文章编号:0253-2778(2020)06-0825-07

一种读写均衡的高性能键值存储系统

吴加禹,李永坤,许胤龙

(中国科学技术大学计算机科学与技术学院,安徽合肥 230026)

摘要:日志结构合并树(LSM-tree)因能利用外存设备的顺序访问性能,被广泛应用于键值存储系统的核心数据结构.由于LSM-tree层次化的、有序的数据组织结构需要通过大量的数据合并操作维护,故引起了严重的写放大效应.最近的研究工作提出了若干优化方案缓解LSM-tree的写放大,但是牺牲了查询性能和空间利用率.为此基于LSM-tree的键值存储系统提出一种新的架构,其核心设计是采用键值分离的方案降低数据合并开销,并以一种新型的树状结构vTree为值维护一定程度有序性,保障高效的范围查询;同时为vTree设计了相应的数据合并和空间回收方法.实验结果表明,基于该架构实现的键值存储系统在写入、点查询、范围查询各方面有均衡的高性能表现,且空间开销较低.

关键词:日志结构合并树;写放大;键值分离;范围查询

中图分类号:TP311 **文献标识码**:A doi:10.3969/j.issn.0253-2778.2020.06.015

引用格式:吴加禹,李永坤,许胤龙. 一种读写均衡的高性能键值存储系统[J]. 中国科学技术大学学报,2020,50(6):825-831,838.
WU Jiayu, LI Yongkun, XU Yinlong. A read/write balanced high-performance key-value store[J]. Journal of University of Science and Technology of China, 2020,50(6):825-831,838.

A read/write balanced high-performance key-value store

WU Jiayu, LI Yongkun, XU Yinlong

(*School of Computer Science and Technology, University of Science and Technology of China, Hefei* 230026, *China*)

Abstract: Log-structured merge tree (LSM-tree) is widely used as the core data structure of modern key-value stores due to its ability to utilize sequential access performance of external storage. However, it suffers from expensive merge operations to maintain the layered and ordered data organization, which induces significant write amplification. Recent researches have proposed various optimizations to mitigate write amplification, but at the cost of query performance and space utilization. A new architecture for LSM-tree based key-value store is proposed. The key idea is to leverage the key-value separation design to mitigate merge overhead, while maintaining a certain degree of order for values by using a new tree structure called vTree to improve the range query performance. In the meantime, corresponding data merge and space reclaim algorithms were developed for the vTree. The experimental results show that the key-value store developed with this architecture has well balanced performance on write, point lookup, and range query, with low space cost.

Key words: log-structured merge tree; write amplification; key-value separation; range query

0 引言

随着互联网技术的飞速发展以及网络终端接入的普及,博客、即时通信、短视频分享等新型服务平台不断涌现,获取服务的用户也越来越多,使得网络上的数据规模不断增长.以知名社交网站Facebook为例,其每天都会新增数十亿条内容,对数据的访问量也达到了每秒钟几亿次[1].为了应对海量数据的存储、访问带来的挑战,基于键值(key-value,KV)模型的存储系统近年来受到广泛关注.键值存储系统将数据抽象成键值对(key-value pair)的形式,key可由任意字符串表示,value可以是文本、图片、视频等任意类型的数据,并使用Put、Delete、Get和Scan这种简单的命令执行数据的写入、删除、点查询和范围查询.这种简单的存取模型使得键值存储系统具有高效的读写性能和可扩展性,作为后端存储引擎被广泛部署在文件系统、对象存储系统、SQL数据库等数据密集型应用中.

键值存储系统一般使用哈希表(Hash table)、B+树(B+-tree)或日志结构合并树[2](LSM-tree)

收稿日期:2020-03-13;**修回日期**:2020-06-04
基金项目:国家自然科学基金(61772484);统筹推进世界一流大学和一流学科建设专项资金(YD2150002003)资助.
作者简介:吴加禹,男,1994年生,硕士生,研究方向:键值存储系统.E-mail:wujy@mail.ustc.edu.cn
通讯作者:李永坤,博士/副教授.E-mail:ykli@ustc.edu.cn

作为核心数据结构，其中，主流系统如Google公司的LevelDB[3]和Bigtable[4]，Facebook公司的RocksDB[5]等都基于LSM-tree实现.这种数据结构通过批量地将对外存的随机写转换成顺序写，有效利用了外存设备顺序写吞吐量显著高于随机写的特征.为了维护键值对外存的有序性以提供高效的数据查询能力，LSM-tree需要周期性地合并外存的键值对.这种数据合并操作导致同一键值对被重复写入外存多次，引起严重的写放大效应，一方面消耗了大量的外存带宽，显著影响系统性能，另一方面还会缩短外存设备的寿命.最近的研究工作致力于降低LSM-tree的数据合并对写性能的影响，但是不同程度地牺牲了查询性能或空间利用率[6-8].在大多数应用场景下，键值存储系统的读写性能都很重要[9]，如网络应用除了存储用户发布的消息外，还需要处理其他用户的访问请求，同时可能对用户数据进行大范围的分析处理；有学者提出使用键值存储系统作为SQL数据库的底层存储引擎，对于复杂的SQL查询操作更是对系统的综合性能提出了全方位要求[10].

