discussed.

Keywords　LSM-tree,NoSQL,Storage management,Key-value system,Data retrieval

到稿日期:2022-09-19　返修日期:2023-02-26
基金项目:国家自然科学基金(61762075);青海省科技厅重点研发与转化计划项目(2021-GX-112)
This work was supported by the National Natural Science Foundation of China(61762075) and Key R&D and Transformation Project of Qinghai Province(2021-GX-112).
通信作者:谢平(xieping@qhnu.edu.cn)

1 引言

随着现代科技水平的日益提高，键值存储系统在各类数据密集型的系统中发挥着至关重要的作用。目前的大多数键值存储系统都使用 LSM 树或者 B 树作为主要数据结构存储数据，这两种不同的数据结构分别适用于不同类型的工作负载：RocksDB1)，LevelDB2)，Dynamo[1] 和 Cassandra[2] 等数据库都使用 LSM 树，而 MongoDB[3] 以及其他的数据库都使用 B 树以及它的衍生树。相较于 B 树等其他数据结构，LSM 树可以维持顺序写入的特性，可以充分发挥固态存储设备和硬盘的性能优势。LSM 树的这种设计带来了较好的写性能、较高的空间利用率和故障恢复等一系列的好处，因此 LSM 树被应用于多种写密集型的应用场景当中。

LSM 树[4]因其较高的写性能获得了较高关注，并被广泛地应用在现代 NoSQL 系统的存储层，但其频繁的日志更新操作会导致额外的开销。在刷新数据和进行合并操作时，系统中已经存在的数据会有多次重复写入的操作，而随着这些无效数据的写入，系统的写放大也会显著增加。同时，这一过程也会带来读放大，进而影响 LSM 树的读性能。由于 LSM 树是一种横跨内存和磁盘的分层结构，在 LSM 树的磁盘部分，查询数据往往需要逐层向下查询，一次数据的查询会带来对磁盘的多次读写操作，反复的擦除写入很容易缩短固态硬盘等存储介质的寿命，从另一个角度来说，这对整个系统的性能也有很大损耗。与 B 树相比，LSM 树虽然能在较长的时间内提供对文件的高速增删，但是在需要高效查询时性能相对较低，因此 LSM 树更适用于写入比查询操作更频繁的系统。LSM 树的这些特性导致其在某些特定的场景下，例如读取和更新频率较高的键值存储系统中，并不能充分发挥其性能优势。在优化读性能等其他方面，研究人员还需要做大量工作。

研究人员针对不同的应用场景对 LSM 树本身进行了不同的优化。本文对不同的优化方案进行了分类和详细阐述，包括解决写放大、压缩合并策略、硬件、存储介质以及特殊的工作负载。此外，本文列举了几种具有代表性的 NoSQL 系统来对 LSM 树的实际应用做一个简单说明，包括 LevelDB，RocksDB，HBase3)。本文的主要内容包括以下几个部分：第 2 节对传统和现代 LSM 树做基本介绍，提出了几种改进 LSM 树的技术手段，阐述了当前基于 LSM 树的键值存储系统的工作原理，并以 LevelDB 为例说明其工作流程；第 3 节对近来年提升 LSM 树性能的优化方法进行了分类；第 4 节从多个方面对这些优化方法进行了对比分析；第 5 节介绍了 LSM 树的几个应用场景；第 6 节针对键值存储系统存在的问题以及未来可能的发展方向进行总结和展望。

2 相关工作

传统关系型数据库通常在读性能上有较高的要求，通过二分查找、哈希、B+树和外部文件等方式组织数据能够在复杂的读场景下高效地进行查找，但这些方式都将总体的结构信息强加在了数据上，而数据必须按照特定的方式存储。当需要保存数据到磁盘时就有一个明显的缺陷：逻辑上相距很近的数据在物理上却可能相距很远，这就可能造成大量的磁盘随机写，严重影响写性能。也就是说，传统关系型数据库实现高性能读操作的代价是有相对低效的写操作。基于磁盘随机操作慢、顺序读写块的基本情况，若要提高写操作性能，最好避免随机写而设计成顺序写。一种简单的方法是将数据直接追加到文件而不对存储位置做特殊要求，日志中经常使用这个策略，因为它们完全是顺序的，所以有较高的写性能。但是与要求较高的读操作性能面临的问题类似，这种做法会导致读一些数据花费的时间比写操作更多，因为查找需要扫描所有数据，直到找到所需的内容。在高性能读操作与写操作不可兼得的情况下，需要面对不同的场景做出取舍，LSM 树就是一个权衡的产物。本节分别对传统和现代的 LSM 树进行介绍，并分析了现代 LSM 树优化的相关技术手段。

2.1 传统的 LSM 树

LSM 树是一种支持高效索引的数据持久性存储结构，它通过聚合内存中的多个更新操作并将其批量刷新到外存中，从而将随机写转换为顺序写以提升写性能。最原始的 LSM 树是将传统的关系型数据库使用的单个整体查找结构变换为多个相似的将数据顺序保存的有序文件。LSM 树通过这些有序文件实现了管理一组索引文件而不是单一的索引文件，同时对它们进行批量的数据更新，充分利用内存来存储近期或常用数据以提高读效率，利用硬盘存储不常用数据以减少存储代价。图 1 中给出了 LSM 树的基本结构。

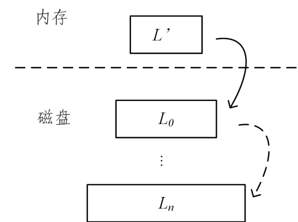


图 1 原始 LSM 树的结构

Fig. 1 Original structure of LSM-tree

从图 1 可以观察到，LSM 树是一个和树一样上小下大的多层结构。最上层位于内存中的 L' 层，其中保存了所有最近写入的数据，这个内存结构是有序的，可以随时随地更新并支持随机查询，剩下的 $L_0, L_1, \cdots, L_n$ 层都在磁盘上，每一层都是根据键的大小排列的顺序结构。当系统接收写操作的请求时，首先将数据记录在预写日志(Write Ahead Log，WAL)中，以便发生故障时恢复数据，随后将数据追加到内存中的 L' 层。

当 LSM 树的数据量达到一定的大小时，就会触发 compaction 操作。从图 2 可以看出，compaction 操作主要分为两种类型：一种是 immutable memtable 传入磁盘的压缩过程，另一种是在磁盘上发生的 SSTable 之间的压缩过程。其中，后者是主要的压缩过程。在写密集型的键值存储系统中，

1) RocksDB. [OL]. [2022-05-27] http://rocksdb.org/

2) LevelDB. [OL]. [2022-05-27] http://leveldb.org/

3) HBase. [OL]. [2022-05-27] https://hbase.apache.org/

immutable memtable 需要将数据刷到磁盘中以保证后续的写入操作。随着写入操作的执行，磁盘中会产生大量 SSTable 文件，各个文件之间的键（key）可能会重叠，系统在读取磁盘上的文件时会耗费大量的时间，因此 SSTable 文件之间的压缩过程是必不可少的。以图 2 为例，L_0 层的 3 个 SSTable 参与了压缩操作，在执行完 compaction 操作后它们被合并到 L_1 层的第一个 SSTable 中，避免了重复数据的读取，可以加速读操作的执行过程。

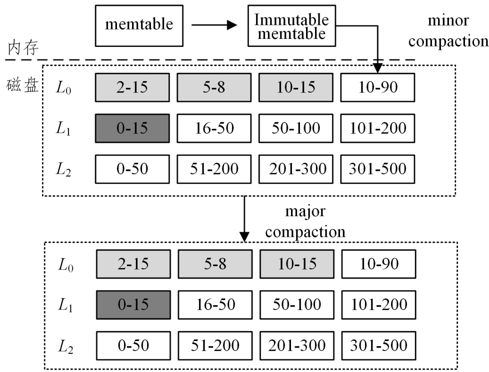


图 2　两类压缩操作的流程

Fig. 2　Flow of two types of compression operations

从 LSM 树的结构中可以推断出，最新的数据在内存中，最原始的数据在 L_i 层，因此查询也是先在内存中查询，如果没有查找到数据则逐层向下查询。除了 L_0 层，磁盘中的每层数据都是有序的，因此可以用效率较高的二分查找，但是随着层级的增加，读操作会越来越慢。为了解决这个问题，可以引入页缓存、布隆过滤器和周期执行 compaction 合并等优化手段。

2.2　改进的 LSM 树

尽管 LSM 树的应用广泛，但 LSM 树本身也存在着读放大和写放大的问题。由于 LSM 树的查找是按照自上而下的顺序进行的，一次目标数据的查询过程往往会有不必要的数据读取操作，这就引起了读放大。在读取数据时需要经过多次读出和写入的过程才能获得目标数据，在这一过程中，非目标数据的写入就会引起写放大。

2.2.1　相关技术手段

为了提升读取性能，研究人员向 SSTable 中引入了布隆过滤器来加快键值对（key-value）数据的查询。当一个元素被加入集合时，通过 k 个散列函数将这个元素映射成一个位数组中的 k 个点，把它们置为 1。检索时只需要看这些点是否都为 1 就大概知道集合中是否存在该元素；如果这些点有任何一个 0，则被检元素一定不存在；如果都是 1，则被检元素很可能存在。相较于其他的数据结构，布隆过滤器在空间和时间上都有巨大的优势，但其本身也存在误判率的问题。一种比较有效的解决办法是将出现误判率的数据元素单独保存起来，如果数据量较少，就可以将它们存储到一个散列表中。文献[5]中介绍了多种布隆过滤器的优化方案和各自的应用场景，如拆分向量的布隆过滤器、改进结构的布隆过滤器和优化哈希策略的布隆过滤器等。

面对海量数据，数据的访问需要经过磁盘的多次 I/O 操作，数据的不断擦除和写入会严重影响磁盘的寿命，研究人员也使用了多种索引结构来降低磁盘的性能损失和快速访问数据。PFHT[6] 维护了两个索引结构，一个是基于布谷鸟哈希的主表，另一个是用于连接哈希表的存储表。当插入键值对时，PFHT 首先检查是否有可能将其插入到最多只有一个位移的主表，如果不可能，则将该键值对插入到存储器中。在 GHStore[7] 中，系统将访问频率较高的数据索引存储在 GH-map 的数据结构中，便于在更新频率较快的场景中迅速查找目标数据。当重复数据较多时，也可以使用缓存技术来合理维护数据。DualKV[8] 提出对频繁更新的数据采用快速写的方式，对其他数据项进行慢写，同时在非易失性存储器（NVM）[9] 上建立键值缓存，这样不仅可以防止掉电导致的缓存数据丢失，还可以提高缓存命中率。

LSM 树有两种合并策略，分别是 leveling 和 tiering 合并策略。在 leveling 合并策略中，每一层的数据被划分成为多个标有各自键的范围的 SSTable。由于磁盘中第一层的数据是直接从内存中取得的，因此该层的 SSTable 并未被分区。在 L_i 层的 SSTable 合并到 L_{i+1} 层时，系统会选择 L_{i+1} 层所有与 L_i 层重叠的 SSTable，并在 L_{i+1} 层生成新的 SSTable，随后将旧的 SSTable 回收。这种 leveling 合并策略能够减少读放大和空间放大，但会造成一定的写放大。Tiering 合并策略有垂直分组和水平分组两种方案，在这两组方案中，每一层的 SSTable 都被组织成多个 SSTable 组，垂直分组的方案将具有重叠范围的 SSTable 分在一起，各个 SSTable 组相互之间键的范围都不相交；而水平分组方案中的数据被划分为固定大小的 SSTable 来直接作为一个逻辑组，每一层有一个“动态 SSTable 组”用于接收上一层由重叠 SSTable 合并生成的新 SSTable。Tiering 虽然能够减小写放大，但是会以牺牲读放大和空间放大为代价。因此 Dostoevsky[10] 采用了一种 Lazy leveling 的策略，它对低层数据采取 tiering 合并策略，对高层数据采取 leveling 合并策略，从而权衡了读写性能。

