文章编号:1671-4598(2024)06-0262-07 DOI:10.16526/j.cnki.11-4762/tp.2024.06.038 中图分类号:TP391.1 文献标识码:A

基于 LSM-Tree 的键值存储系统的读写性能优化

程浩津，胡乃平
（青岛科技大学 信息科学技术学院，山东 青岛 266061）

摘要：在写密集型工作环境中，日志结构合并树（LSM-Tree）已逐渐成为主流存储系统，LSM-Tree 存在读操作速度慢、写操作成本高、范围查询操作效率低等问题；针对这些问题，为提升 LSM-Tree 的性能进行了研究，提出了一种基于 LSM-Tree 的键值存储系统的读写性能优化策略，通过键值分离策略设计 vTree 结构，并提出层内归并与消极的层间合并相结合的方法，以及范围查询优化合并的策略，从而优化系统的范围查询性能，在 LSM-Tree 和 vTree 采用不同的压缩结构，以实现系统读写性能的提升；实验结果表明，与 RocksDB 相比读性能提升 30%，与 RocksDB-vTree 相比范围查询性能提升 10%。

关键词：读性能；LSM-Tree；消极的层间合并；范围查询优化合并；范围查询

Read and Write Performance Optimization of Storage System Based on LSM-tree Key Value

CHENG Haojin, HU Naiping
(College of Information Science and Technology, Qingdao University of Science & Technology, Qingdao 266061, China)

Abstract: In a write-intensive work environment, log-structured-merge (LSM-Tree) has gradually become a mainstream storage system, there are problems in LSM-tree such as slow read operation speed, high cost of write operation, and low efficiency of range query operation, etc. In view of these problems, this paper presents a study to improve the performance of LSM-tree, and optimize the read and write performance of key-value storage system based on LSM-tree, and proposes a read and write performance optimization strategy for LSM-tree-based key-value storage system, designs the vTree structure through the key-value separation strategy, and presents the combination of layer merging and negative inter layer merging, as well as the strategy of range query-optimized merging, so as to optimize the range query performance of the system, and adopt different compression structures in the LSM-tree and vTree to improve the system's read and write performance; the experimental results show that the read performance is improved by 30% compared with the RocksDB system, and the range query performance is improved by 10% compared with the RocksDB-vTree system.

Keywords: read performance; LSM-Tree; negative cascade consolidation; range query optimization merge; scope query

0 引言

随着大数据时代到来和数字经济的进一步发展，互联网与实体经济相融合，导致互联网产生的数据爆发性增长。海量的数据对数据存储提出新的要求，给传统的关系型数据库带来了严峻的挑战。由于数据量的规模不断增大，关系型数据库无法满足当今高读写并发的需求。为解决上述问题，非关系型数据库（NoSQL）应运而生[1]。其中基于 LSM-Tree（Log-structured-merge Tree）的键值存储系统，已广泛应用于网络购物、大数据分析和在线视频等场景。许多公司根据自己需求开发基于 LSM-Tree 的键值存储系统，如谷歌的 LevelDB 和 Facebook 的 RocksDB 等[2]。

日志结构合并树（LSM-Tree）作为键值存储系统被广泛用于处理大量写密集型工作，如 RocksDB、HBase 和 Cassandra 等[3]。LSM-Tree 是多层次化数据结构，其每一层由多个 SSTable 组成。LSM-Tree 的主要思想是通过转换写入方式，将小规模的随机写入转换为大规模的顺序写入，进而提高键值对写入效率[4]。常见的 LSM-Tree 的键值存储，使用压缩将低级别一个或多个 SSTable 合并到高级别的 SSTable。

但是，传统的压缩将导致严重的写放大。写放大消耗大部分的磁盘带宽，并严重降低系统的吞吐量，甚至导致写入暂停。目前已经提出了一系列方法来提升 LSM-Tree 的系统性能。如 Cassandra、PebblesDB 和 HBase 提出的层内

收稿日期:2023-05-11； **修回日期**:2023-06-15。
作者简介:程浩津(1997-),男,硕士研究生。
通讯简介:胡乃平(1968-),男,博士,教授。
引用格式:程浩津，胡乃平.基于 LSM-Tree 的键值存储系统的读写性能优化[J].计算机测量与控制,2024,32(6):262-268,275.

归并（Tiered Compaction）的策略，放松每层键值对的有序性，但将导致额外的读成本和空间开销[5]。WiscKey 通过 KV 分离的策略虽然减少了写放大，但读操作将额外地增加一次 I/O，并且日志结构需要增加垃圾回收[6]。DiffKV 将键值对按照大小分为 3 类采用不同的管理方式，该方法丢掉了 KV 分离的优势，并产生额外的空间开销[7]。UniKV 引入哈希索引的方法提高读取效率，同时放松存储热数据的有序性并提高冷数据有序性，由于 UniKV 放弃了布隆过滤器，在冷数据的读取效率不高，同时造成了严重的空间放大[8]。Spooky 为应对空间放大和写放大放松了最高层的有序性，将最高层键值数据划分为几个大小相等的文件[9]。Design Space 提出了 LSM-Compaction 的设计空间，并根据关键性能指标评估最先进的 Compaction 策略[10]。PM Hash Indexes（Persistent Memory Hash Indexes）使用 PM（Persistent Memory）硬件对持久哈希表进行全面的评估，并探讨硬件配置（如 PM 带宽和 CPU 指令）如何影响基于 PM 的哈希表性能[11]。同时 PM（Persistent Memory）重新设计基于 LSM-Tree 的 OLTP（Online Transaction Processing）存储引擎，实现快速地写入而不需要同步日志开销，并实现了接近即时的恢复时间[12]。Zen 主要方案包括元数据增强的元组缓存（Met-Cache），无日志的持久事务和轻量级 NVM 空间管理[13]。RocksDB-vTree 为降低写放大采用键值分离的架构，在 vTree 实现了部分值的有序性，但该方法在 vTree 中形成多个 vTable 组导致 vTree 的有序性较低[14]。上述解决方案无论是牺牲写性能或读性能还是通过结合最新的硬件 PM，均为提升系统性能方面做出了尝试。本文的解决方案参考了 RocksDB-vTree 的设计思想，针对 RocksDB-vTree 在 vTree 中形成多个 vTable 组，对系统范围查询性能有所限制的问题，本文提出层内归并与消极的层间合并相结合的方法，以及范围查询优化合并提升不同 vTable 组的有序性，进而克服 RocksDB-vTree 中 vTree 存在多个 vTable 组降低系统范围查询性能的问题。