本文为基于LSM-tree的键值存储系统提出了一种新的架构，相比传统架构在提高写性能和点查询性能的同时保证了相当的范围查询性能，拥有读写均衡的高性能表现.本文的主要工作有：

(Ⅰ)基于键值分离的思想提出为key和value维护不同程度的有序性，以LSM-tree存储key和value地址，利用其较强的有序性提供快速的数据索引，并设计了与LSM-tree松耦合的树状结构vTree存储value，通过放松value的有序性限制来降低数据合并开销，同时保证范围查询依然能够顺序读取外存；

(Ⅱ)为vTree设计了高效的数据合并策略和相应的空间回收方法，保障了较低的数据合并开销和存储空间开销；

(Ⅲ)基于RocksDB完成了系统原型实现以及性能实验，且接口与RocksDB相兼容，可以直接替换RocksDB部署到应用环境.

1 背景技术介绍

1.1 基于LSM-tree的键值存储系统

我们首先介绍LSM-tree的数据组织结构和读写流程.图1展示了基于LSM-tree的键值存储系统的传统架构，包括LevelDB、RocksDB等流行系统均采用类似架构.系统分为内存与外存两部分，内存中维护两个写缓存MemTable和Immutable MemTable，用来为用户最近写入的键值对提供快速存储和访问，键值对在写缓存中按key的字典序有序存储.外存中的键值数据被组织成多层结构，简称第i层为L_i，L_{i+1}的容量为L_i的10倍.每层由若干SSTable(sorted string table)文件组成，键值对在SSTable内有序存储，且除L_0外各层的SSTable之间同样有序，即同一层SSTable之间key范围不存在重叠.除存储键值对外，SSTable中还存有一些元数据用于支持查询操作.此外，外存中还维护了一个日志文件WAL(write ahead log)，在系统掉电重启后通过重放其中的内容恢复写缓存中丢失的数据，以及一个元数据文件METAFILE，用来记录SSTable文件的增删等元数据信息.

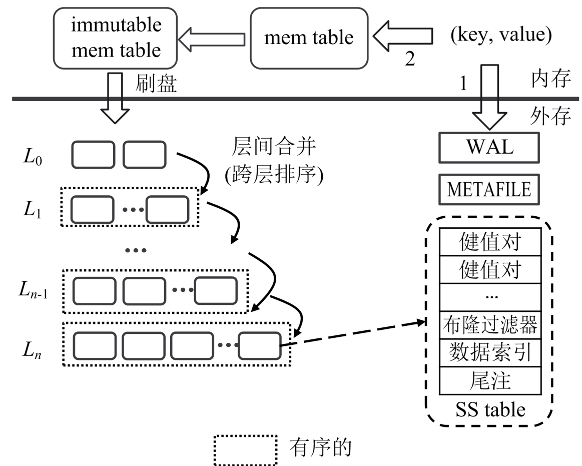


图1 基于LSM-tree的键值存储系统

Fig.1 LSM-tree based key-value store

向LSM-tree写入键值对时，首先将其追加写到WAL，然后写入MemTable. MemTable被写满后转换成Immutable MemTable，并生成一个新的MemTable接收后续写入. Immutable MemTable在后台将键值对组织成一个SSTable文件追加写入L_0，称为刷盘(flush).当L_i达到最大容量时，该层SSTable会通过层间合并(compaction)操作被合并进下一层.层间合并操作的流程如下：首先从L_i选择一个SSTable(对于L_0，选择所有存在key范围重叠的SSTable)，并选择L_{i+1}中所有与该SSTable存在key范围重叠的SSTable，将这些SSTable中的键值对排序，排序结果按固定大小(例如64MiB)切分生成一组新的SSTable并写入L_{i+1}.层间合并保障了键值对在除L_0外各层内均按key有序组织，而分层是为了保证数据合并时上下层的数据量差异不会过大.系统在每次刷盘和层间合并完成后向METAFILE中记录SSTable的变更信息.

对已有键值对的更新和删除同样以向MemTable写入一个新键值对的方式实现，不需要在外存进行原地更新(update-in-place).其中，删除操作即为写入一个value为特殊删除标记的键值对.老键值对的实际清理在层间合并时完成，相同key的键值对只有较新版本会出现在层间合并的排序结果中，被更新或删除的无效版本直接丢弃.

从LSM-tree中读取一个键值对时，由于同一个key可能在不同层存在多个版本，首先需要搜索MemTable与Immutable MemTable，然后从低向高逐层执行一次二分查找(L_0需要搜索每一个SSTable).由于新版本的数据一定出现在更低层，因此返回最先搜索到的结果.此外，bLSM[11]引入了布隆过滤器[12](Bloom filter)技术来跳过一些层的I/O，以减少读操作的I/O次数.范围查询分为两

步,由于查询范围内的键值对分布在内外存各层之中,第一步需要在各层定位到查询范围的起始位置,即第一个大于或等于起始key的键值对,相当于在每一层执行一次点查询;第二步从各层的起始位置开始顺序读取所有查询范围内的键值对,并将各层的查询结果排序作为最终结果.