2.2.2　LevelDB

LevelDB 是一个基于 LSM 树存储引擎的高效的键值存储系统。构成 LevelDB 的静态结构包括内存中的 memtable 和 immutable memtable 以及磁盘上的几种主要文件：current 文件、manifest 文件、log 文件以及 SSTable 文件。当 memtable 写入的数据达到它的阈值时会自动转换为不可修改的 immutable memtable，为后续合并 SSTable 文件做准备。这些数据转存到磁盘后内存中会生成新的 memtable。memtable 的底层采用跳表 SkipList[11] 实现，以方便用户进行查找删除等操作。manifest 文件记录了各个 SSTable 文件的管理信息、分别属于哪一层和哪一个范围，以及最小的键和最大的键分别是多少。随着压缩操作的进行，SSTable 文件也会发生变化，会有新的 SSTable 产生，也会有旧的 SSTable 被删除，manifest 文件也随之变化，同时也会生成新的 manifest 文件来记录这种变化。current 文件用来记录当前 manifest 的文件名，方便用户查看当前的数据变化。Log 文件在系统中的作用主要是确保系统崩溃后恢复时不丢失数据。每当写入一条键值对记录，LevelDB 会先向 log 文件中写入，随后将记录插入到 memtable 中。在这个过程中只涉及一次磁盘追加

写和内存中 skiplist 的插入操作，因此 LevelDB 写入速度非常快。由于最初的写入记录是保存在内存中的，假如没有 log 文件，在系统崩溃时，内存中的数据会因还没有来得及写入到磁盘而丢失。SSTable 文件是 LSM 树中有序键值对的集合，它是分层组织的一批文件。SSTable 由两部分组成：一部分是数据存储区，里面包含多个 datablock，用于存放实际的键值对数据；另一部分是数据管理区，包括 filter blocks，metaindex blocks，index blocks 和 footer，这一部分用于提供一些索引指针等管理数据，目的是更快速、便捷地查找相应的记录。

图 3 给出了 LevelDB 在写入数据时的主要传输过程。当 memtable 插入的数据占用的内存达到一个阈值后，需要将内存的记录导出到外存文件中。此时，LSM 树会生成新的 log 文件和 memtable，原来的 memtable 就转变为只读的 immutable memtable，且不能进行写入或删除操作，新到来的数据被记入新的 log 文件和 memtable 中，并重复上述操作。LevelDB 后台调度程序会将 immutable memtable 导出到磁盘中的第一层，形成新的 SSTable 文件。因此，SSTable 是由内存中的数据不断导出并进行 compaction 操作后形成的。

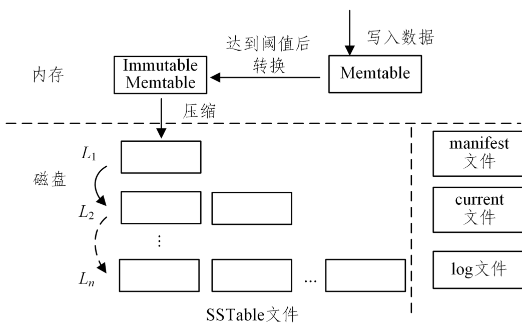


图 3　基于 LSM 树的 LevelDB 的数据传输过程

Fig. 3　Data transmission process of LevelDB based on LSM-tree

总体来说，LSM 树将修改的数据保存在内存中，达到一定数量之后再将要修改的数据批量写入磁盘，在写入的过程中与原来磁盘中存在的数据合并。LSM 树利用批量写入解决了随机写入的问题，其核心特点是利用顺序写来提升写性能。由于 LSM 树的分层设计会造成读性能的降低，但是通过牺牲小部分读性能来提高写性能是值得的。

3　基于 LSM 树的键值存储系统性能优化方法

针对键值存储系统的优化方法有多种分类方式：从存储介质上可以分为单一存储介质的键值存储系统和混合存储介质的键值存储系统；从存储结构上可以分为基于 LSM 树、B+树，以及其他多种结构的键值存储系统。国内外研究者对诸多优化方法进行了不同的分类，文献[12]将 LSM 树的优化方法分为 6 类：写放大、压缩操作、硬件、特殊的工作负载、自适应调节和二级索引。通过对多种基于 LSM 树的键值系统优化方案的分析，本文进行了如下分类。

3.1　基于写放大的优化策略

目前的绝大部分键值系统主要采用 LSM 树结构，它是一种分层、有序、面向磁盘的存储结构。LSM 树最大的特点就是牺牲部分读性能并将其转化为写性能，因此围绕着 LSM 树的核心思想就可以对其进行优化，从而使整个键值系统的性能达到最优。下面整理了几种具有代表性的针对 LSM 树的写放大问题的优化方法。

3.1.1　WB-tree

Amur 等提出了一种可以应用于无序键值存储的写缓冲树(WriteBuffer-tree，WB 树)[13]，它是一种支持批量的插入和删除操作的新型写优化数据结构，可以用于在无序的键值存储中实现每个节点的存储，在提高插入性能的同时可以保持快速的随机读取访问，可以被认为是具有垂直分组的分区分层设计的一种变体。相较于传统的 LSM 树，WB 树进行了如下优化操作：

(1)它依赖于散列分区来实现负载的平衡，使得每个 SSTable 组大致存放相同规模的数据。

(2)它将 SSTable 组织成一个与 B+树相似的结构来实现自身的平衡，从而最小化 LSM 树的高度。因此相较于当前在很多高性能键值存储中使用的 LSM 树，WB 树可以提供更快的写入速度。

3.1.2　LWC-tree

轻量级压缩树(Light-Weight Compaction tree，LWC 树)[14]同样采用类似的垂直分组分区分层设计，它进一步提出了一种实现 SSTable 组负载均衡的方法。在垂直分组的方案下，SSTable 的大小不是固定的，每次新产生的 SSTable 是根据下一层 SSTable 组键的重叠范围生成的，而不是 SSTable 本身的大小。总的来说，如果一个 SSTable 组包含的数据条目较多，为实现负载均衡，LWC 树会在 SSTable 合并后缩小这个组键的范围，同时扩大其兄弟组键的范围。

为了减少过度 I/O 引起的写放大，LWC 树受键值分离策略的启发提出了一种“轻量级压缩”的方法。压缩时，LWC 树根据下一层重叠表中键的范围将数据划分为段并添加到相应的数据管理单元 DTable 中，同时合并少量的元数据。由于 SSTable 的元数据大小远比 SSTable 本身小得多，因此只合并元数据会显著减少压缩过程中所需操作的数据量，从而加快压缩进程。在此过程中，LWC 树首先将 L_i 层的 DTable 读取到内存中，其中包含它本身的数据和元数据以及重叠表的聚合元数据；其次，LWC 将其按照重叠表的 DTable 键的范围划分为与之相对应的多个段并进行合并排序；最后，LWC 树将生成的新数据段和元数据添加到 L_{i+1} 层的 DTable 中。以图 4 为例，当 L_1 与 L_2 层的数据进行合并时，LWC 树只将 L_1 层第一个块的 DTable 读取到内存中，根据 L_2 层重叠键的范围对其进行段的划分，完成后再将其写回到 L_2 层直接与重叠的 DTable 进行合并，在这个过程中 LWC 树会舍弃无效键值对并对有效键值对进行合并排序。LSM 树中传统的压缩过程需要将参与合并的所有 SSTable 都读取到内存中，而 LWC 树只需读取 L_i 层的数据，从而大大减少了 I/O 写放大。

LWC 树有 4 个比较显著的特征：1)LWC 树采用一个轻量级的压缩机制，通过在重叠的表中添加数据并合并元数据来降低 I/O 的写放大；2)每个表都在上一层中保留重叠表中已合并的元数据来进一步减少压缩过程中的随机读取；3)LWC 树为 SSTable 提供了一个新的数据结构 DTable，以

提高查找性能;4)LWC 树在每一层中的多个 DTable 之间实现工作负载平衡,以确保读写效率和 LWC 树的平衡。因此,相较于传统的 LSM 树,LWC 树的写性能有较大的提升。

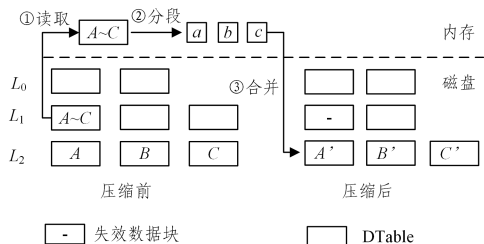


图 4　LWC 树"轻量级压缩"过程

Fig. 4　Light-weight compression process of LWC-tree

3.1.3　PebblesDB

PebblesDB[15]提出了一种新的数据结构——分段日志结构合并树(Fragmented Log-Structured Merge Trees,FLSM 树),同时也采取了具有垂直分组的分区分层设计。与其他设计的不同之处在于,它在 LSM 树的基础上使用与 SkipList 中的节点较为相似的标记点"Guard"来确定 SSTable 组间的范围。这个标记点可以根据插入的键进行概率性的选择以实现负载均衡,一旦被选择,它会在下一次压缩合并的过程中延迟应用,进一步提升对 SSTable 的并行查找效率,以提升范围查询的性能。PebblesDB 可以同时实现较低的写放大和较高的读写吞吐量,它不是重写一个 SSTable,而是将一个新的 SSTable 段写入下一层。这样就可以保证在压缩操作时,对大多数层的数据只需要写入一次,对于后面较高层的数据则采用另一种压缩算法。

在 FLSM 树的压缩策略中,每一个给定标记点所管理的 SSTable 都是排好序再进行分割的,这样上一层第 i 个标记点 G_i 所管理的 SSTable 组接收新的 SSTable 时就会在下一层插入到 G_i 所管理的 SSTable 组中,而在 LSM 树中只需要检查每层中的一个 SSTable。为解决过量读操作的问题,PebblesDB 在每个 SSTable 上引入了一个布隆过滤器[16]来检测是否存在给定的键,避免了读取不必要的 SSTable,并大大降低了数据结构导致的读取性能开销。在执行 Get 操作时,PebblesDB 通过二分查找定位某个 Guard 和对其中的 SSTable 进行搜索,并将迭代器放在其中合适的位置来进行范围查询;PebblesDB 插入的键包含这些键的版本信息,便于系统识别和放弃以前的版本以及进行读取和范围查询的操作。