本文为提升系统读性能、范围查询性能和写性能在 RocksDB-vTree 的基础上设计了 VtreeKV 架构，VtreeKV 提出层内归并与消极的层间合并相结合以及范围查询优化合并，通过提升 vTree 数据排序程度产生有限的写开销。与业界广泛应用的存储系统 RocksDB 和 RocksDB-vTree 进行对比实验，结果表明 VtreeKV 有效提升范围查询性能同时表现出优秀的综合性能。

1　LSM-Tree 的架构及原理

1.1　基于 LSM-Tree 的键值存储系统

LSM-Tree 是一种多层次存储结构，主要是为了写密集型工作负载而设计的一种架构，现在主流的存储结构，如 RocksDB，其架构如图 1 所示，在 LSM-Tree 分为内存部分和外存部分，内存部分由 MemTable 和 ImmuTable 组成。其中外存为多层次架构由 L_0 层，……，L_n 层组成，每层数据呈指数型增长，如 L_{i+1}层的数据量是 L_i 层的 10 倍，其中 L_i 层（$i>0$）是由多个排序字符串表（SSTable，sorted-string-table）组成，并且每个 SSTable 中键值数据均为完全有序，在 L_0 层为提高刷新效率，不同 SSTable 之间存在键值对重叠[15]。

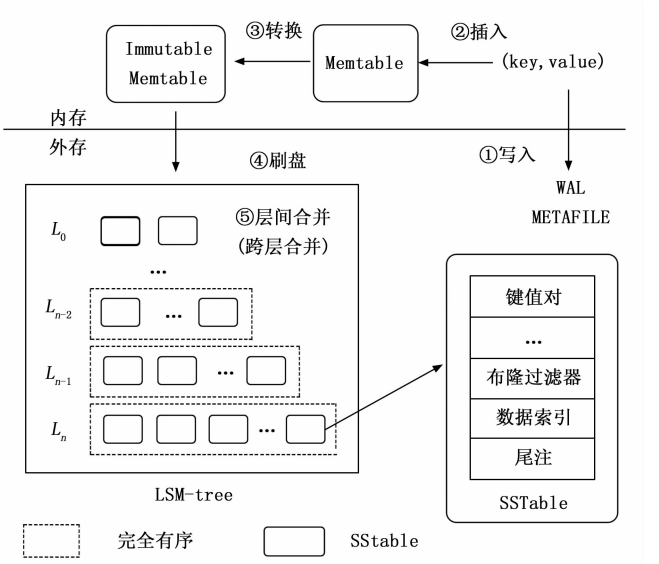

图 1　LSM-Tree 的系统架构

当有键值数据写入时，键值对会首先写入日志文件（WAL，write-ahead-log），保证系统崩溃时可以恢复数据，METAFILE 是一个元数据文件，记录 SSTable 的压缩操作信息用于系统崩溃时数据恢复。然后将数据写入 MemTable，当 MemTable 的数据达到阈值时，MemTable 转换为 ImmuTable MemTable 同时生成新的 MemTable 用于接受新写入的键值数据。随后 ImmuTable MemTable 在后台将键值对，组织成一个新的 SSTable 追加写到 L_0 层。SSTable 中包含排序的键值对和一些元数据（如数据索引和布隆过滤器等）。对于 LSM-Tree 上的两个相邻级别，当 L_i 层中 SSTables 的容量超过其阈值时触发层间合并（Leveled Compaction）。层间合并的操作流程，首先在 L_i 层选择 SSTable（对于 L_0 层，选择所有存在 key 范围重叠的 SSTable），然后在 L_{i+1} 层挑选存在 key 范围重复的 SSTable。将 L_{i+1}层和 L_i 层存在重复的 SSTables 读取到内存，进行排序合并生成新的 SSTables 写回 L_{i+1}层。层间合并虽然消除了大量的冗余数据，保证了层数据的有序性，但是压缩需要大量的数据重写导致严重的写放大[16]。

当读操作时，首先在 LSM-Tree 的内存中搜索键值对。如果键值对不存在，便在 LSM-Tree 的每个层进行搜索，从最低级别 L_0 层到最高级别 L_n 层。在每个级别进行搜索时，通过布隆过滤器（Bloom Filter）确定 SSTable 是否存在[17]。