从上述读写流程可以看出,在LSM-tree中所有数据都通过刷盘和层间合并操作批量地顺序写入外存,没有任何随机写,从而能利用外存设备顺序访问性能远高于随机访问的特点获得较高的吞吐量.同时,其层次化组织的有序数据也能提供较好的点查询和范围查询性能.由于层间合并操作每合并一个SSTable到下一层都需要从外存读取并写回多个SSTable,导致键值对在每一层都会被重复写多次,引起了严重的写放大效应.由于每层的数据量是上一层的10倍,从 L_i 合并一个SSTable到下一层时至少需要与 L_{i+1} 的10个SSTable合并,因此写放大在一层中可达10倍以上[7].

层间合并引起的写放大效应消耗了大量的外存带宽,其与前台请求竞争I/O资源,严重影响了键值存储系统的写性能,同时还会缩短外存设备的寿命.针对这个问题,工业界与学术界展开了大量研究工作.Facebook利用SSD的并发性能设计了RocksDB[5],通过多线程执行层间合并来加快数据合并速度,但没有从根本上缓解写放大问题;PebblesDB[6]将LSM-tree的各层数据划分成若干子区间,并允许区间内部的SSTable之间存在key范围重叠以减少合并次数,但是这种方法增加了查询操作的I/O次数,同时损害了点查询和范围查询性能;LSM-Trie[7]结合字典树和哈希表来组织数据,在降低写放大的同时也提供了优秀的点查询性能,但是不支持范围查询操作.

1.2　键值分离的存储结构

区别于修改LSM-tree结构本身,WiscKey[8]针对现代SSD设备提出了键值分离的设计,通过将value从LSM-tree中分离存储来降低LSM-tree的大小,从而降低系统整体的数据合并开销.键值分离的设计由于其良好的性能表现得到了广泛关注,图2展示了其数据组织结构.在键值分离的存储系统中,写入键值对时将value直接以追加写的方式存储在外存的一个循环日志文件vLog中,并将key和value在vLog上的地址组成新的键值对通过MemTable写入LSM-tree.查询数据时,首先从LSM-tree获取value地址,然后在vLog中读取实际value.鉴于value地址的大小仅若干字节,在多数场景下value的平均大小显著大于key和value地址[13],因此LSM-tree的体积和层数在键值分离后明显降低,整个系统的写放大可以降低到2以下.同时,随着LSM-tree层数的降低,点查询以及范围查询的定位步骤I/O次数也有所减少.实际运行时,一个体积很小的LSM-tree可以几乎整个被缓存到内存里,此时点查询仅会对vLog产生一次I/O.

在键值分离以后,vLog需要额外的垃圾回收

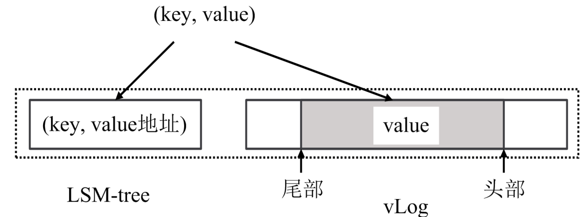


图2　键值分离的存储结构
Fig. 2　Key-value separated structure

(garbage collection, GC)机制回收空间.虽然LSM-tree在执行层间合并时即可丢弃被更新或删除的键值对,但索引的无效value仍然分布在vLog中.为了保障对外存的顺序写,vLog的GC方案如下:在vLog中同时存储整个键值对,当外存空间不足时,从vLog尾部开始遍历键值对,使用key向LSM-tree查询value地址是否与最新版本一致,并将一致的键值对重新写入头部,然后向LSM-tree更新value地址,遍历过的部分即可删除以回收空间.

从以上描述可看出,键值分离系统存在两个缺陷.第一,value在vLog中是完全无序存储的,范围查询时需要以随机方式读取value,而外存设备的随机读吞吐量远低于顺序读,因此会显著影响范围查询的性能;第二,GC操作需要重写大量的有效数据,引起额外的I/O开销,且执行过程中需要为每一个键值对访问LSM-tree,由于LSM-tree的读写操作开销较大,最终导致空间回收效率低,既影响了前台性能,同时还使系统的空间开销显著增大[14].

2　系统设计

2.1　系统概览

为克服上述问题,获得读写均衡的高性能表现,我们为基于LSM-tree的键值存储系统设计了一种新的数据组织结构,其核心思想是为key和value适用差异化的有序性限制,以尽可能低的写开销维护高查询性能.