PebblesDB 采用的是多线程并行搜索的搜索策略,每个线程读取一个 SSTable 时采用的是二分查找的方式。与 LSM 树的查找延迟相比,FLSM 树只会产生很小的开销,但是整个系统并没有将压缩过程中对 SSTable 分区时产生的 I/O 量考虑在内;另一方面,FLSM 树也可以在压缩时为每一层分配 Guard 来使分区最小化,但是这也会产生每一层 SSTable 分区不平衡的问题。同时,PebblesDB 也存在其他弊端:首先,如果工作负载完全适合存放于内存中,PebblesDB 会比 LSM 树具有更高的读取和范围查询的延迟。在执行查询操作时,定位一个正确的 Guard 和 SSTable 会产生更高的延迟,随着数据量的增加,大多数的数据集将不适合存放在内存中。其次,当需要将含有顺序键的一组数据插入到存储系统中时,PebblesDB 会产生更多的 I/O 开销,如果涉及大量的小范围查询操作时,PebblesDB 会比 LSM 树多产生大约 30%的开销,但随着查询范围的扩大,开销会不断下降,如果范围查询中穿插着插入或更新操作,开销就会完全消失。

3.1.4　dCompaction

dCompaction[17]引入了虚拟静态和虚拟合并的概念以降低合并的频率。在虚拟合并的过程中会生成一个虚拟的 SSTable,这样的 SSTable 只会进行简单的输入,而不对其进行实际的压缩操作。但是虚拟的 SSTable 会指向具有重叠范围的多个 SSTable,因此查询性能会降低。为了解决这个问题,dCompaction 根据触发真实压缩操作的实体 SSTable 的数目设置了一个阈值。如果在一次查询过程中一个虚拟的 SSTable 指向过多的 SSTable,此时也会触发压缩操作。dCompaction 是通过 SSTable 的数目来触发合并操作,因此它也可以被看作是分层合并策略的一个变体。总体来说,整个系统通过降低压缩频率来提升写性能,但这也会导致其在读性能方面有所下降。

3.1.5　SifrDB

SifrDB[18]是一种基于多层森林(Multi-Stage forest,MS-forest)的存储模型,其在继承了多层森林的优势的基础上通过实施分割存储机制弥补了 MS-forest 读取效率相对较低的缺点。在 SifrDB 的顶层采取类似于 stepped-merge 和 Size-Tered 的方式进行压缩来利用 MS-forest 的随机写入性能,底层采用分割存储来检测键的重叠范围以实现顺序写优化。此外,SifrDB 采用"早期清理技术"来尽早回收合并操作所保留的操作空间,避免数据存储失效。SifrDB 在所有的工作负载模式中能够实现最低的写放大以及较高的空间利用率,同时设计了一种并行搜索算法,充分利用 SSD 访问的并行性来提升读取性能,结果表明 SifrDB 在大型数据存储中具有极大的竞争力。

3.1.6　Skip-tree

Skip-tree[19]采取了一种合并跳跃的思想,通过减少从内存到磁盘上经过的 SSTable 的数目来减少读写 I/O 和写放大,并使用布隆过滤器来减少由版本约束引起的读取 I/O,在缓冲区缓存的基础上实现了基于日志结构的可靠性机制来防止数据丢失。与那些可以平衡读写性能、主要用于嵌入式存储的系统不同,Skip-tree 主要被用于密集型写入操作数据中心的后端存储。整个系统通过降低压缩频率来提升写性能,因此在读性能方面会有一定程度的下降。

3.1.7　TRIAD

TRIAD[20]提出了 3 种优化方案:TRIAD-MEM,TRIAD-DISK 和 TRIAD-LOG。TRIAD-MEM 主要缓解了更新频率较快的工作负载中的写放大问题,其基本思想是将内存组件中的冷热数据分离,将经常需要更新的数据保存在内存中,将冷数据写入磁盘。当热数据需要更新时,可以直接在内存中执行,减少了磁盘中大量的压缩操作。TRIAD-DISK 通过为磁盘中第一层设定阈值延迟该层的压缩操作来缓解写放大问题,在设置阈值时采用的是 HLL(HyperLogLog)概率估算器。当磁盘中 L_0 层的数据达到阈值时,TRIAD-DISK 可以删除该层多余的文件来推迟压缩操作的执行。TRIAD-LOG

将日志事务本身作为磁盘组件，并在上面构造一个索引结构，以此来提升查询性能，但是由于日志中的文件是无序的，因此范围查询会受到影响。

3.1.8 小结

综上，WB 树、LWC 树、PebblesDB 和 dCompaction 策略都是具有相似的垂直分组的分区分层结构，属于 tiering 合并策略，可以对其进行写方面的优化，它们的主要区别在于在执行 SSTable 的工作负载方面有所不同。WB 树依赖于哈希函数，但它牺牲了支持范围查询的能力；LWC 树采取“轻量级压缩”的方式，动态缩小了密集型 SSTable 键的范围；PebblesDB 在 SkipList 的基础上设置了具有概率性选择的标记点；dCompaction 通过设置阈值的方式对虚实 SSTable 进行了划分，有效地促进了工作负载的平衡，但直到现在还不清楚不平衡 SSTable 组会对各自的结构会产生何种影响，因此需要对其进行进一步的研究；SifrDB 利用 I/O 并行性，通过并行检查多个 SSTable 来提升查询性能，同时它采用的是水平分组的 tiering 合并策略和基于森林模型的压缩策略，因此 SifrDB 会有较低的写放大；Skip-tree 参考了 skiplist 的方式，采取合并跳跃的思想来减少写放大，加强了预写式日志的执行，确保存储系统在可变缓冲区中数据的持久性，但实际上，其在实现管理可变缓冲区方面非常复杂；TRIAD 通过将冷热数据分离的方式缓解了更新频率较快的工作负载中写放大的问题，但其中将日志文件作为磁盘组件的优化方案会降低范围查询性能。

3.2 基于合并的优化策略

压缩合并(compaction/merge)过程是 LevelDB 最为复杂的过程之一，同样也是 LevelDB 性能的瓶颈之一，其本质是内部数据重新整合的过程，是一种平衡读写效率的有效手段。LevelDB 设计 compaction 的目的之一也是提升读取的效率。下面列举了几个关于合并优化的代表性的例子。

3.2.1 bLSM-tree

尽管与传统的 B+树相比，LSM 树可以提供更高的写吞吐量，但它经常会有写暂停和不可预测的写延迟。bLSM 树[21]提出了一种“spring-and-gear”合并调度程序将写延迟最小化，它的基本思想是在每一层上设置一个额外的组件来使合并操作在不同的层之间并行执行。在合并操作的过程中，该策略控制 L 层的合并进度，以确保在 L_{i+1} 层的合并操作完成后才在 L_{i+1} 层产生这个新的组件。这种策略限制了内存组件中最大的写入速度并在一定程度上消除了写延迟，但是 bLSM 树本身是为未分区分层合并策略设计的，且该策略仅仅只限制了内存组件中的最大写延迟，而忽略了排队延迟。

3.2.2 LSbM-tree

“日志结构缓冲压缩树”(Log-Structured buffered-Merge tree，LSbM 树)[22]能最好地利用缓冲区缓存来进行快速访问，其基本思想是通过添加一个压缩缓冲区，使由压缩引起的缓冲区中无效缓存数目达到最小。

在 LSbM 树中，读取请求被发送到压缩缓冲区的 B_i 层来获取频繁访问的数据。当一个数据查询请求到达时，先在缓冲区缓存中查询数据，如果其中有该数据，则被标记为频繁访问的数据，否则在缓冲区缓存中记录该数据；查询数据时从 LSbM 树的第一层开始查询，频繁数据会直接被索引到压缩缓冲区中，其他数据请求会被发送到 C_i 层，若在第 i 层未查询到该数据，则转向下层压缩缓冲区继续查询。以图 5 为例，系统对 a，b 和 c 发出访问请求，其中 a 和 b 是经常被访问的数据，且在缓冲区缓存中有所记录，它们被分别存放于 B_1 和 B_2 中，因此它们会被直接从压缩缓冲区中读取，而不经常访问的数据项 c 被存放在 LSM 树的底层，因此读取请求需要到底层的 LSM 树中读取数据项 c。

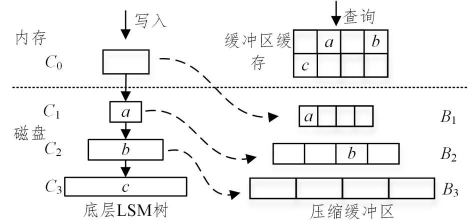


图 5 三层 LSbM 树结构

Fig. 5 Structure of three-layer LSbM-tree

由于压缩缓冲区主要维护经常被访问且在压缩操作的过程中不经常更新的数据，因此压缩缓冲区需要选择性地只保留其中经常访问的数据，并保持它们的稳定性。图 6 给出了缓冲区合并的工作原理。当 C_i 满时，它的所有数据将被合并到 C_{i+1} 中进行合并排序。同时，它也会作为最新的数据被添加到 B_{i+1} 层。需要注意的是，B_{i+1}^0 是与 C_i 中的文件一起产生的，而这些文件是已经在磁盘上存在的，因此在添加的过程中不会产生额外的 I/O 操作，且 B_{i+1}^0 的数据不会在压缩过程中进行更新操作。此外，B_{i+1}^0 中包含 B_i 层的所有数据，且这些数据都是有序的，因此与 B_i 相比，它更适合帮助用户搜索数据。此时，B_i 中的排序表会被从压缩缓冲区中删除，最后也会被从磁盘中移除。当有大批量的重复数据写入 $i+1$ 层时，C_{i+1} 层中重复且过时的数据会随着压缩操作被移除。然而，压缩缓冲区 B_i 层中的旧数据不会被移除，因为数据一旦被添加到压缩缓冲区中就不会被压缩。由于这些废弃的数据会占用额外的磁盘和内存空间，因此压缩缓冲区列表并不适用于有重复数据插入的层。在 LSbM 树中，通过比较 C_{i+1} 层中数据本身的大小和压缩到 C_{i+1} 层的数据大小可以检测出 $i+1$ 层有无重复数据。如果 C_{i+1} 的大小小于压缩到它其中的数据量大小，那一定存在重复的数据。当检测到重复数据时，LSbM 树会将 B_{i+1} 层冻结。当 C_{i+1} 层写满并合并到下一层时，B_{i+1} 会被解冻，并继续作为 C_{i+1} 的压缩缓冲区列表。

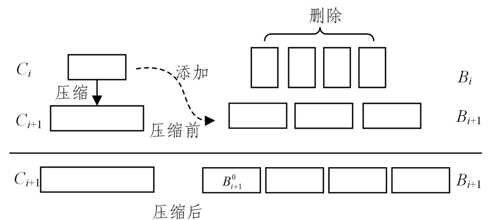


图 6 缓冲区合并过程

Fig. 6 Process of buffer merging

实验结果表明，LSbM 树可以实现最佳的性能，与上文的 bLSM 树相比，缓冲区失效的问题得到了进一步解决，且吞吐量得到了大大提升。底层 LSM 树上的排序结构可以有效地支持磁盘的范围查询；同时，压缩缓冲区可以通过有效地在

缓冲区缓存中保持频繁访问的数据来提供快速的数据访问。

3.2.3　VT-tree

VT-tree[23]结合了 B+树、文件系统和 LSM 树的性能优势,在此基础上设计了拼接(stitching)技术和单个事务的高效事务型架构,stitching 指在合并多个 SSTable 时,如果参与操作的 SSTable 中的某个键的范围与其他任何一页的范围不重叠,就可以直接将这一键范围直接指向压缩后新生成的 SSTable,通过减少不必要的数据复制限制了在一次查询操作中的查询次数,提升了 LSM 树的性能,进一步提升了顺序访问和随机访问工作负载的性能。为解决 stitching 技术产生的碎片化问题,VT-tree 通过设置阈值和采用商型过滤器[24]的方式来减少负面影响,但是仍引入了额外的随机 I/O。