当更新操作时，同样采用向 MemTable 写入新的键值数据，不需要找到相应键值对的位置进行更新。删除操作

只需要写入一个键值对的删除标记。当点查询时，首先将在 MemTable 进行搜索，若未发现将会到 ImmuTable MemTable 中继续查找，然后依次查找 L_0 层、L_1 层、……、L_n 层直到找到目标键值对。

1.2 基于 LSM-Tree 键值分离的存储系统

传统 LSM-Tree 采用层间合并，造成了严重的写放大同时造成了大量的 I/O 开销。因此，RocksDB-vTree 等均采用键值分离的架构，将 key 和 value 解耦后单独存储，其中 key 和 value 的地址存储到 LSM-Tree 中，value 被存储到日志结构，其架构如图 2 所示。由于键和值地址的数量级远远小于值的数量级，所以在 LSM-Tree 中数据量显著减少，I/O 开销和压缩开销得到进一步缓解，但与传统的 LSM-Tree 相比将会增加空间放大[18]。

由于值存储到 vTree 结构中，当进行读操作时会额外增加一次读操作。为提高键的范围查询性能，在 vTree 结构中对数据进行合并排序，以形成多个有序 vTable 排序组。同时，将 LSM-Tree 和 vTree 的压缩相耦合，以减少额外的写开销。为了降低写放大，vTree 设计了延迟归并方法。RocksDB-vTree 在 vTree 使用层内归并形成了多个有序的 vTable 组，但不同 vTable 组之间存在键范围重叠，所以范围查询性能有较大的影响[19]。

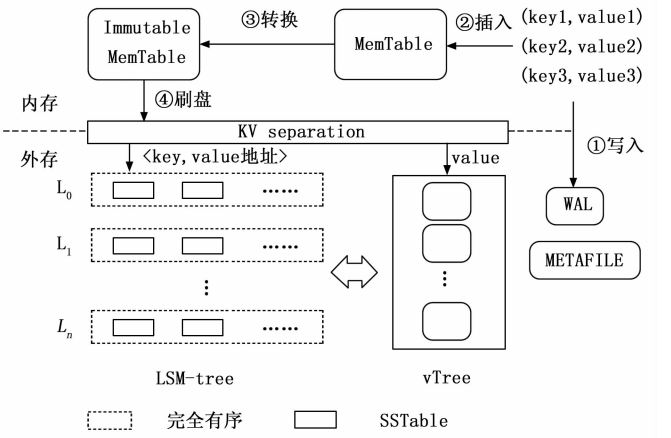

图 2 LSM-Tree 键值分离存储结构

2 系统设计

2.1 系统概览

为实现 LSM-Tree 读写性能提升做出新的尝试，本文将提出名为 VtreeKV 的架构，如图 3 所示。VtreeKV 采用在键值分离的架构，键和值地址存储到 LSM-Tree 而将值存储到 vTree。在 vTree 提出层内归并与消极的层间合并相结合的数据合并方法，以及范围查询优化合并提升数据的有序性。

2.2 数据读写流程概述

如图 3 所示，与传统的 LSM-Tree 类似，当执行写操作时，键值对首先写入日志(WAL)，然后写入 MemTable，当 MemTable 超过其阈值时，将 MemTable 转换成 ImmuTable，并生成新的 MemTable 接受新写入的键值对，在刷盘之前进行键值分离，将值在后台组织成 vTable 存储到 vTree，并将键和值地址在后台组织成 SSTable 写入 LSM-Tree。值地址记录了值在的 vTable 的 ID、vTable 的偏移量和值的大小。当执行查询操作时，首先查询内存中 Memtable 是否存在相应键数据，若不存在便依次遍历 LSM-Tree 的每一层，若发现查询的键数据便会根据其值索引，在 vTree 中读取相应的值。VtreeKV 为降低写放大采用了键值分离架构，但键值分离架构会导致范围查询性能降低，因此本文提出了层内归并与消极的层间合并相结合的数据合并方法以及范围查询优化合并的方法，对 vTree 中数据的有序性进行优化。

由于 LSM-Tree 和 vTree 结构相耦合，当 LSM-Tree 发生压缩便会触发 vTree 结构进行合并操作，当 LSM-Tree 的 L_i 层达到压缩阈值，L_i 层的 SSTable 与 L_{i+1} 层的存在范围重叠的 SSTable 进行合并排序，并且 LSM-Tree 的压缩将会触发 vTree，在 vL_i 层相应索引的 vTable 进行合并，SSTable 对应索引的 vTable 进行合并排序后，在 L_{i+1} 层形成多个有序的 vTable 组。VtreeKV 参考 RocksDB-vTree 引