图3是基于该结构实现的键值存储系统架构.受键值分离思想的启发,我们将外存的key和value存储在不同区域,以有序性较强的LSM-tree存储key和value地址,为查询操作提供快速索引.同时,设计了vTree结构来存储value,通过放松value部分的有序性限制来降低合并开销,但仍保证在执行范围查询时能够有效利用外存的顺序读取性能.此外,在内存中维护有一个元数据表用于辅助后续细粒度的GC操作.

2.2　数据读写流程

与传统的LSM-tree一样,最近写入的数据首先被存储在内存的MemTable以及Immutable MemTable中.区别在于刷盘时将value按其对应的key有序组织,并切分成一组固定大小(默认8MiB)的vTable文件分离存储,同时将key和value在vTable的地址作为新的键值对组织成SSTable写入LSM-tree的 L_0 层.Value地址记录了value所在vTable的ID、在vTable中的偏移量以及value

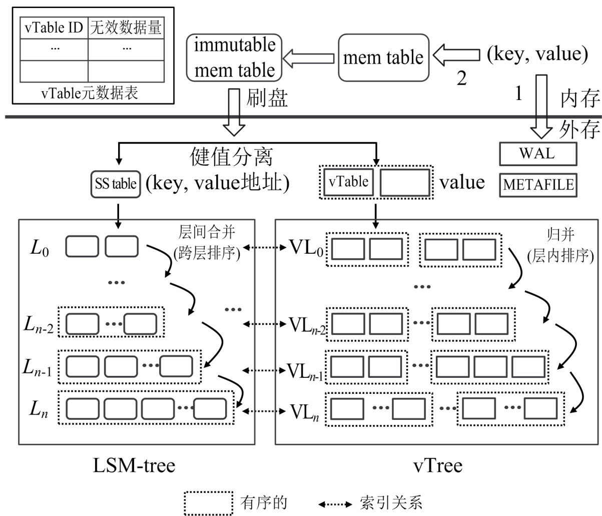


图3 系统架构

Fig. 3 System architecture

大小. 在读取数据时，首先利用LSM-tree快速索引value地址，并根据地址在vTable中读取实际value. 由于value仍保持了一定的有序性，在执行范围查询时依然能在多个vTable中顺序读取.

与 L_0 的SSTable类似，不同刷盘操作生成的各组有序vTable之间是存在key范围重叠的，随着组数增多value的整体有序性会越来越差，一次范围查询会分布到多组有序vTable中，退化为随机读. 为此，我们设计了树状结构vTree来管理vTable，进一步维护value的有序性. 在vTree中，vTable仍然以层次化组织，记第 i 层为 VL_i，每次刷盘生成的vTable直接追加到 VL_0 中. 不同于LSM-tree，vTree中每一层均不要求完全有序，当vTable向高层流动时仅与同层有key范围重叠的vTable合并，新生成的vTable直接追加到下一层，称该操作为归并(Merge). 这种设计一方面显著降低了数据合并带来的写放大，value在每层仅会被写入一次；另一方面每次归并均使一层中key范围重叠的vTable合并成一组有序的vTable，加强了value的有序性，避免了范围查询退化成随机读.

这种设计方案的挑战在于：第一，vTable在合并后value地址发生变化，会导致LSM-tree的索引失效；第二，由于采用键值分离的设计，vTree也需要额外GC操作回收空间，并且归并操作会导致被合并的vTable失效，进一步增加空间回收开销. 针对这两个问题，我们为LSM-tree和vTree维护了一种松耦合关系，由LSM-tree的层间合并触发vTree的归并，并使 L_i 唯一对应于 VL_i，即 L_i 索引的value均保存在 VL_i 中，在 L_i 执行层间合并的同时执行 VL_i 的归并.

2.3 数据合并方法

前文提到我们使用树状结构vTree管理vTable，通过归并操作合并同层key范围重叠的vTable来维护value有序性. 为保证LSM-tree索引的正确性，需要在vTable合并后及时更新LSM-tree中的value地址. 若由vTree独立执行归并，为避免无效value被重写，在vTable合并过程中需为每个value访问一次LSM-tree查询其是否有效，并在将有效value写入新的vTable后重新向LSM-tree中写入其新地址. 在1.2节中已经提到读写LSM-tree存在很大的开销，且为了能够访问LSM-tree还需要在vTable中存储value相应的key，增加了vTree的空间占用. 由于向LSM-tree重写value地址会导致高层的地址被写到低层，在一定程度上影响了对最近写入数据的查询性能.

针对以上问题，本文在LSM-tree执行层间合并的同时执行vTree的归并，且令归并范围(参与归并的键值对的上下限)等于层间合并的范围. 由于层间合并操作只合并新版本的键值对，需要把结果写回输出层，若能利用层间合并的排序结果来合并vTable，并将写回输出层的value地址修改为vTable合并后的最新地址，即可避免额外读写LSM-tree，同时还避免了合并vTable的排序开销.