3.2.4　Tebis

Tebis[25]是一种高效的基于 LSM 的键值存储,它通过减少 I/O 放大和 CPU 的运行时间来维护副本索引,并提出了一种 primary-backup 的复制方案,它只在主节点上执行压缩,将预先构建的索引发送到备份节点,从而避免了备份节点中的所有压缩。

Tebis 使用 Kreon 来管理数据,同时通过 RDMA 实现了一个 primary-backup 协议。当它接收到来自客户端的更新和插入时,primary 将分 3 步将每个操作复制到 backup 上:首先,它将键值对插入 Kreon 中,并返回日志尾段中键值对的偏移量;其次,它将键值对加入相应的偏移量处的每个 backup 的 RDAM 缓冲区中。Tebis 等待确认所有 RDMA 写操作都已被复制到 backup 的内存中。Tebis 采用一个可靠队列来确保 RDMA 的写操作将所有数据都写入到 backup 区,当客户端收到确认信号时,表示所有操作已完成。当主日志中的尾段变满时,primary 会将其进行持久化存储,并向每个 backup 发送信号以持久化各自的 RDMA 缓冲区,backup 区向 primary 区发送确认信号表示完成持久化操作。

Tebis 提出的 Send Index 方法可以在 backup 区有效地保持最新的索引。Tebis 中的 primary 区在 L_i 与 L_{i+1} 的每次压缩后,会将其预构建的 L'_{i+1} 索引发送给所有 backup 区,而不是在 backup 区上执行压缩。因此,backup 区的 I/O 操作会大大减少。此外,它们用一个轻量级的索引重写操作替换了内存中的排序操作,在这一过程中产生的 CPU 开销更少,可以有效地提升系统的整体性能。

3.2.5　LDC

为降低延迟和提升吞吐量,LDC[26]提出了一种新的下层驱动压缩方法,打破了传统的上层驱动压缩方式的局限性,触发了“自下而上”的压缩操作,并通过减小压缩粒度来实现更小的延迟,通过减少写放大来实现更高的吞吐量。此外,文中通过添加一个自适应策略来调整键压缩的阈值以适应工作负载特征的变换,并将 LDC 扩展为自适应的 LDC(ALDC),可以显著地降低尾延迟,同时实现更高更稳定的吞吐量。

LSM-tree 传统的压缩方式会导致两个问题:1)在压缩时由于有额外的数据不断地从磁盘读取到内存中,而这个过程会产生额外的 I/O 操作,因此降低了 LSM 树的吞吐量;2)在压缩粒度较大的情况下,需要较长的时间来完成压缩操作,这一期间伴随着较大的尾延迟等负面影响。因此,为降低写放大,系统应积攒更多的高一层的数据以进行压缩。LDC 算法将传统的 LSM 树的压缩操作分为两步:1)当上层的 SSTable 要被合并到下层时,LDC 不会执行实际的 I/O 操作来进行压缩,而是采用一种轻量级的链表,根据键的重叠范围将上层的 SSTable 分组和下层的 SSTable 分组连接起来,并将上层的 SSTable 变为 Frozen SSTable,表示不会再连接到其他 LSM 树;2)只有当底层 SSTable 积攒了足够多重叠的键范围的分片时,才会产生实际的操作将低层的数据合并到更低一层。

自适应的 LDC(ALDC)是在 LDC 算法的基础上进行的一种扩展算法,该算法通过自适应地调整参数来适应读写比例的波动。ALDC 设置了一个阈值 T_{sr},该数值表示切片的总大小与较低层的 SSTable 的大小之比。ALDC 考虑多方面的影响(包括读/写操作的实时比率、LSM 树的放大率)动态地调整阈值 T_{sr} 的大小,以在 LDC 的基础上实现提升整体性能的目的。

3.2.6　小结

LSbM 树和 bLSM 树在缓冲区缓存丢失等方面优化了合并操作的实现,但它们在某些方面仍然会有明显的限制。LSbM 树中旧组件的延迟删除操作会对冷数据的查询产生消极影响,同时由于异地更新,写延迟也是 LSM 树中存在的一个问题;bLSM 树在解决这类问题时也仅仅是限制了写入内存组件的最大延迟,但由于队列问题,其端到端的写延迟仍会表现出很大的差异。在缓解缓存失效问题方面,Leaper[27]采取了机器学习的方法来预测 LSM 树中的热数据,并将它预存到缓存中,且不会受到导致缓存失效问题的后台 LSM 树操作的干扰。在解决写延迟的问题上,Faster 高性能键值存储系统[28]结合冷热数据的思想,将从磁盘中读取到内存中的数据存放在混合日志中直接进行就地更新,减少了异地更新带来的写延迟;VT-tree 为加速合并操作引入了 stitching 技术,但这会导致碎片化,且与在 LSM 树中广泛使用的布隆过滤器不兼容,因此有一定的局限性;Tebis 中的 primary-backup 协议可以有效减少两个压缩区中额外的压缩操作,并通过 Send Index 方法解决了更新索引的问题;LDC 和 ALDC 算法打破了传统压缩方式的束缚,减小了压缩粒度,将原始大规模的压缩操作调整为“局部的压缩操作”,实现了较高、较为稳定的吞吐量。上述压缩策略都是通过清除冗余数据的方式来减少磁盘的 I/O 操作,用户可以根据不同的应用场景采用不同的策略来提升查询性能。

3.3　基于硬件的优化策略

3.3.1　WiscKey

由于 SSD 支持高效的随机读取,同时键一般比值所占的空间小,因此可以采取将值与键分离的方式来提升 LSM 树的写性能。在 WiscKey[29]中,LSM 树中存放的是键和这个键对应的值所在的位置,实际的 value 则存放在友好存储的 SSD 中。对于一个给定大小的数据库,Wisckey 中 LSM 树的大小要比 LevelDB 中 LSM 树的大小小得多,而较小的 LSM 树可以在传统的工作负载下显著减少它的写放大,同时也可以存放较大的数据量。除了写性能提升这一方面,写放大的减少也减少了对 SSD 的擦除操作,因此也提升了 SSD 的寿命。

WiscKey 较小的写放大也提升了整个系统的查询性能。在查询操作期间，WiscKey 首先查找 LSM 树中的键和对应的值的所在位置；一旦找到，就会有另一个读操作去检索这个值，这就导致 WiscKey 比 LevelDB 的查询速度更慢。然而，在同样数据库大小的条件下，WiscKey 中 LSM 树的大小要比 LevelDB 中的 LSM 树小得多，因此前者检索的 SSTable 层数要少很多。此外，每一次的查询操作只需要一次简单的随机读操作，因此 WiscKey 相比 LevelDB 查询性能更好。

WiscKey 把键值对存储在日志结构中，用 LSM 树作为索引将每个键映射到日志中的位置。虽然其可以通过只合并键的方式降低写入成本，但由于值的排列不是顺序的，因此范围查询的性能将会受到显著影响。同时，WiscKey 采用垃圾回收的方式来解决存储空间占用问题：首先，WiscKey 扫描日志的尾部并通过对 LSM 树执行点查找来检查每个数据条目，判断每个键的位置是否已经更改；其次，位置未被更改的有效数据条目被附加到日志中，并在 LSM 树中更新它的位置；最后，日志的尾部会被消除来回收存储空间。但由于随机点查找会产生较大的开销，因此垃圾回收的过程也会影响整个系统的性能。

3.3.2 HashKV

HashKV[30] 是一种持久的专门针对更新密集型工作负载的键值存储系统，它加强了对 LSM 树顶层数据键值分离的管理性能，实现了较高的更新性能，并在一定程度上解决了 WiscKey 中垃圾回收的开销问题。其基本思想是根据键将值划分为多个分区，并独立地收集垃圾分区。它支持键值对的如下操作：PUT（写入键值对）、GET（检索键对应的值）、DELETE（删除键值对）和 SCAN（检索一批键的值）。

HashKV 建立在键值分离的基础上，采用了一种新的基于哈希的数据分组方式来对值进行存储，将值的存储划分为固定大小的分区，并确定地将值映射到存储空间。但是这种确定性的性质会限制键值对的存储位置，因此文中提出了动态保留空间分配、热度感应（关于冷热数据的区分原则在文献[31]中有详细说明）和选择性的键值分离这 3 种新的策略来减少基于哈希的数据分组的限制，图 7 给出了 HashKV 的主要结构。由于确定性映射，基于哈希的数据分组支持轻量级更新，HashKV 对每个键进行了一系列操作以查找每个键的最新值，将有效的键值对添加到日志中，显著地降低了垃圾回收的开销。

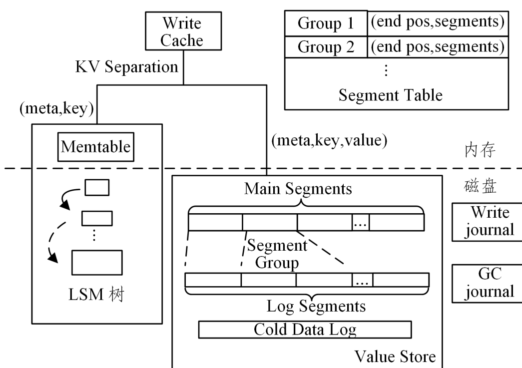


图 7 HashKV 的主要结构

Fig. 7 Main structure of HashKV

3.3.3 LOCS

基于多通道并发访问的优化策略（LSM-Tree based key-value Store on Open-Channel SSD，LOCS）[32] 是基于开放通道 SSD 上 LSM 树的一种实现方式，它采取了多个通道接口，允许应用程序灵活地安排磁盘的写入过程，以提高内部的 I/O 并行性。为充分利用磁盘读取和数据存放一一对应的特性，LOCS 采用最小加权的队列合并策略以平衡分配各个通道的工作量。

LOCS 使用包含有 44 条通道的开放通道 SSD-SDF 来对 LevelDB 进行扩展，以启用对这些通道的并发访问。相较于传统的 SSD，SDF 向用户应用程序公开 SSD 内部并行性，并将每个通道作为一个独立的设备呈现给应用程序，使用户应用程序可以有效组织和调度数据，并向用户提供不对称的 I/O 接口，减少了写放大，并保持了 SSD 原有的随机读取性能，同时增加了擦除操作，赋予了软件调度擦除操作以最小化延迟的能力，减轻与高优先级请求的冲突。但是常规的 SSD 控制器调度的擦除操作隐藏在应用程序中，会造成服务的不可预见性，增加了工作负载。SDF 中还有一个专门简化输入输出的栈，在 Linux 系统中会为传统的低速磁盘设计一个相对复杂的 I/O 堆栈，但是 I/O 栈的分层设计是高性能 SSD 的瓶颈。SDF 支持绕过内核的大多数 I/O 层，使系统可以直接使用 ioctl 接口与 SDF 驱动程序进行通信。图 8 给出了 LOCS 的基本架构。