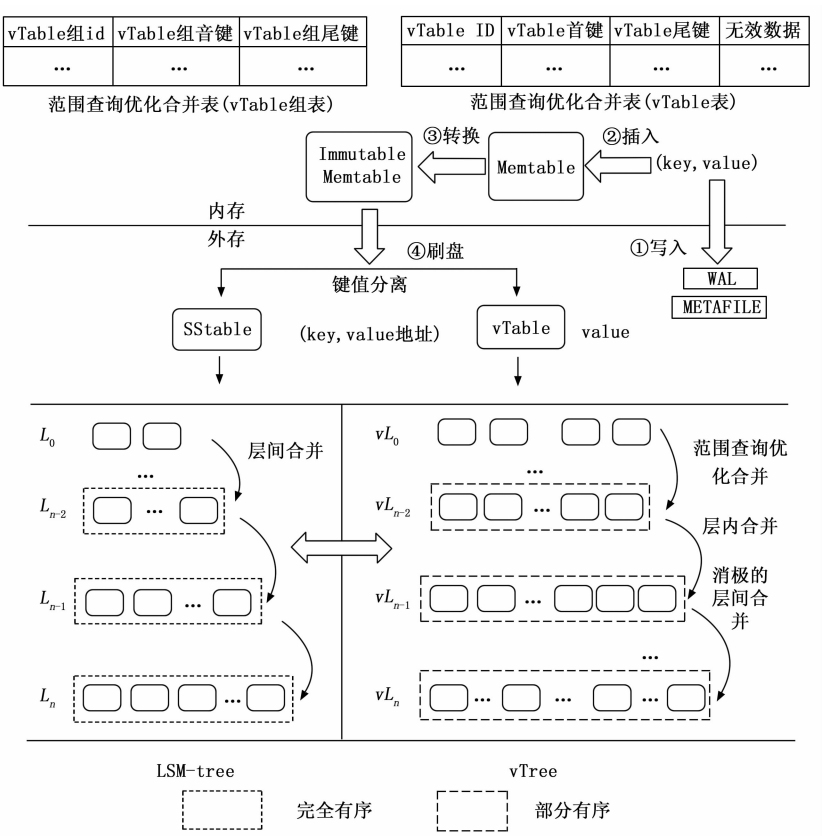

图 3 VtreeKV 系统架构

入延迟归并机制，L_0～L_{n-2}层压缩将不会触发 vL_0～vL_{n-2}层合并。当 vL_{n-2}～vL_{n-1}层合并，将会把 vL_0～vL_{n-2}层的数据合并到 vL_{n-2}层。

本文为提升系统范围查询性能，在 vTree 使用层内归并、范围查询优化合并和消极的层间合并。与层间合并相比，层内归并与消极的层间合并相结合减少了压缩数据量降低了写放大。

2.3　层内归并与消极的层间合并相结合的数据合并方法

在 LSM-Tree 执行层间合并，vTree 将进行层内归并，虽然有效提升 vTree 有序性，但参与层内归并的 vTable 只合并排序了 vL_i 层的数据，对 L_{i+1}层的数据没有进行合并排序操作，本文提出层内归并与消极的层间合并相结合的方法。消极的层间合并与传统的层间合并相比，在 vL_{i+1}层挑选超过压缩阈值（Vct）vTable 参与消极的层间合并，与层间合并相比减少了 vL_{i+1}层操作数据量。由于 LSM-vtree 和 vTree 架构在高级别层的数据是低级别层数据的 10 倍，因此 90％的数据将会集中在 L_{n-1}层和 L_n 层，所以消极的层间合并将会应用在 vTree 的 L_{n-1}层和 L_n 层。

层内归并与消极的层间合并相结合的压缩算法的流程如图 4 所示，图中阴影部分为本次操作涉及的数据范围。LSM-Tree 的键从 L_i 层合并到 L_{i+1}层时触发 vTree 合并操作。

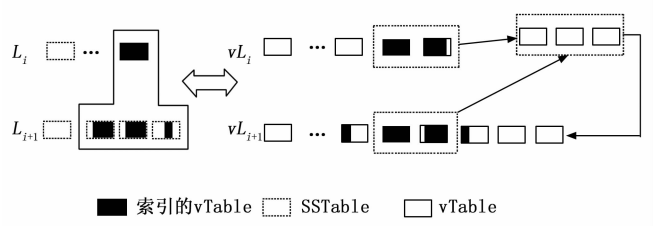

图 4　层内归并与消极的层间合并相结合

L_{i+1}层第 k 个 SSTable 在第 t 个 vTable 索引的值大小，记为 V_{kt}。其中 k 的范围为（m_s，m_e），其中 m_s 和 m_e 分别为 L_{i+1}层参与压缩的 SSTable 的上限和下限，而 t 的范围为（n_{ks}，n_{ke}），其中 n_{ks} 和 n_{ke} 分别为第 k 个 SSTable 索引的 vTable 上限和下限。而压缩阈值记为 V_{ct}。若 vL_{i+1}层的 vTable 需要满足公式（1），则与 SSTable 在 vL_i 层索引的 vTable 一起参加压缩追加 vL_{i+1}层。

$$(V_{kt}/8M) > V_{ct} \qquad (1)$$

式中，V_{kt} 为第 k 个 SSTable 在第 t 个 vTable 索引的值大小，8 M 为 vTable 的大小，V_{ct}为压缩阈值。

因此，压缩之前，L_i 层和 L_{i+1} 层索引的值分别分布在 vL_i 层和 vL_{i+1} 层的多个相互重叠的 vTable 组，压缩后将 L_{i+1} 层超过 V_{ct} 的 vTable 与 L_i 层的 vTable 进行排序后，生成新的 vTable 重新写入 L_{i+1} 层。

层内归并与消极的层间合并相结合提升 vTree 中数据的有序性，并且产生了有限的合并开销。主要原因有：首先，vTree 中的每个级别都有多个有序的 vTable 组。其次，vL_{i+1} 层仅选择部分 vTable 参与消极的层间合并。

2.4　范围查询优化合并

系统为继续优化范围查询性能提出范围查询优化合并的方法。范围查询优化合并来调整 vTree 数据的有序性，从而提升系统范围查询性能。层内归并与消极的层间合并相结合的方法，在 vL_i 层和 vL_{i+1} 层部分 vTable，被重新组织并附加到较高级别 vL_{i+1} 层上，但 vL_{i+1} 层会造成有序的 vTable 组过多。因此，不同 vTable 组之间存在键范围重叠，针对该问题本节提出范围查询优化合并。范围查询优化合并的操作流程如下：