图4展示了由层间合并触发的归并流程，图中阴影部分为本次层间合并涉及的数据范围. 我们首先假设 L_i 索引的value一定保存在 VL_i 中，当LSM-tree执行从 L_i 到 L_{i+1} 的层间合并时，对于合并的每个键值对，若其属于 L_i (换言之，其索引的value属于 VL_i)，则将其索引的value读出并写入新生成的vTable中. 新的vTable按同样的固定大小切分，并直接追加到 VL_{i+1}. 由此可知，在层间合并开始前，L_i 索引的value分布在 VL_i 的若干组互相之间key范围重叠的vTable中，这些value在合并结束后被重新组织成一组有序vTable合并进了 VL_{i+1}，它们的地址也被合并到了 L_{i+1}. 回想在刷盘后 L_0 索引的value都存储在 VL_0 中，当value地址从 L_0 合并进 L_1 时其索引的value也都进入了 VL_1，以此可以推定 L_i 索引的value一定都存储在 VL_i 中.

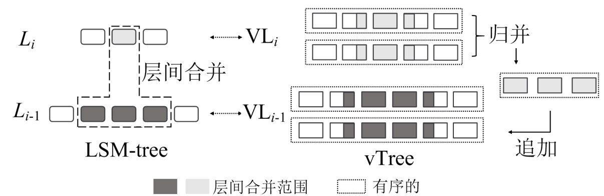


图4 vTree的数据合并

Fig. 4 Data merge of vTree

考虑某一时刻 L_i 存在 N 个SSTable，当这些SSTable均被合并到 L_{i+1} 后，其索引的value也被组织成 N 组vTable合并到了 VL_{i+1}. 由于这 N 个SSTable之间是有序的，其vTable之间也必然有序，从而共同在 VL_{i+1} 组成了一组有序vTable. 由于LSM-tree中每一层的容量为上一层的10倍，可以期望vTree中每层有近10组这样的有序vTable，范围查询会在各组vTable中顺序读取. 这样，当范围查询操作的查询长度(一次范围查询读取的键值对个数)较长时，可以在各组有序vTable中读取较多value，有效利用外存的顺序访问性能. 当查询长度较短时，从各组vTable读取到的value数量较少，将导致对顺序访问性能的利用能力下降. 随着查询长度变短，定位步骤的开销所占比重

变大,而 LSM-tree 部分在键值分离后较小,定位速度较快,因此在实际运行时上述架构的长、短范围查询性能均能与键值分离前的 LSM-tree 结构相当.

同样基于层次间的数据量差异,LSM-tree 中大约 99%的键值对都存储在最高两层,由于 L_i 唯一对应于 VL_i,这部分键值对索引的 value 也都存储在 vTree 的最高两层.在范围查询时,多数读取都落在这两层中,低层的各组有序 vTable 上仅有极少读取,难以发挥顺序读的性能优势,同时对整体查询性能的影响也较小,因此为低层的 vTable 执行归并对范围查询性能提升有限,然而写开销却与高层一致.为此,我们引入了延迟归并(lazy merge)机制避免合并低层 vTable,来进一步降低数据合并开销.如图 5 所示,假设 LSM-tree 最高层为 L_n,当 L_0～L_{n-2} 执行层间合并时并不触发归并,其索引的 vTable 均保存在同一层,直到执行 L_{n-2} 到 L_{n-1} 的层间合并时才通过归并将这些 vTable 一起合并到 VL_{n-1} 中.基于延迟归并机制,由 vTable 的合并引起的写放大被进一步降低到 2,并且最后两层中占系统总量约 99% 的 value 的有序性依然得到了维护.

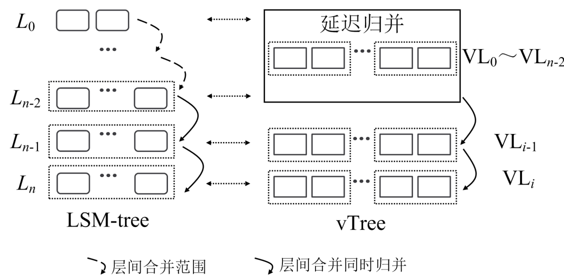


图 5　延迟归并机制

Fig. 5　Lazy Merge mechanism

2.4　空间回收方法

在键值分离后,需要额外的 GC 操作来删除 vTree 中的无效 value 回收空间.不同于 vLog,vTree 中的 value 以 vTable 为单位组织,可以选择任一 vTable 作为 GC 目标.为了提高 GC 效率,避免重写过多有效数据,我们仅选择无效数据占比较高的 vTable 作为目标.为此,我们在内存中维护了一个哈希表记录各 vTable 中的无效数据量,称为 vTable 元数据表.每当一个新的 vTable 生成时将其加入元数据表,记录无效数据量为 0.在层间合并时,统计合并前后的 SSTable 在各 vTable 中索引的 value 大小,分别记为 VS_{in} 和 VS_{out}.因为被删除的键值对不会再次出现在合并结果中,所以 VS_{in}-VS_{out} 即为该 vTable 增加的无效数据量,合并结束时以该结果更新 vTable 元数据表中的相应表项.