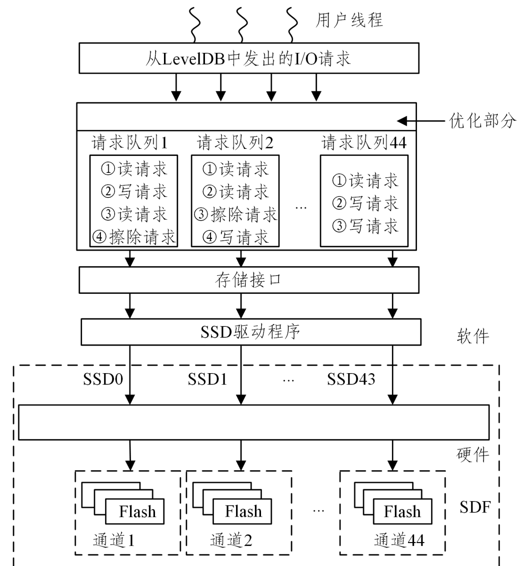


图 8 LOCS 的基本架构

Fig. 8 Basic architecture of LOCS

相较于 LevelDB，LevelDB 启用了多个通道进行并发访问，通过增加内存中 immutable memtable 的数量以充分利用 SDF 的 44 条通道传入更多的数据，可以产生多个来自 immutable memtable 的写入请求；提出了写流量控制策略，其中心思想是当 LevelDB 第 0 层的 SSTable 的数量到达阈值时，LevelDB 会限制用户发出的写请求数量，从而减少对 level 0 重叠 SSTable 的检索，将写性能转换成读性能；LOCS 为提高系统访问效率，提出了循环调度策略和基于队列长度的请求调度策略，平衡了读、写和擦除 3 种请求，缓解了循环调度队列长度不平衡的问题。每一个请求队列的队列长度可以表示为：

$$Length_{\text{weight}} = \sum_{1}^{N} W_i * Size_i \tag{1}$$

其中，N 为请求队列的总个数，W_i 和 $Size_i$ 分别表示每个请求的权重和大小。每种请求类型的权重根据它们相应的延迟确定。写入和擦除请求的大小都是基本确定的，只有读取请求的大小可能会有所不同。与循环调度不同的是，LWQL 策略把 3 种 I/O 请求都考虑在内，从而保证队列之间是相互平衡的。

3.3.4　小结

由此部分内容可以看出，这些改进主要是通过对 LSM 树的硬件设备进行优化来提升整个系统的性能，例如 Wisc-Key 和 HashKV 采用的是键与值分离的策略，它们通过只合并键的方式可以显著提升 LSM 树的写性能，但同时也会导致查询性能和空间利用率的降低，因此需要采取垃圾回收策略来回收磁盘空间。另一方面，LOCS 通过实行多通道并发访问在提高整个系统并发性的同时优化了 I/O 请求的调度策略，提高了数据访问效率。此外，在后面的一些研究中也会通过存储设备——类似于 SSD 和 HDD 等，来提升整个键值系统的性能。

3.4　基于混合存储系统的体系结构

LSM 树作为键值存储系统主流的存储引擎之一，为系统提供了良好的写入性能。但是 LSM 树在合并数据时产生了较高的写放大，严重影响了存储系统写入性能的进一步提升。混合存储能够充分利用不同类型存储介质的特性组成高效的存储系统，既能支持存储系统容量的大幅扩展，又能在保证系统低成本的前提下提升存储系统的性能。当前存在的存储介质包括磁盘(HDD)、固态盘(SSD)、非易失性存储器(NVM)等，研究者可以利用这些存储介质不同的性质，根据数据访问特点和系统负载情况将数据请求交给最适合处理这些请求的设备，进而提升整个系统的性价比、使用寿命、可靠性、容量等指标。

3.4.1　NVMe SSD

现如今很多常见的应用和服务系统都是基于键值存储系统设计的，它们执行快速内存处理，但是通常会受到 I/O 性能的限制。SpanDB[33] 适应了当今比较流行的 Rocks DB，以此来利用高速 SSD 的选择性部署。SpanDB 允许用户将其大部分数据存储在更便宜和存储量更大的 SSD 上，同时将预写式日志(WAL)和 LSM 树的顶层迁移到更小和更快的非易失性存储器(Non-Volatile Memory express SSD，NVMe SSD)上，主要结构如图 9 所示。为了更好地使用这种快速磁盘，SpanDB 通过一套基于 NVMe SSD 的开发套件——SPDK 提供高速并行的 WAL 写入操作，并支持异步请求处理以降低线程之间的同步开销，通过基于轮询的 I/O 操作来大幅提高工作效率。

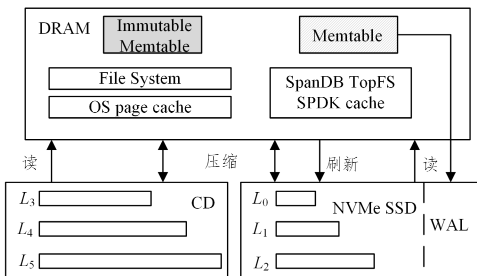


图 9　SpanDB 基本架构

Fig. 9　Basic structure of SpanDB

磁盘中的数据主要分布在 CD(Capacity Disk)和 SD(Speed Disk)这两个物理存储分区当中。SD 部分被进一步划分为两个区域：一部分是较小的 WAL 区域，另一部分是数据管理区域。SpanDB 将 SD 作为一个原始设备进行管理，并重新设计了 Rocks DB 中 WAL 使其有更快的写入操作。数据管理区域存放了 LSM 树中顶层的热数据，其接收所有的刷新操作，这些操作将整个 memtable 中的数据写入到 SSTable 的 L_0 层。SpanDB 使用了一个轻量级的文件系统 TopFS，来为 SPDK 顶层 I/O 提供接口以实现简单和动态重定位的功能。CD 部分包含较多的冷数据，这部分数据主要通过文件系统访问，由操作系统的页缓存进行辅助访问。此外，SD 和 CD 两部分都适用于用户的读取和压缩操作，SpanDB 对这两部分进行了额外的优化，实现了两部分的同时压缩操作，并自动地协调前台和后台的任务。由此，SpanDB 能够对两部分实现动态的规划管理。

SpanDB 与传统的存储结构最大的不同在于它采用异步处理模型来减少同步开销，并自适应地调度任务。SpanDB 采用了通过 SPDK 进行快速访问的 SD 来提升 WAL 的写入性能和高并行性；使用 SD 进行数据存储，优化了在快速 SSD 中的带宽利用率；采用自主分配的工作负载 SD-CD，对冷热数据进行区分，跨设备地聚合了 I/O 的可用资源；通过将部分 I/O 操作引入 SD 中，减少了密集型读操作工作负载的尾部延迟；提升了系统的写性能，调整了前台/后台的 I/O 需求，利用快速轮询算法节省了 CPU 的资源。另一方面，该系统也存在局限性：由于 SPDK 的访问约束，SD 需要绑定一个进程，导致系统难以共享资源；对于读操作较多的工作负载来说，SpanDB 的速度并没有显著提高，反而还会在异步处理中引入性能的开销。

与 SpanDB 类似的还有一种基于 SSD-SMR[34] 的混合存储架构，这个结构是一种逐层增加文件大小的分布方案，它对 SSTable 设置了不同的大小，并设置了一个最大存储值和小文件阈值，小于阈值的被放入 SSD 中存储，大于阈值的被放入 SMR 磁盘中存储。这种情况下，前面几层的小文件 I/O 访问密集，可以充分利用 SSD 的高性能；后面的大文件写入 SMR 磁盘时会控制写放大率的大小，提高 SMR 的访问速度。它与 SpanDB 都是将数据分类存储，但二者的分类标准不同：SpanDB 是根据数据的冷热程度进行划分，而 SSD-SWR 混合存储架构是根据文件大小区分。相对于普通的机械硬盘，SSD-SMR 的性能都有所提升。

3.4.2　DRAM 与 NVM

DuetKV[35] 是一种基于 DRAM-NVM 内存的混合索引的持久化键值存储系统，主要工作流程如图 10 所示。

DuetKV 的整个系统采用 Hash Table 和 B+ Tree 两种数据结构实现了混合内存索引，前者实现在 DRAM 中，避免 Hash Table 索引项持久化以及 Rehash 导致 NVM 寿命短，而后者使用了 DRAM 与 NVM 两种介质，并创建了叶子节点、内部节点和叶子缓存节点，其部分数据结构借鉴了 pmemKV[36]。叶子节点主要用于索引所有的键值对数据，其数据结构借鉴了 FP-tree[37] 和 wB+-tree[38] 的思想，在占用 NVM 的内存时提供一致性和持久化保证。当叶子节点中的

数组已满且需要插入新的索引项时会通过分裂产生新的叶子节点，通过修改一次 next 指针将产生的新叶子节点添加到链表最后，同时将 next 指针指向下一个叶子节点以完成对所有叶子节点的遍历。DRAM 中的内部节点是通过叶子节点创建的，不需要保证持久化，它主要由两个数组组成，一个数组存放有序的键，另一个数组存放孩子节点的指针。当内部节点保存的键值对数目大于有序的键时，则会触发内部节点的分裂和递归更新操作。叶子缓存节点和叶子节点同时创建，占用 DRAM 的内存，其中包含两个数组和两个指针，其好处在于对键的查找不需要读取 NVM，从而避免了 NVM 较高的读延迟对 Scan 的影响，也减少了对 NVM 的损耗。在减少 NVM 损耗方面，DuetKV 还设计实现了 FIFO-Lock 和 Write-Batch 两种命令缓存队列，并利用 NVM 非易失性的特点实现了对原始键值存储系统的恢复。

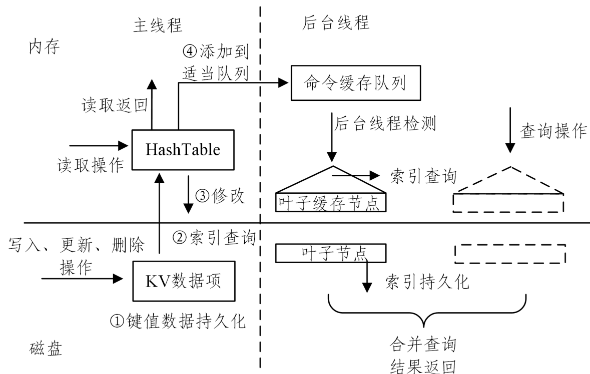


图 10 DuetKV 主要工作流程

Fig. 10 Main workflow of DuetKV

新型非易失性存储器具有扩展性好、低功耗、非易失性等特点，基于 NVM 的内存系统有望在未来补充甚至代替 DRAM 内存。但是与 DRAM 相比，NVM 写延迟较长、写入寿命有限。现有的工作通过构建混合索引可以支持高效的键值操作，同时在 NVM 上索引的读写暴露在键值对操作的关键路径上，导致键值对操作的延迟较高，影响了系统的寿命。因此将二者结合起来可以有效提升整体性能。