1）根据图 5 所示的算法计算 vTable 组的合并范围（不同 vTable 组之间的数据合并排序范围）。

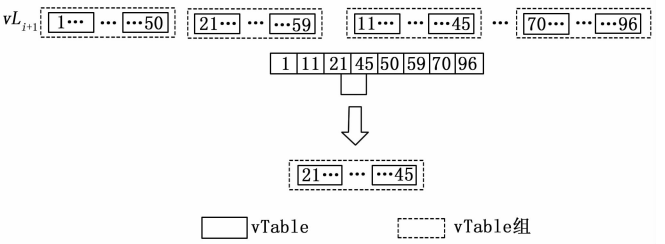

图 5　确定 vTable 组的合并范围

2）根据图 6 所示的算法，在 vTable 组的合并范围中选择大于优化阈值（max_run）的 vTable。

范围查询优化合并首先需要确定 vTable 组的合并范围，其算法操作流程如图 5 所示，在不同 vTable 组中确定 vTable 组的合并范围的操作步骤如下：

1）在确定 vTable 组的合并范围之前，首先排除那些不存在键范围重叠的 vTable 组，比如 vTable 组［70，96］。这可以通过内存哈希表（vTable 组表）中每个 vTable 组的首键和尾键来实现。使用公式（2）依次计算每个 vTable 组的首尾键，如果某个 vTable 组的首键大于所有 vTable 组的尾键，则该 vTable 组将被排除在范围查询优化合并之外。然后，利用公式（3）在剩余的 vTable 组中进行计算，如果

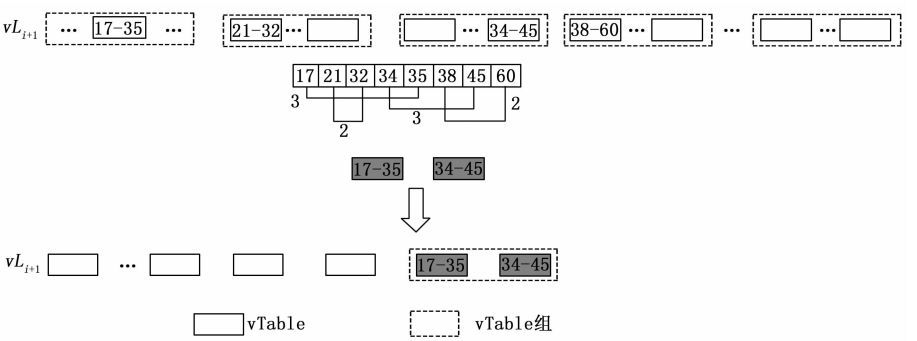

图 6　选择 vTable

某个 vTable 组的尾键小于所有 vTable 组的首键，则该 vTable 组将不参与范围查询优化合并。

$$sk_i > max(ek_1,\ldots,ek_n) \tag{2}$$

式中，sk_i 为任意一个 vTable 组的首键，ek_1 和 ek_n 为不同 vTable 组的尾键。

$$ek_j < min(sk_1,\ldots,sk_n) \tag{3}$$

式中，ek_i 为任意一个 vTable 组的尾键；sk_1 和 sk_n 为不同 vTable 组的首键。

2）根据公式（2）和（3），排除不参与范围查询优化合并的 vTable 组后，在剩余的 vTable 组中选择最大的首键作为合并范围的首键。vTable 组的合并范围首键的选择满足公式（4），因此选择键 21 作为合并范围的首键。

$$MaxStart = max(sk_1,sk_2,\ldots,sk_n) \tag{4}$$

式中，sk_1，sk_2 等为不同 vTable 组的首键；$MaxStart$ 为 vTable 组中首键的最大值。

3）从剩余的 vTable 组中选择最小的尾键作为合并范围的尾键。选择的尾键需满足公式（5），因此选择键 45 作为合并范围的尾键，这样确定范围［21，45］构成 vTable 组的合并范围。

$$MinEnd = min(ek_1,ek_2,\ldots,ek_n) \tag{5}$$

式中，ek_1，ek_2 等为不同 vTable 组的尾键；$MinEnd$ 为 vTable 组中尾键最小值。

由上文可以确定 vTable 组的合并范围为（$MaxStart$，$MinEnd$）。基于不同 vTable 组之间键范围的重叠情况确定 vTable 组的合并范围。由于不同 vTable 组的键范围存在差异，但 vTable 组内的键是有序的，所以 vTable 组的合并范围尽可能包含 vTable 组重叠部分。vTable 组的合并范围首键的选择：在有序的 vTable 组中，选择每个 vTable 组中首键的最大值作为 vTable 组的合并范围的首键。vTable 组的合并范围尾键的选择：在有序的 vTable 组中，选择每个 vTable 组中尾键的最小值作为 vTable 组的合并范围的尾键。通过这种方式，可以确保 vTable 组的合并范围，尽可能涵盖键范围重叠相关的 vTable 组，同时提高 vTable 组的合并范围的有序性。这种设计使得 vTable 的合并操作能够有效地处理不同 vTable 组之间的键范围重叠。