基于统计的无效数据量,若任一 vTable 中的无效数据占比超过了预定义值 gc_threshold(默认为 30%),则将其作为 GC 目标.若使用类似 vLog 的方案对 vTable 执行 GC,则仍需要为每个 value 访问 LSM-tree.为了避免该开销,我们将 GC 操作延迟到后续的归并中执行.具体来说,在每次层间合并结束后,若某个 vTable 的无效数据量被更新且无效数据占比超过 gc_threshold,仅为该 vTable 加上一个 GC 标记但不立即回收空间.在归并过程中,若发现某 value 所在的 vTable 存在 GC 标记,无论该 vTable 属于哪一层,均将此 value 一起写入新生成的 vTable,如图 6 所示.

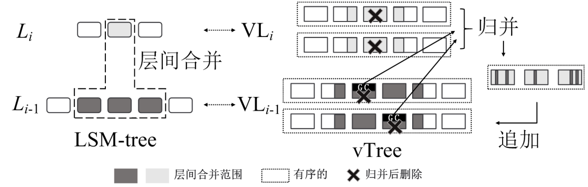


图 6　vTree 的空间回收

Fig. 6　Space reclaim of vTree

归并结束后,参与归并的 vTable 中除 key 范围在归并范围边缘的个别 vTable 外,其余 vTable 的有效 value 均被写入了新的 vTable,可以直接删除这些文件.对于范围边缘的 vTable 中遗留下的无效 value,考虑 value 的平均大小远大于地址,且 SSTable 和 vTable 存在大小差异(如 RocksDB 中 SSTable 默认大小为 64MiB[5],vTable 默认大小为 8MiB),一个 SSTable 可以索引大量的 vTable,因此这些 value 仅占归并总量的较少部分,空间开销较小,且会因层间合并选择算法的局部性很快在相邻范围的归并中被删除.

2.5　崩溃恢复

键值存储系统需要提供持久化保障,能够在系统意外崩溃后的下一次重启时恢复丢失的数据.在上述基于 vTree 的架构中,键值对写入系统时仍首先保存在 MemTable,在刷盘时才执行键值分离,因此依然使用 WAL 保障系统的数据恢复,以提供与传统基于 LSM-tree 的键值存储系统同等的持久化保障.此外,vTree 也利用 METAFILE 持久化 vTable 相关元数据.由于 vTable 的创建、删除和无效数据量的更新均在层间合并或刷盘时进行,这些信息可以和 SSTable 的元数据一起记录到 METAFILE 中.

2.6　开销分析

本节我们对比 vTree 方案与现有技术的读写开销.首先比较写流程的开销.根据 2.3 节的描述,vTree 方案相比键值分离前的传统架构减少了 value 在外存被重写的次数,降低了写流程的 I/O 开销,因而有更高的写性能.相比 vLog 方案,vTree 对 value 有序性的维护引入了额外的 value 合并操作,但 value 仅会在 vTree 的最高两层各被合并一次,I/O 开销相对较低.此外,vTree 在回收空间时能快速识别出无效数据占比高的 vTable 执行 GC,相比 vLog 减少了需要重写的有效数据量,同时还消除了 vLog 在 GC 时对 LSM-tree 的额外读写开销,有更高的空间回收效率;因此,在更新操作较少的场景下,vTree 的写开销略高于 vLog,而在更新操作较多需要频繁回收空间时,vTree 相比 vLog 有相当乃至更高的写性能,以及更低的空间开销.

下面对比读流程的开销.相比于传统架构,

vTree方案降低了LSM-tree部分的体积和层数，在读取数据时有更快的索引速度，与vLog方案类似.同时，由于vTree维护了value在外存的有序性，相比vLog在范围查询时消除了大量的随机I/O，避免了键值分离设计引起范围查询性能下降.

vTree方案对元数据表的维护存在一定内存开销.元数据表需要为系统中每一个vTable记录其ID和无效数据量，两者各占用4B内存.按照vTable的默认大小8MiB计算，每个vTable元数据的内存开销仅为其外存开销的不到0.0001%，几乎可以忽略不记.

3 实验结果

本节通过实验比较基于vTree方案实现的系统、传统基于LSM-tree的键值存储系统以及基于vLog的键值分离存储系统的性能表现.实验在戴尔PowerEdge R730服务器上运行，配置如表1所示.

RocksDB是目前最流行的基于LSM-tree的键值存储系统之一，我们使用RocksDB作为传统架构的代表，并在RocksDB的代码基础之上实现了vTree以及vLog方案.其中，vLog根据WiscKey[8]的描述实现，下文以vTree和vLog指代基于其实现的整个系统.实验中，vLog在无效数据总量超过30%时开始后台GC，且不限制空间使用总量；vTree配置vTable大小为8MiB，gc_threshold为30%；各系统使用4个线程并发执行层间合并，其余均按默认配置.所有工作负载均由YCSB[15]生成，并由16个前台线程并发执行，生成的键值对中key的大小为24B，value大小平均为1KiB，且除特别说明外所有工作负载的键值数据遵循Zipf分布，zipfian常数为0.9.