3.4.3 NVM 与 SSD

Kannan 等和 Kaiyrakhmet 等分别提出了 NoveLSM[39] 和 SLMDB[40]，这两者都是基于 SCM 和 SSD 的混合键值存储系统，前者减少了非易失性存储器中 memtable 存储到达阈值时系统产生的请求处理中断，但当数据量增加时，底层的压缩进程仍会导致较长时间的暂停；而 SLMDB 构建了一个基于 SCM 的单层键值存储，它将原始的多层 LSM 树结构改为一层，并在 SCM 中添加了全局的 B+树索引以减少系统的读取放大。而 SSHKV[41] 同样结合了 SCM 和 SSD 特点，构建了 DRAM-SCM-SSD 混合存储架构，使用 SCM 的非易失性、高性能和字节寻址的特性，在 SCM 中构建一个散列表，并把键和元数据一起存储到 SCM 中。与前面两种 SCM 及 SSD 混合存储架构相比，SSHKV 重新设计了数据组织策略以减少 LSM 树的读/写放大开销，同时应用了“逻辑空间放大策略”以减少 SSD 中的有效数据迁移；它充分利用了 SSD 和 SCM 两种存储介质，相较于 LevelDB，其随机写性能提升了 6.8 倍。

3.4.4 DRAM，NVM 和 SSD

MatrixKV[42] 采用了多层 DRAM-NVM-SSD 结构的键值存储系统，通过在 NVM 中建立 Matrix Container 来管理 LSM 树的 L_0 层，并在 L_0 层和 L_1 层之间使用列压缩。当外部数据传入 MatrixKV 时，首先会写入到 DRAM 中的 Memtable 中，随后将 Memtable 写入到具有较高性能的 NVM 中的 L_0 层，由其中的 Matrix Container 进行存储和管理，随后 L_0 层中的数据会通过列压缩写入到 SSD 中的 L_1 层，最后在 SSD 中扁平化存储 LSM 树的其他层数据。Matrix Container 包括 Receiver 和 Container 两部分，前者用于接收并存储 Memtable 中的数据，将它们序列化为单个行，并组织为 RowTable。当其中存储的数据量到达上限时，Receiver 停止接收并动态地变成 Compactor，同时将创建一个新的 Receiver 来接收刷新的 Memtable。这一过程与内存中 Memtable 到 Immutable Memtable 传输数据的过程相似。此外，MatrixKV 还通过减小从 L_0 到 L_1 层的压缩粒度来解决写停顿的问题。然而，当 Memtable 从内存写入到 NVM 中的 L_0 层时会重构为 RowTable，并为其加入相应的元数据，同时还需要处理指针来指向前一个 RowTable 相应键的位置，这一过程会影响 flush 的速度。

3.4.5 小结

各种混合存储系统将不同存储介质相互结合以达到最优的性能，它们都能充分利用硬件提供的特性以实现 LSM 树更好的性能。其中 NVM 因具有高效的性能引起了较多关注，并被广泛应用于各种混合存储介质的键值存储系统中，在处理冷热数据方面扮演着重要的角色。表 1 从容量、成本、寿命、随机性能、顺序性能这 5 个方面对 HDD，SSD 和 NVM 进行了比较。

表 1 3 类存储介质不同属性之间的对比

Table 1 Comparison of different properties of three types of storage media

	容量	成本	寿命	随机性能	顺序性能
HDD	大	高	长	一般	好
SSD	较大	较高	较长	较好	较好
NVM	小	低	短	很好	很好

不同的存储介质具有不同的特性，使得它们有着各自的优势和不足，这就导致单一存储介质的键值系统不能充分发挥存储介质的优势。采用不同存储介质结合的键值系统可以最大化存储介质的优势，因此具有较高的研究价值。

3.5 其他特殊的工作负载

在实际的应用场景中，许多键值系统的优化方案对工作负载的要求较高。本节主要以 LSM-trie，SlimDB 和 Lethe 为例来分析如何实现更好的性能。

3.5.1 LSM-trie

LSM-trie[43] 是基于 LSM 树使用键的哈希值对数据进行索引，用于管理大量占用空间较小的键值对。它提出了一系列优化方式以减少元数据的开销，其设计目标是能够在容量和 KV 数据项的数量方面实现尽可能大的键值存储。LSM-trie 同样采用了分区分层设计来减少写放大：在每一层中，存储的数据被键的哈希值分割到不同的区间。然而，在一个

区间内允许存在多个“树”，这里的“树”不是真正意义上的 B 树，而是以哈希进行索引的一批数据。合并操作将一个区间里的多个“树”进行进一步哈希，以放入下一个阶里对应的区间，且合并过程不会影响下一个阶里原有的数据。因此，LSM-trie 是一种基于森林模型[44]的结构。

LSM-trie 提出了一种被称为 SSTable-trie 的数据结构，进一步消除了索引界面，它在键上应用了一个加密的哈希函数，用哈希键来确定一个 SSTable 属于哪一层。SSTable-trie 的每个节点都包含着一定数量的 SSTable，同时每个节点都拥有固定数量的子节点，子节点的数目和 AF(放大系数)相同。假设这个数为 8，那么这个节点的子节点可以被定义为 8 个三位二进制的节点(000，001，…，110，111)。这样就可以根据二进制位组找到节点路径，相同深度的节点构成 SSTable 中的一层，相当于 LSM 树中的一层。与 LevelDB 中的键值对存放方式不同的是，SSTable-trie 中每一层的 SSTable 之间都是相互独立的，一次压缩操作只在一个节点内部进行，且每次操作结束后节点中参与压缩的键会根据各自的哈希键移动到各自的子节点中。通过这种操作可以确保每次压缩的键的范围都是唯一的，且压缩产生的新的 SSTable 的范围也都不重叠，因此这种压缩方式会产生较小的写放大。

为实现较高的数据访问效率，LSM-trie 为每个节点设置了一个布隆过滤器。由于在每一层的每个节点中都存在多个 SSTable，LSM-trie 将同一个节点中 SSTable 的布隆过滤器聚集在一起，这样就可以通过点查找使用单一 I/O 来获取它们。

总的来说，LSM-trie 只适用于数据量比较大的键值对，这种规模可能大到元数据、索引页、布隆过滤器都无法进行缓存；同时，由于 LSM-trie 的优化在较大程度上依赖于哈希，因此它只支持点查询。

3.5.2 SlimDB

SlimDB[45]把键值存储中的主键划分为两个片段：前缀 x 和后缀 y。它支持给定的前缀和后缀的点查询以及对具有相同前缀的键值对进行检索，因此它可以支持半排序的数据。针对 SlimDB 中半排序的数据，SlimDB 中采用三层块索引来代替 LevelDB 中 SSTable 原始的块索引。

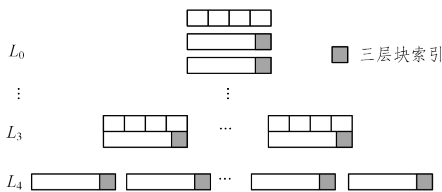


图 11 SlimDB 三层架构

Fig. 11 Three-layer structure of SlimDB

SlimDB 采用了混合存储架构，其中低层次的使用“tiering 结构”，较高层次的使用“leveling 结构”，同时进一步使用多层的“cuckoo filters[46]”来提高分层合并策略的点查询性能。这种多层次的存储能够利用混合键值存储的设计来平衡其读放大、写放大和内存使用三者之间的关系。图 11 给出了一个四层 SlimDB 架构，其中 L_0 至 L_2 都采用阶梯式合并算法的数据布局，其中包括多层 cuckoo filter 和三级块索引，且所有的过滤器和索引都存储在内存中；L_3 和 L_4 使用原始的 LSM 树的数据布局，并在内存中缓存三层块索引。对于存储着大量数据的 SSTable 来说，SlimDB 会将它们存放到磁盘中。每一层的 cuckoo filters 会将每个键映射到其所对应 SSTable 的 id，这些 SSTable 存储着这些键的最新版本，这样点查找时 cuckoo filters 只需要做一次检查。在索引数据时，SlimDB 首先将指定键的前缀映射到包含着这些前缀键的一系列页面中，这样给定一个键的前缀就可以检索到这个前缀对应的键值对；其次，它将这些键的后缀范围存放在这些页中，以便高效地支持前缀键和后缀键的点查询。实验证明，SlimDB 的这种混合存储架构和多层索引在一定程度上能够减少写放大和元数据开销。

3.5.3 Lethe

Lethe[47]优化了 sort key 大范围删除带来的问题。LSM 树在删除一条记录时是通过追加标记来实现的，但这个标记本身会占用空间。大部分场景下，键的大小远远小于值的大小，这就导致由删除操作带来的空间放大要比更新操作更严重；另一方面，追加的标记需要一直压缩到最底层之后才能被删除，这一过程会造成严重的写放大。因此 Lethe 提出了 Fast Delete(FADE)策略，用于保证逻辑上被删除的数据在一定时间内会永久地从磁盘上消失，缓解了由冗余数据带来的写放大问题；此外，它支持较为高效的非 sorted key 的范围删除。在 compaction 的过程中，FADE 的文件选择策略有 3 种模式，第一种是经典的 LSM 树文件选择策略；第二种是估算失效最多的文件并将其删除，这主要是针对空间放大进行的优化；第三种是选择 time-to-live(TTL)到期的文件进行压缩。Lethe 采用更加激进的压缩操作来删除记录，采取 FADE 策略可以较快地清理无用的空间，有利于缓解空间放大，同时可以减少后续压缩操作中无效数据参与压缩的可能，将待删除的数据的停留时间严格限定在某一时间内，总体上减少了写放大；且能够提升布隆过滤器的过滤效率，减少范围查询需要扫描的个数。

3.5.4 小结

本节所述的 3 个改进方案都是针对特殊的工作负载，但并不适用于通用的工作负载。LSM-trie 只支持点查询，在一定程度上会影响范围查询的性能，且不利于空间的优化；SlimDB 和 LSM-trie 同样放弃了一些查询性能，仅局限于通过给定前缀键的有限形式的范围查询；Lethe 是一种基于 LSM 树的对删除操作友好的优化方案，因此适用于删除数据较多的应用场景。因此研究者需要通过给定工作负载的类型来判断是否可以采取某些优化策略。

4 分析与对比

在上述章节中，本文对 LSM 树有可能进行改进的几个方面进行了详细的阐述。通过对 LSM 树性能的分析可以得出结论：LSM 树的各个优化方案之间并不是完全独立的，一种方案的思想在其他优化方案中也起着不可小觑的作用；而在提升 LSM 树性能的过程中，性能优化的各方面之间也可能会互相影响，对其中一方面改进的同时又或多或少地影响着其他几个方面，一方面性能提升的同时可能会带来另一方面

性能的提升或下降。表 2 从降低写放大、优化压缩策略、硬件、混合存储介质和特殊的工作负载几个方面对现有典型的 LSM 树优化策略进行了对比。

表 2 LSM 树不同优化策略的分类

Table 2 Classification of different optimization strategies for LSM trees

	写放大	压缩策略	硬件	混合存储介质	特殊的工作负载
WB-tree[13]	√				
LWC-tree[14]	√	○			
PebblesDB[15]	√				
dCompaction[17]	√				
SifrDB[18]	√				
Skip-tree[19]	√				
TRIAD[20]	√				
bLSM-tree[21]		√			
LSbM-tree[22]		√			
VT-tree[23]		√			
Tebis[25]		√			
LDC[26]		√			
WiscKey[29]	√		○		
HashKV[30]	√		○		
LOCS[32]			√		
SpanDB[33]				√	
DuetKV[35]				√	
NoveLSM[39]				√	
SLMDB[40]				√	
SSHKV[41]				√	
Matrix[42]				√	
LSM-trie[43]	√				○
SlimDB[45]	√				○
Lethe[47]	√				○