范围查询优化合并，在 vTable 组的合并范围中选择 vTable 的算法操作流程如图 6 所示。在 vTable 组的合并范围中，需计算每个 vTable 的重叠数量（end _ num-begin _ num），并查找是否存在超过优化阈值（max _ run）的 vTable。若存在超过优化阈值的 vTable，则为该 vTable 添加范围查询优化标记。对于带有优化标记的 vTable，它们将参加下一次 vTree 的合并操作。因此，在 vTable 合并范围内，选择 vTable 的操作步骤如下所示。

1）在 vTable 组的合并范围中，通过比较 vTable 首键和尾键，使用 vTable 组表和 vTable 表来计算不同 vTable 键范围的重叠数量。图 6 以其中一个 vTable 为例，其范围是［17，35］。

$$\max(sk_1,sk_2,\ldots,sk_m) < ek_i \tag{6}$$

式中，sk_1，sk_2 等为不同 vTable 组的首键；ek_i 为任意一个 vTable 组的尾键。

2）通过公式（6），计算 vTable 中首键小于 35 的数量为 3 个（即 end _ num）。然后，通过公式（7），计算 vTable 中尾键小于 17 的数量为 0 个（即 begin _ num）。最后，通过计算 end _ num-begin _ num，得出 vTable 中键范围［17，35］重叠数量为 3 个。其包含自身在内范围重叠 vTable 为［17，35］、［21，32］和［34，45］。

$$\max(ek_1,ek_2,\ldots,ek_m) < sk_i \tag{7}$$

式中，ek_1，ek_2 等为不同 vTable 组的尾键；sk_i 是任意一个 vTable 组的首键。

3）最后，对于键范围重叠数量大于 max _ run 的 vTable（即满足公式（8）），为添加一个扫描优化标记。

$$(\text{end\_num} - \text{begin\_num}) > \text{max\_run} \tag{8}$$

式中，max _ run 是优化阈值。

范围查询优化合并将优化标记保存到 METAFILE 文件中，现有的系统使用该文件跟踪记录每次压缩后 KV 对的版本，所以存储的开销可以忽略不计。

范围查询优化合并将会在内存维护两个哈希表，vTable 组表用于记录 vTable 组的 id，以及 vTable 组的首键和尾键每个占 4 B，总共占用 12 B 内存。vTable 表记录 vTable 的 id，以及 vTable 的首键和尾键每个占 4 B，总共占用 12 B 内存。

层内归并与消极的层间合并相结合的方法提升 vTree 数据的有序性，但不同 vTable 组之间存在键范围重叠，对于系统的进行范围查询是不友好的。而范围查询优化合并，对 vTable 组中键范围重复过高的 vTable 进行合并，vTree 的有序性得到提高，并且范围查询优化合并产生了有限的合并开销。

2.5 空间回收及崩溃恢复

VtreeKV 参考 Rocks-vTree 和 RocksDB 的设计思想，在空间回收和系统崩溃恢复并未进行优化，因此本节只简单叙述。由于 vTree 需要将 vTable 中索引失效的值数据清除，因此系统需要垃圾回收（Garbage Collection）来回收额外的值空间（在 LSM-Tree 中无效数据通过压缩回收）。

VtreeKV 在内存建立一个哈希表，用于跟踪记录每一个 vTable 中无效值的数量和 vTable 的 id，其占用 8 B 内存。当每次 vTable 参与压缩时，系统会统计 vTable 中 value 数量，并更新哈希表中旧的 vTable 中无效值的数量。当 vTable 中无效值达到空间回收的阈值时，vTree 将不会马上回收 vTable 而是为每一个 vTable 添加一个 GC 标记，回收将会延迟到下一次 LSM-Tree 发生压缩触发 vTree 的合并。具体来说，在 vTable 中有效的 value 将会参加下一次 LSM-Tree 压缩而触发 vTree 的合并。

哈希表中哈希值的更新，在系统发生压缩期间完成，并且每个 vTable 哈希值仅占用几个字节，因此造成的内存开销非常小，同时空间回收减少了数据在磁盘上的频繁移

动，极大地提升了系统的性能。

系统为了保证崩溃恢复后数据的一致性，VtreeKV 在键值对写入 MemTable 之前写入日志（WAL）。此外，VtreeKV 的哈希表记录每一个 vTable 中无效数据量，提供了崩溃的一致性。由于哈希表在压缩后更新，VtreeKV 在压缩后将更新信息附加到 METAFILE，因此当 VtreeKV 从崩溃中恢复时可以保证数据的完整性。

3　实验结果与分析

实验所需配置如下：处理器为：Intel（R）Xeon（R）Gold 6240R CPU @ 2.40 GHz，内存为：64 GB，操作系统：Ubuntu18.04LTS，600 GB 磁盘。

由于 RocksDB 被广泛地使用，本文在 RocksDB 的基础上并参照 WiscKey 的描述和 RocksDB-vTree 思想实现了 VtreeKV。由于 VtreeKV 参照了 RocksDB-vTree 思想，因此该架构将会在实验中进行对比分析。为了保证实验数据的准确性，所有的实验对象均采用相同实验配置。MemTable 默认设置为 8 MB、vTable 默认设置为 8 MB。各存储系统均采用 16 线程，键值对中 key 设置 16 bytes，value 设置 1 kB，所有工作负载的键值数据遵守 zipfan 分布，其常数为 0.9[20]。

3.1　基本性能测试

首先，我们测试 RocksDB、RocksDB-vTree 和 VtreeKV 分别在加载、点查询、更新和范围查询操作的性能。在系统中随机加载 100 GB 的键值数据，键值数据保证不会出现 key 重复。然后进行如下操作：