表1 实验环境

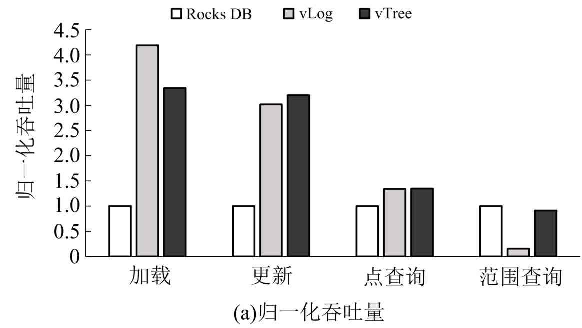
Tab.1 Experimental environment

处理器	Intel(R) Xeon(R) E5-2650 v4 @ 2.2GHz
内存	64GiB
外存	Samsung860EVO480GiBSSD
操作系统	Ubuntu18.04LTS

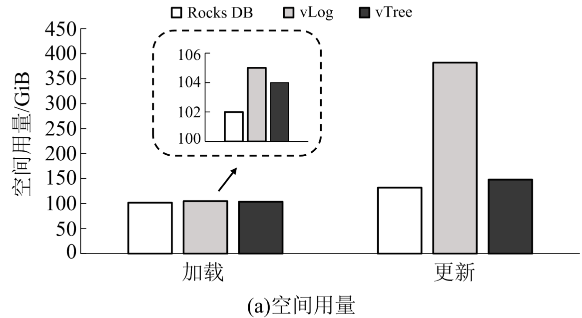
3.1 综合性能

我们首先测试RocksDB、vLog和vTree分别在各类读写工作负载下的综合性能，比较它们的吞吐量以及存储空间开销.本组实验分加载、点查询、范围查询、更新4个阶段.首先，在加载阶段，各系统从空开始载入均匀分布的100GiB键值数据，这些数据中不会出现重复的key.然后，在这些键值数据之上执行接下来各阶段的操作：①点查询10GiB键值数据；②范围查询10GiB键值数据，单次查询量为1MiB(即查询长度为1000个键值对)；③在已加载的100GiB数据之上更新300GiB键值数据.图7(a)展示了各系统在不同操作阶段相对于RocksDB的归一化吞吐量，图7(b)展示了加载和更新阶段结束后各系统的空间用量.

我们首先比较写性能.得益于键值分离设计带来的低数据合并开销，vTree在加载和更新阶段的吞吐量均显著优于RocksDB，分别是RocksDB的



(a)归一化吞吐量



(a)空间用量

图7 综合性能对比

Fig.7 Overall performance comparison

331%与325%.相比于vLog，由于vTree存在额外的value合并开销，在加载阶段吞吐量略低，约为vLog的79%.由于加载阶段不涉及数据更新，3个系统在加载结束时的空间开销接近，vTree的空间开销略高于RocksDB，这是因为键值分离带来了额外的地址和元数据存储开销，同时略低于vLog，并且vTree中只需要存储value，而vLog中需要存储整个键值对.在更新阶段，vLog受到GC开销的影响性能优势开始下降.得益于更高效的空间回收方法设计，vTree的吞吐量约为vLog的106%，并且空间开销显著低于vLog.更新结束后，vTree的空间开销为RocksDB的112%，而vLog的空间开销高达RocksDB的289%.

读性能比较：由于相比vLog我们未针对点查询优化，在点查询阶段vTree与vLog的吞吐量基本相同.相比于RocksDB，vTree的键值分离设计降低了LSM-tree部分的体积和层数，减少了点查询操作的平均I/O次数，因此吞吐量达到前者的134%.在范围查询阶段，得益于vTree对value有序性的维护，键值分离并未明显影响vTree的范围查询性能，吞吐量达到了RocksDB的94%.相比于vLog，vTree的范围查询吞吐量是前者的606%.需要注意的是，由于vTree中各层的value并非完全有序，范围查询性能还会受查询长度的影响，查询长度越短对value的读取越倾向于随机读.

从实验结果可以看出，相比于RocksDB和vLog，vTree在写入、点查询、范围查询各方面性能均衡，在3者间均有最优或接近最优的吞吐量，并且相比RocksDB未显著增加空间开销.本组实验中，各阶段仅执行了一类操作，实际生产环境下通常有各类操作混合在一起，我们将在下文比较在这类场景下vTree与其他系统的性能表现.

3.2 范围查询性能

本节比较 RocksDB、vLog 和 vTree 在不同查询长度下的范围查询性能.各系统基于预加载的 100GiB 键值数据执行范围查询,单次查询长度分别为 20、100、1000、10000 个键值对,且每类查询长度共查询 10GiB 数据.图 8 展示了各系统在不同查询长度下的吞吐量.