注：√为优化的主要方面；○为优化的次要方面。

从表 2 中可以看出，由于 LSM 树本身具有较高的写放大率，因此这些改进中大多数都提高了分层合并策略的写性能。WB 树用"溢出和分割"策略替代了原有的压缩操作，并通过快速分割技术来提升性能，虽然写性能有所提升，读性能却有所降低。LWC 树为减少写放大采取了在压缩过程中添加数据的方法，在不同的存储介质、不同 value 大小和不同的访问模式下性能均有提升。PebblesDB 在跳表和 LSM 树的基础上构造了 FLSM 树，能有效平衡读写性能，实现了较高的读写吞吐量和较低的写放大。dCompaction 提出了"虚拟压缩"的概念，通过延迟部分压缩进程和降低压缩频率来减少压缩过程中对数据的写入和读取操作。TRIAD 结合冷热数据分离的思想，将经常更新的数据保存在内存中，将不经常更新的数据写入到磁盘中，而由于日志文件是无序的，在 TRIAD-LOG 的实现过程中，范围查询会受到影响。LSM-trie 放弃了一些查询性能来换取写放大性能的提升，但因其不支持范围查询，所以可以用于某些对范围查询要求比较低的工作负载；SifrDB 继承了多层森林模型的优势，并提出"早期清理技术"，通过分割存储机制避免了模型本身的缺陷，因此写性能会有所提升。SlimDB 采用混合合并策略降低了写放大，并引入了多层 cuckoo filters 来提升较低层 SSTable 的点查询性能，但它在通过后缀键查找对应的 value 时仅支持限定的范围查询。Lethe 采用 FADE 策略缓解了写放大和空间放大的问题，但它对删除密集型的工作负载较为友好，其中的 KiWi 策略为提高范围删除的效率牺牲了一定的查询性能。bLSM-tree 提出了"spring and gear"合并策略来降低写延迟，并通过 Bloom Filter 来提升索引性能，加速了读取和查询速度。VT-tree 结合了 B+树、文件系统和 LSM 树的性能优势，采用 stitching 策略减少了不必要的数据拷贝，提升了整体性能，但这会导致数据碎片化进而影响查询和压缩性能，可以采用 DF(direction-finding)技术对其进行扩展。LDC 和 ALDC 策略采用了一种新的"自下而上"的方式，细化了压缩粒度，后者还采用动态调整阈值的方式来适应读写比例的波动。WiscKey 和 HashKV 采用的键值分离策略对写性能的提升作用比较明显，但它的范围查询也会受到影响，虽然可以采用 SSD 的 I/O 并行性来缓解，但仍会导致空间放大率高的问题，需要采用垃圾回收来减少旧的值所占用的空间，同时由于其自身的局限性，键值分离只适用于值占用空间较大的情况，当值占用空间较小时，采用传统的 LSM 树会更合适。在混合存储介质的场景下，键值存储系统需要根据用户提供的数据特性和不同存储介质的特性来进行存储。

通过对这些优化方案的对比和分析可以发现，个别优化策略在多种方案中均有涉及，例如 HashKV 和 WiscKey 都采用键值分离策略来提升 LSM 树的写性能，TRIAD，Leaper 和 SpanDB 等都采用冷热数据分离的策略，针对性地将数据存储在内存或磁盘中。而根据不同存储介质的特性，有多种方案选择将不同的优化策略相结合来提升键值存储系统的性能，例如 Leaper 将冷热数据分离的思想和缓存技术相结合，用机器学习的方法来将热数据预取到缓存中；SpanDB 将冷热数据分离的思想与混合存储技术相结合，把热数据存放在性能较高的 NVMe SSD 中，将冷数据置于普通 SSD 上。采用多种技术相结合的方式可以充分发挥各种技术的优势，有效缓解键值存储系统中出现的各种问题。于 LSM-trie，SlimDB 和 Lethe 等其他特殊的工作负载而言，只有在某些特定的场景下(如删除密集型、更新密集型等场景)，系统才会充分发挥键值系统高效的性能，因此需要针对不同的应用场景来选择合理的优化方案。

5 应用场景

文中第 2 节和第 3 节讨论并分析了 LSM 树自身存在的问题和可以提升的主要方面，由此可以根据 LSM 树的特点来对不同键值系统的性能进行进一步优化提升。本节对几个基于 LSM 树的应用场景和国产关系型数据库 TiDB 进行介绍。

5.1 LevelDB

LevelDB 是一个 Google 实现的非常高效的 KV 数据库，目前的 1.2 版本可以支持 billion 级别的数据量，因其设计良好，在这个数量级别下具有非常高的性能。和 Redis 这种内存型的 KV 系统不同，LevelDB 不会像 Redis 一样占用大量的内存空间，而是将大部分数据存储在磁盘上；其次，LevelDB 在存储数据时是根据键值有序存储的，相邻的键值在存储文件中是按顺序存储的；像大多数 KV 系统一样，LevelDB 的操作接口简单，基本操作包括读、写和删除记录，同时也支持对多条操作的原子批量删除操作；由于它自身支持快照功能，使得读数据的过程不受写操作的影响，保证了数据的一致性；此外，LevelDB 还支持数据压缩等操作，这对于减少存储空间

以及提高I/O效率都有帮助,具体实现细节在本文的2.2节中有详细阐述。从LevelDB的读取流程中可以发现,如果在L_0到L_n中存在较多的冗余数据,则会带来多次无效的I/O操作,因此可以根据不同的应用场景制定不同的压缩策略来及时清除冗余数据。

5.2 RocksDB

RocksDB是FaceBook开发的基于KV存储的一种嵌入式、持久化存储且适用于fast storage的存储引擎,它主要支持适用于多CPU场景、高效利用storage以及弹性架构和支持扩展。

在LSM树中,由于第L_0层的SSTable是无序的,在L_0与L_1层进行合并时会造成大量的写放大。为解决这一问题,RocksDB对L_0层的数据采用了分层合并策略,避免了查询性能的降低。RocksDB在LevelDB的round-robin策略的基础上支持"冷数据优先"和"删除数据优先"两种策略,前者选择需合并的冷数据来避免工作负载不平衡,保证热数据处于LSM树的较低层以降低总的写成本;后者选择有大量无效数据占用的SSTable以快速回收已删除数据占用的存储空间。

在基于LSM树存储的应用场景中,合并操作会消耗大量CPU和磁盘资源,这对查询性能会产生负面影响,而合并的时间直接取决于写入速率,一次写延迟往往会导致合并速率的降低。RocksDB对刷新和合并操作添加了一定的权限来控制写入速率,在执行每次刷新和合并操作前必须获取一定数量的许可,从而达到控制写入速率的目的。

5.3 HBase

HBase是一个高可靠性、高性能、面向列和可伸缩的分布式存储系统,它参考了Google的Bigtable[48]的设计。HBase基于主从属架构,将一个数据集划分为一组区域,其中每个区域由一个LSM树管理,并支持动态区域分割和合并策略。HStore存储是HBase存储的核心,其中包含了数据的更新和删除操作,数据传输的基本过程同LevelDB类似。关于HBase的LSM树的实现通常基于基本的分层合并策略,同时还支持其他的一些变体,如exploring合并策略和date-tiered合并策略。第一种合并策略会在所有参与合并的SSTable中选取写入成本最小的进行合并,这比传统的分层合并策略更有效,尤其是在SSTable数据分布不平衡时,因此被用作HBase中常用的合并策略;第二种合并策略对SSTable进行时间分区,以便系统实现高效的查询操作。

5.4 TiDB

TiDB是PingCAP公司自主设计、研发的开源分布式关系型数据库,同时支持在线事务处理与在线分析处理的融合、金融级高可用、实时HTAP、云原生的分布式数据库、兼容MySQL5.7协议和MySQL生态等重要特性,适用于高可用、强一致要求较高、数据规模较大等各种应用场景。TiDB采用的是存储与计算分离的架构,整个系统可以分为TiDB,PD,TiKV,TiFlash 4个组件,其中TiKV属于存储层,它是基于Multi-Raft协议在RocksDB之上实现的分布式高可用的键值数据库。TiKV将数据有序存储,根据键的范围将数据划分为多个大小接近的分片,当单个分片的数据量过大或过小时,会自动进行分裂或合并操作。其为每个分片维护多个副本存储在不同的TiKV节点上,分片的副本之间使用Raft协议进行同步,保证了数据的高可用性和强一致性。TiKV是针对联机事务处理的场景而设计的,在TiKV内部数据以行的形式进行存储,提供了原生的分布式事务支持,通过检测并发事务间的写冲突,实现了快照隔离。

6 总结与展望

现有的优化方案主要包括3种形式:通过更改LSM树本身的数据结构来提升LSM树的性能,主要体现在读性能方面;通过在键值存储系统的内存中增加索引结果来辅助LSM树最大化其性能优势;此外,还可以通过在磁盘中更换存储介质来弥补键值存储系统的缺陷。本文通过对现有的键值系统的优化方法进行分类和对比,从不同角度对键值存储系统的优劣进行分析。下面对未来的键值存储系统可能的发展方向做进一步展望。

(1)键值分离技术

本文中提到的WiscKey和HashKV都采用了键值分离技术,虽然它们在减少写放大方面的效果较好,但由于键值分离会产生额外的空间消耗,这就会出现空间放大率(空间放大率=数据存储的实际大小/数据的逻辑大小)较高的问题,进而导致存储成本较高。而采用键值分离的键值存储系统在更新密集型的工作负载下会频繁地触发垃圾回收的操作,导致其无法实现较高的性能。针对这一问题,可以通过对失效数据进行有效管理来实现就地更新,从而消除存储过程中的垃圾回收操作,避免系统中有效数据的重复写,减少系统的写放大,进而提升系统性能。

(2)非易失性存储器

如上文所述,非易失性存储器具有扩展性好、静态能耗低、非易失性等特点,它可以在系统宕机的情况下避免数据丢失。相较于DRAM,非易失性存储器在存储数据时具有非易失性的特点,研究人员可以利用这一点优势直接对内存中的部分数据持久化,但非易失性存储器也存在因其高效的刷新操作影响其使用寿命的问题。同时,硬件接口和系统软件必须适应非易失性存储器的优势,尤其是在I/O密集型的程序中,否则会降低系统的整体性能。Moneta-D[49]通过将文件系统权限检查移动到硬件中,并使用不受信任的用户空间驱动程序发出请求,来避免这种危险的发生。研究者也可以通过修改应用程序本身来进一步提高非易失性存储器为整个系统带来的性能优化。

(3)混合存储

从根本上看,存储系统的性能取决于其所依赖的存储介质,当今大部分键值系统采用的都是单一存储介质,虽然这些存储介质的优势都比较明显,但是它们各自都存在不同的缺点,比如:与非易失性存储器相比,SSD的容量较大,但成本高,且顺序性能和随机性能相对较低;而非易失性存储器顺序性能和随机性能较高,因此可以将二者结合使用,于是出现了混合SSD的存储架构SpanDB。但基于存储介质的混合存储[50]对实验平台要求较高,对研究人员来说具有较高的挑战性。

(4)内存/缓存技术

LevelDB中的SSTable是以Table的形式存在的,在一次

读操作中会涉及多个 SSTable 的操作,每使用一个 SSTable 必须先进行 open 操作,此时会将 index block 和 filter block 都读到内存中,但是 data block 仍保存在磁盘上。因此,用户在发起读取请求时,可以将数据先放入缓存中,下次读取数据时就可以直接从内存中读取而不需要经过磁盘,避免磁盘中大量的 I/O 开销。与在 RocksDB 中允许开辟多个 memtable 的道理相似,可以在内存中开辟多个缓存块,虽然这会占用额外的空间,但是一方面性能的提升在一定程度上都会导致其它方面性能的下降,而以牺牲空间为代价来换取读性能的提升对于整个系统来说是值得的。基于内存的存储系统也可以被看作是大型高性能存储系统的一个较好的发展方向,如 Ramcloud storage[51] 和 SILT[52]。同时,近几年对键值系统缓存的研究工作也逐渐兴起,如 DualKV 和 Segcache[53] 等,因此可以将其作为一个热点来进行研究。

结束语 本文从相关概念、技术手段、各类优化方案这几个方面对键值存储系统进行了综述,对各类优化方案进行了详细的讨论和利弊分析,并从写放大、压缩策略、硬件、混合存储系统以及特殊的工作负载方面对其进行分析和对比。最后对未来可能的发展方向进行了总结和展望。

参 考 文 献

[1] DECANDIA G,HASTORUN D,JAMPANI M,et al. Dynamo:amazon's highly available key-value store[C]// Symposium on Operating Systems Principles. 2007:205-220.