1）点查询（Read）操作：查询 10 GB 的键值数据。

2）更新（Update）操作：在随机加载 100 GB 的基础上更新 100 GB 的键值数据。

3）范围查询（Scan）操作：范围查询 10 GB 的键值数据。

不同存储系统进行加载、点查询、更新操作和范围查询操作的归一化吞吐量如图 7 所示。VtreeKV 在加载阶段的吞吐量是 RocksDB 的 3.2 倍，并且吞吐量达到了 RocksDB-vTree 的 98%，由于 VtreeKV 在加载阶段并未做出优化，所以加载吞吐量与 RocksDB-vTree 相比差别不大。VtreeKV 使用层内归并与消极的层间合并相结合与 RocksDB-vTree 相比增加了额外的合并开销，所以 VtreeKV 更新吞吐量比 RocksDB-vTree 的吞吐量略低。但 VtreeKV 在更新阶段的吞吐量是 RocksDB 的 3 倍并且吞吐量达到了 RocksDB-vTree 的 92%。由于 VtreeKV 与 RocksDB-vTree 相比在 vTree 的使用层内归并与消极的层间合并相结合提升了数据的有序性，并且在 vTree 中使用范围查询优化合并。在范围查询阶段，VtreeKV 吞吐量达到了 RocksDB 的 97%，RocksDB-vTree 的吞吐量达到了 RocksDB 的 94%，VtreeKV 在范围查询操作的吞吐量较 RocksDB-vTree 略有提升。VtreeKV 与 RocksDB-vTree 均没有在点查询操作进行优化，因此，VtreeKV 和 RocksDB-vTree 的吞吐量相差不大，但吞吐量是 RocksDB 的 1.3 倍左右。

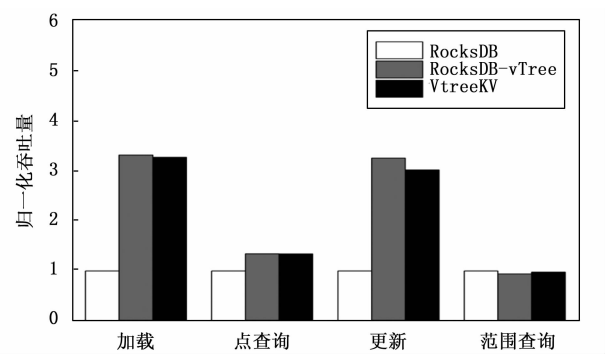

图 7　基本性能测试

3.2　不同范围查询长度性能对比

由于本文在 RocksDB-vTree 基础上优化范围查询性能设计了 VtreeKV，本节将比较 RocksDB-vTree 和 VtreeKV 在不同范围查询长度下的范围查询性能。每个系统加载 100 GB 的键值数据后执行范围查询，每次查询长度分别为 20，100 和 1 000 个键值对，并且查询长度需要涵盖 10 GB 的键值数据。RocksDB-vTree 和 VtreeKV 在不同查询长度下的归一化吞吐量如图 8 所示。

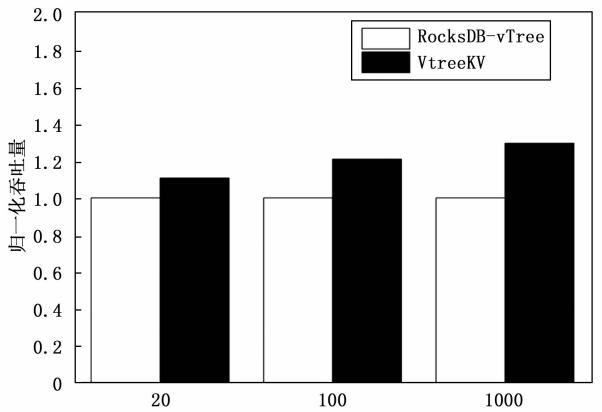

图 8　不同范围查询长度性能对比

范围查询长度为 20 个键值对时，VtreeKV 相较 RocksDB-vTree 范围查询性能提升 10%，范围查询长度为 100 个键值对时，VtreeKV 相较 RocksDB-vTree 范围查询性能提升 21%，范围查询长度为 1 000 个键值对时，VtreeKV 相较 RocksDB-vTree 范围查询性能提升 29%。随着范围查询长度的增大，VtreeKV 相较于 RocksDB-vTree 表现出不断提升的范围查询性能。这主要是因为在 vTree 中，value 的有序性成为影响系统性能的主要因素。VtreeKV 提出了层内归并与消极的层间合并相结合的方法以及范围查询优化合并的方法，优化 vTree 中值的有序性。随着范围查询长度的增加，VtreeKV 中 vTree 的值更倾向于顺序读取。此外，VtreeKV 和 RocksDB-vTree 在键和值定位开销方面差别不大。因此随着范围查询长度的不断增加，VtreeKV 相对于 RocksDB-vTree 表现出的范围查询性能更为显著。

3.3 YCSB 工作负载性能

YCSB（Yahoo! Cloud Serving Benchmark）基准测试是工业界评测键值存储系统的标准。因此，我们设计如表 1 所示的 5 种工作负载进行测试。本实验将会比较 RocksDB、RocksDB-vTree 和 VtreeKV 在 5 种负载的性能。每个工作负载提前加载 400 000 的键值数据，各个系统独立地完成工作负载，图 9 显示了各系统在不同工作负载下的归一化吞吐量对比。