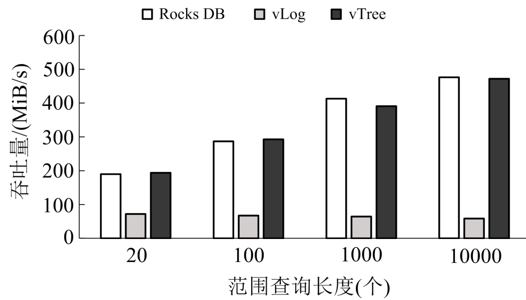


图 8　不同查询长度下的范围查询性能对比

Fig. 8　Range query performance with different scan length

根据实验结果,vTree 的吞吐量按查询长度的递增顺序分别是 RocksDB 的 102%、102%、94%、99%,是 vLog 的 269%、434%、605%、805%.由此可以看出,在各种查询长度下 vTree 的范围查询性能均与 RocksDB 相当,且显著优于 vLog.此外,RocksDB 和 vTree 的吞吐量均随查询长度变短而降低,原因有两点:第一,范围查询长度越短,在 vTree 中对 value 的读取越倾向于随机读;第二,随着范围查询长度变短,定位步骤的开销所占比重变高,而定位步骤类似于在 LSM-tree 的各层中执行一次点查询,性能低于顺序读.由于 vTree 的定位速度快于 RocksDB,在范围查询长度较短时该优势对整体性能影响更明显,因此 vTree 的吞吐量反而略高于 RocksDB.而 vLog 的定位步骤和读取 value 步骤皆为随机读,因此吞吐量随查询长度变化不大.

3.3 YCSB 核心工作负载性能

YCSB 提供了 6 类模拟实际应用环境的工作负载,如表 2 所示.我们比较 RocksDB、vLog 和 vTree 在这 6 类工作负载下的性能表现.各系统基于预加载的 100GiB 键值数据执行各类工作负载,且针对不同工作负载的实验相互独立.图 9 展示了各系统在不同工作负载下的吞吐量.

表 2　YCSB 核心工作负载

Tab. 2　YCSB core workloads

工作负载	操作组成
A	50%更新,50%点查询
B	5%更新,95%点查询
C	100%点查询
D	5%写入,95%点查询
E	5%更新,95%范围查询
F	50%读后写(read-modify-write),50%点查询

根据实验结果,对于写入频繁的工作负载 *A* 和 *F*,vTree 的吞吐量分别达到了 RocksDB 的 337%与 255%,与 vLog 性能接近.其中,工作负载 *F* 的写入为读后写,此时读操作占比更大,由于 vTree 与

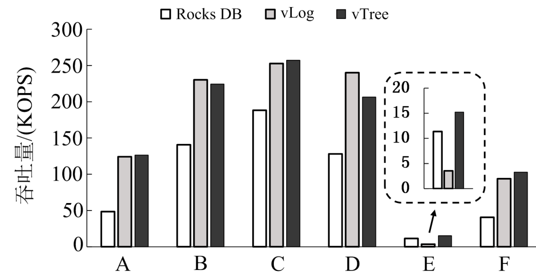


图 9　YCSB 核心工作负载性能对比

Fig. 9　Performance comparison under YCSB core work loads

RocksDB 的读性能差距相对较小,因此吞吐量差距小于工作负载 *A*.同理,对于点查询为主的工作负载 *B* 和 *D*,两者的性能差距更小,但是 vTree 的吞吐量仍分别达到了 RocksDB 的 160%与 161%.值得注意的是,在工作负载 *D* 下,vTree 的吞吐量略低于 vLog,为后者的 86%,这是因为工作负载 *D* 的写操作为写入新的键值对而非更新已有键值对,因此不会触发 vLog 的 GC,此时 vLog 的写性能略高于 vTree.工作负载 *C* 为只读操作,在 3.1 节中已作分析,不再赘述.对于范围查询为主的工作负载 *E*,vTree 的吞吐量达到了 vLog 的 432%,同时在写操作开销的影响下,vTree 的吞吐量相比于 RocksDB 也有明显提高,达到了后者的 134%.

从实验结果可以看出,在模拟实际应用环境的各类读写混合的工作负载下,vTree 相比于 RocksDB 和 vLog 仍有最优或接近最优的性能表现.

4 结论

本文研究了基于 LSM-tree 的键值存储系统的数据组织结构,指出了现有系统设计无法同时兼顾高读写性能的缺点,提出了一种新的系统架构,基于键值分离的思想设计了与 LSM-tree 松耦合的 vTree 结构管理 value,通过为 key 和 value 维护不同程度的有序性获得了均衡的读写性能,并完成了系统实现.实验结果表明,相较传统基于 LSM-tree 的键值存储系统和基于 vLog 的键值分离存储系统,该系统在各类工作负载下均表现优秀,且空间开销较低.

参考文献(References)

[1] NISHTALA R, FUGAL H, GRIMM S, et al. Scaling memcache at facebook[C]// Presented as part of the 10th USENIX Symposium on Networked System Design and Implementation. 2013: 385-398.

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

[3] GHEMAWAT S, DEAN J. LevelDB[EB/OL]. [2020-06-01]. https://github.com/google/leveldb.

[4] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems, 2008, 26(2):1-24.

(下转第 838 页)