[2] LAKSHMAN A,MALIK P. CASSANDRA-A Decentralized Structured Storage System[J]. Operating Systems Review,2010,44(2):35-40.

[3] CHODOROW K,DIROLF M. MongoDB-The Definitive Guide:Powerful and Scalable Data Storage[M]// DBLP. 2010.

[4] O'NEIL P,CHENG E,GAWLICK D,et al. The log-structured merge-tree(LSM-tree)[J]. Acta Informatica,1996,33(4):351-385.

[5] HUA W D,GAO Y,LYU M,et al. Research on Bloom Filter:a survey [J]. Journal of Computer,2022,42(6):1729-1747.

[6] DEBNATH B,HAGHDOOST A,KADAV A,et al. Revisiting hash table design for phase change memory[J]. ACM SIGOPS Operating Systems Review,2015,49(2):18-26.

[7] LI J Y,YUE Y L,WANG W P:GHStore:A High Performance Global Hash Based Key-Value Store[C]// 27th DASFAA 2022:Virtual Event-Part I. 2022:493-508.

[8] KE Z M,LI Y Z,CHANG D W. Dual-KV:Improving Performance of Key-value Caches on Multilevel Cell Non-volatile Memory[C]// 50th International Conference on Parallel Processing Workshop. 2021:1-9.

[9] XIA F,JIANG D J,XIONG J. Analysis of factors affecting the performance of nonvolatile memory system [J]. Computer Research and Development,2014(S1):25-31.

[10] DAYAN N,IDREOS S. Dostoevsky:Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging[C]// Proceedings of the 2018 International Conference on Management of Data. 2018:505-520.

[11] PUGH W. Skip lists:a probabilistic alternative to balanced trees [J]. Communications of the ACM,1990,33(6):668-676.

[12] CHEN L,CAREY M J. LSM-based storage techniques:a survey [J]. The VLDB Journal,2020,29(1):393-418.

[13] AMUR H,ANDERSEN D G,KAMINSKY M,et al. Design of a Write-Optimized Data Store[J/OL]. https://repository.gatech.edu/server/api/core/bitstreams/79429d60-38b7-4885-9a88-c8211c7acba8/content.

[14] TING Y,WAN J G,PING H,et al. Building Efficient Key-Value Stores via a Lightweight Compaction Tree[J]. ACM Transactions on Storage,2017,13(4):29:1-29:28.

[15] RAJU P,KADEKODI R,CHIDAMBARAM V,et al. Pebblesdb:Building key-value stores using fragmented log-structured merge trees[C]// Proceedings of the 26th Symposium on Operating Systems Principles. 2017:497-514.

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

[17] PAN F F,YUE Y L,XIONG J. dCompaction:Speeding up compaction of the lsm-tree via delayed compaction[J]. Journal of Computer Science and Technology,2017,32(1):41-54.

[18] MEI F,CAO Q,JIANG H,et al. SifrDB:A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter[C]// ACM Symposium on Cloud Computing. 2018:477-489.

[19] YUE Y L,HE B S,LI Y Z,et al. Building an Efficient Put-Intensive Key-Value Store with Skip-Tree[J]. IEEE Transactions on Parallel and Distributed Systems,2017,28(4):961-973.

[20] BALMAU O,DIDONA D,GUERRAOUI R,et al. TRIAD:Creating Synergies Between Memory,Disk and Log in Log Structured Key-Value Stores[C]// 2017 USENIX Annual Technical Conference(USENIX ATC 17). 2017:363-375.

[21] SEARS R,RAMAKRISHNAN R. bLSM:A general purpose log structured merge tree[C]// Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. 2012:217-228.

[22] TENG D,GUO L,LEE R,et al. A Low-cost Disk Solution Enabling LSM-tree to Achieve High Performance for Mixed Read/Write Workloads[J]. ACM Transactions on Storage,2018,14(2):15:1-15:26.

[23] PRADEEP S,RICHARD P. SPILLANE,RAVIKANT M,et al. Building workload-independent storage with VT-trees[C]// FAST. 2013:17-30.

[24] BENDER M A,FARACH-C M,JOHNSON R,et al. Don't Thrash:How to Cache Your Hash on Flash[C]// Proceedings of the VLDB Endowment. 2012:1627-1637.

[25] VARDOULAKISM,SALOUSTROS G,PILAR G F,et al. Tebis:index shipping for efficient replication in LSM key-value stores[C]// EuroSys. 2022:85-98.

[26] CHAI Y P,CHAI Y F,WANG X,et al. Adaptive Lower-Level Driven Compaction to Optimize LSM-Tree Key-Value Stores [J]. IEEE Transactions on Knowledge and Data Engineering,2022,34(6):2595-2609.

[27] YANG L,WU H,ZHANG T,et al. Leaper:A learned prefetcher for cache invalidation in LSM-tree based storage engines [C]// Proceedings of the VLDB Endowment. 2020:1976-1989.

[28] CHANDRAMOULI B,PRASAAD G,KOSSMANN D,et al.

FASTER:A concurrent key-value store with in-place updates [C] // The 2018 International Conference. Houston: ACM, 2018:275-290.

[29] LU L Y,PILLAI T S,GOPALAKRISHNAN H,et al. WiscKey: Separating Keys from Values in SSD-Conscious Storage [J]. ACM Trans. on Storage,2017,13(1):5:1-5:28.

[30] LI Y,CHAN H H W,LEE P P C,et al. Enabling Efficient Updates in KV Storage via Hashing:Design and Performance Evaluation[J]. ACM Transactions on Storage,2019,15(3):20:1-20:29.

[31] HSIEH J W,KUO T W,CHANG L P. Efficient identification of hot data for flash memory storage systems[J]. ACM Transactions on Storage,2006,2(1):22-40.

[32] WANG P,SUN G Y,JIANG S,et al. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD[C] // European Conference on Computer Systems. 2014. 16:1-16:14.

[33] CHEN H,RUAN C Y,LI C,et al. SpanDB:A Fast,Cost-Effective LSM-tree Based KV Store on Hybrid Storage[C] // USENIX Conference on File and Storage Technologies. 2021:17-32.

[34] WANG Y Y,WEI H C,CHAI Y P. Performance optimization of LSM tree key value storage system based on ssd-smr hybrid storage [J]. Computer Science,2018,45(7):61-65.

[35] WU H Y. Research on persistent key value storage system based on dram-nvm hybrid memory [D]. Wuhan: Huazhong University of Science and Technology,2019.

[36] ROB D. pmemKV[OL]. [2022-05-27]. https://github.com/pmem/pmemkv.

[37] ISMAIL O,JOHAN L,ANISOARA N,et al. FPTree:A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory[C] // ACM SIGMOD Conference. 2016:371-386.

[38] CHEN S,JIN Q. Persistent b+-trees in non-volatile main memory[C] // Proceedings of the VLDB Endowment. 2015: 786-797.

[39] KANNAN S,BHAT N,GAVRILOVSKA A,et al. Redesigning LSMs for Nonvolatile Memory with NoveLSM[C] // USENIX Annual Technical Conference. 2018:993-1005.

[40] KAIYRAKHMET O,LEE S,NAM B,et al. SLM-DB: single-level key-value store with persistent memory[C] // 2019 Conference on File and Storage Technologies. 2019:191-205.

[41] ZHAN L,ZHANG Y,YU K. Design and Implementation of SCM and SSD based Hybrid Key-Value Store[C] // Proceedings of the 2019 International Conference on Artificial Intelligence and Computer Science. 2019:566-572.

[42] YAO T,ZHANG Y,WAN J,et al. MatrixKV:Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM[C] // 2020 USENIX Annual Technical Conference (USENIX ATC 20). 2020:17-31.

[43] WU X B,XU Y H,SHAO Z L,et al. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items[C] // USENIX Annual Technical Conference. 2015:71-82.

[44] MEI F. Research on optimization method of large-scale key value storage system based on log structure merging tree [D]. Wuhan:Huazhong University of Science and Technology,2019.

[45] REN K,ZHENG Q,ARULRAJ J,et al. SlimDB:A space-efficient key-value storage engine for semi-sorted data[C] // Proceedings of the VLDB Endowment. 2017:2037-2048.

[46] FAN B,ANDERSEN D G,KAMINSKY M,et al. Cuckoo filter: Practically better than bloom[C] // Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies. 2014:75-88.

[47] SARKAR S,PAPON T I,STARATZIS D,et al. Lethe:A tunable delete-aware LSM engine[C] // Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020:893-908.

[48] CHANG F. Bigtable:A Distributed Storage System for Structured Data[C] // 7th USENIX Symposium on Operating Systems Design and Implementation(OSDI). 2006.

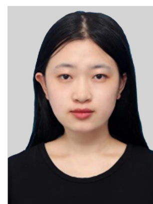
[49] CAULFIELD A M,MOLLOV T I,EISNER L A,et al. Providing Safe,User Space Access to Fast,Solid State Disks[C] // International Conference on Architectural Support for Programming Languages and Operating Systems. 2012:387-400.

[50] ZHU Q. Research on hybrid storage system [D]. Shanghai: Shanghai Jiaotong University,2012.

[51] OUSTERHOUT J K,AGRAWAL P,ERICKSON D,et al. The case for RAMClouds:Scalable high-performance storage entirely in DRAM[J]. ACM SIGOPS Operating Systems Review,2009,43(4):92-105.

[52] LIM H,FAN B,ANDERSEN D G,et al. SILT:A memory-efficient,high-performance key-value store[C] // Proceedings of the 23rd ACM Symposium on Operating Systems Principles. 2011.

[53] YANG J,YUE Y,VINAYAK R. Segcache:a memory-efficient and scalable in-memory key-value cache for small objects[C] // 18th USENIX Symposium on Networked Systems Design and Implementation(NSDI 21). 2021:503-518.



LYU Meng, born in 1998, postgraduate,is a member of China Computer Federation. Her main research interests include network storage and so on.

XIE Ping, born in 1979. Ph.D, professor, is a member of China Computer Federation. His main research interests include distributed file system,network storage, fault-tolerant storage and new storage device.

(责任编辑:何杨)