表 1 YCSB 工作负载

工作负载	操作组成
A	50%点查询,50%更新
B	5%更新,95%点查询
C	100%点查询
D	5%插入,95%点查询
E	5%更新,95%范围查询

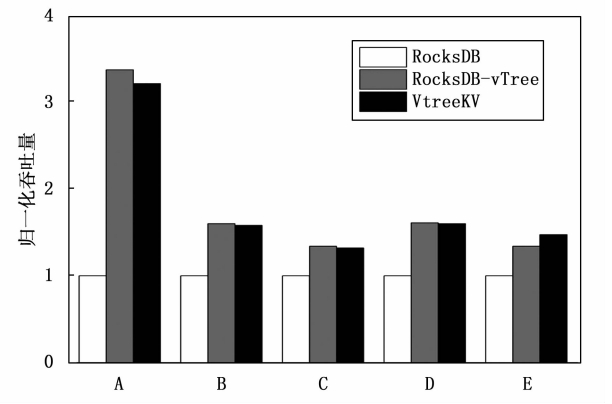

图 9 YCSB 工作负载性能对比

在点查询和更新混合的工作负载 A，同 RocksDB-vTree 相比 VtreeKV 使用了消极的层间合并造成了额外的写开销，但 VtreeKV 吞吐量达到 RocksDB-vTree 的 95%，并且 VtreeKV 吞吐量 RocksDB 的 3.2 倍。工作负载 B、工作负载 C 和工作负载 D 均是以点查询为主的操作，同 RocksDB-vTree 相比 VtreeKV 没有在点查询做出优化，因此两者在工作负载 B 和工作负载 D 差别不大，但与 RocksDB 相比均有较大提升。在工作负载 E 以范围查询为主的工作负载，VtreeKV 吞吐量是 RocksDB-vTree 的 1.1 倍，并且吞吐量 RocksDB 的 1.4 倍，在 3.2 节进行了详细介绍，因此不再重复叙述。通过上述负载操作说明 VtreeKV 除了在范围查询性能表现出色，同时 VtreeKV 还拥有出色的综合性能。

4 结束语

本文基于 LSM-Tree 的键值存储系统，为改善读写性能提出层内归并与消极的层间合并相结合的方法，以及范围查询优化合并，其目的是提升数据的范围查询性能。同时设计 VtreeKV 架构，其采用键值分离的策略并对 vTree 的有序性进行了优化。上述实验结果显示，本文提出的 VtreeKV 与 RocksDB 和 RocksDB-vTree 相比，不但范围查询性能方面表现优秀，而且在综合性能方面也取得了不错的效果。

参考文献：

[1] LAKSHMAN A, MALIK P, et al. Cassandra: a decentralized structured storage system [J]. ACM SIGOPS Operating Systems Review, 2010, 44 (2): 35 - 40.

[2] LU L, PILLAI T S, GOPALAKRISHNAN H, et al. Wisckey: Separating keys from values in ssd-conscious storage [J]. ACM Transactions on Storage (TOS), 2017, 13 (1): 1 - 28.

[3] RAJU P, KADEKODI R, CHIDAMBARAM V, et al. Pebblesdb: building key-value stores using fragmented log-structured merge trees [C] // USA: Proceedings of the 26th Symposium on Operating Systems Principles. Shanghai, China. New York: Association for Computing Machinery, 2017: 497 - 514.

[4] YAO T, WAN J, HUANG P, et al. A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores [C] // International Conference on Massive Storage Systems and Technology. USA, 2017: 1 - 13.

[5] WU F, YANG M H, ZHANG B, et al. AC-key: Adaptive caching for LSM-based key-value stores [C] // USENIX Annual Technical Conference. USA, 2020: 603 - 615.

[6] LUO C, CAREY M J, et al. LSM-based storage techniques: a survey [J]. The VLDB Journal, 2020, 29 (1): 393 - 418.

[7] LI Y, LIU Z, LEE P P C, et al. Differentiated key-value storage management for balanced I/O performance [C] // USENIX Annual Technical Conference. USA, 2021: 673 - 687.

[8] ZHANG Q, LI Y, LEE P P C, et al. The design and implementation of UniKV for mixed key-value storage workloads [J]. IEEE Transactions on Knowledge and Data Engineering, 2023 (11): 11935 - 11949.

[9] DAYAN N, WEISS T, DASHEVSKY S, et al. Spooky: granulating LSM-Tree compactions correctly [J]. Proceedings of the VLDB Endowment, 2022, 15 (11): 3071 - 3084.

[10] SARKAR S, STARATZIS D, et al. Constructing and analyzing the LSM compaction design space [J]. Proceedings of the VLDB Endowment, 2021, 14 (11) : 2216 - 2229.

[11] HU D, CHEN Z, WU J, et al. Persistent memory hash indexes: an experimental evaluation [J]. Proceedings of the VLDB Endowment, 2021, 14 (5): 835 - 848.

[12] YAN B, CHENG X, JIANG B, et al. Revisiting the design of LSM-Tree based OLTP storage engine with persistent memory [J]. Proceedings of the VLDB Endowment, 2021, 14 (10): 1872 - 1885.

[13] LIU G, CHEN L, CHEN S, et al. Zen: a high-throughput log-free OLTP engine for non-volatile main memory [J]. Proceedings of the VLDB Endowment, 2021, 14 (5): 835 - 848.

[14] 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.

（下转第 275 页）