# 面向云边场景的读写均衡键值存储系统

郑宜湉[1]，张余豪[2]，霍志杰[1]，舒继武[2]
1. 厦门大学信息学院，福建 厦门 361102；
2. 清华大学计算机科学与技术系，北京 100084

**摘要**

基于LSM–tree的键值存储因其高效的数据存储机制，成为云端和边端数据管理的理想选择。但LSM–tree采用的Leveled压实策略具有较高的写放大率，会对前台写性能造成明显的负面影响。如何在降低写放大、进一步提升写性能的同时不牺牲读性能，成为优化LSM–tree面临的一大挑战。针对以上问题，提出一种新型的键值存储系统LooseKV，该系统利用Tiered压实策略显著降低写放大，同时引入基于跳表的内存索引，结合轻量级的索引更新策略、与分层结构上的迭代器集成，有效改善了Tiered策略存在的读性能差的问题。实验表明， LooseKV的随机写入吞吐量为LevelDB的1.18~2.28倍，随机读取性能为LevelDB的1.01~1.26倍，顺序读取性能低于LevelDB，接近PebblesDB。

**关键词**

键值存储；日志结构合并树；写放大；读性能优化

中图分类号:TP392 文献标志码：A doi:10.11959/j.issn.2096–0271.2025038

# *A read-write balanced key-value store for cloud-edge computing environments*

**ZHENG Yitian[1], ZHANG Yuhao[2], HUO Zhijie[1], SHU Jiwu[2]**
1. School of Informatics, Xiamen University, Xiamen 361102, China
2. Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

***Abstract***

The LSM-tree based key-value store has become an ideal choice for cloud and edge data management due to its efficient data storage mechanism. However, the leveled compaction strategy used in LSM-tree suffered from high write amplification, which negatively impacted foreground write performance. Reducing write amplification and further improving write performance without sacrificing read performance become a major challenge in optimizing LSM-tree. To address this issue, we proposed a novel key-value store system, LooseKV, which leveraged the tiered compaction strategy to significantly reduce write amplification. Additionally, LooseKV introduced a skip-list-based memory index, combined with a lightweight index update strategy and integration with iterators in the tiered structure, effectively improving the read performance issues associated with the tiered strategy. Experimental results demonstrated that LooseKV achieved 1.18 to 2.28 times higher random write throughput than LevelDB, 1.01 to 1.26 times better random read performance, and slightly lower sequential read performance compared to LevelDB, but comparable to PebblesDB.

***Key words***
key-value store, log-structured merge tree, write amplification, read performance optimization

## 0 引言

随着端侧设备实时生成数据量的急剧增长和对实时处理能力需求的不断提升，边缘计算将数据处理和存储从集中式数据中心移至网络边缘，以实现数据的本地化处理，降低时延并提升实时响应能力。在物联网、智能交通、视频监控等数据密集型场景中，边缘计算需要具备强大的处理能力和低时延响应能力，以应对大规模数据流的实时处理需求。这些任务要求边缘计算环境能够有效存储和管理大量由边缘设备生成的数据，并实时跟踪和更新数据的处理状态。在此基础上，云端负责集中管理和存储数据，提供更强大的计算和存储资源，支持数据的长时间保存和深度分析。然而，边缘设备的计算、存储和带宽资源有限，如何在云、边、端之间高效协同处理和存储数据，成为边缘计算面临的关键挑战。为减少数据读写性能低下对系统产生的不利影响，采用提供高吞吐量的键值存储作为数据管理的后端是一种合理的解决方案[1-4]。

持久性的键值存储是现代大规模存储基础设施的组成部分，在各种现代数据密集型应用中发挥着关键作用。日志结构合并树（log-structured merge tree，LSM-tree）作为主流的键值存储结构之一，在许多大型互联网公司的存储系统中被广泛应用。现有的基于LSM-tree开发的键值存储有：Google的BigTable[5]和LevelDB[6]，Facebook的HBase[7]和RocksDB[8]，阿里巴巴的X-Engine[9]等。本文主要针对基于LSM-tree的键值存储（LSM-tree based key-value store，LSM-KVS）展开研究和设计。

不同于B+树采用的就地更新策略，LSM-tree采用同日志一样的追加写形式，显著提高了数据库的写性能。但LSM-tree的非就地更新策略也存在弊端。旧记录没有被新记录覆盖，数据库中会存在同一个键的新旧多个版本，导致读性能下降以及空间浪费。因此，LSM-tree还需要单独的数据重组过程，称为压实（compaction）。压实操作清除已过时的键值对，维持键值对的部分有序性。然而，LSM-tree高频率的压实操作使部分有效数据被反复读出、写回，严重时会争用前台写工作带宽，引起写性能下降[10]。

改善LSM-KVS的写放大一直是学术界和工业界关注的优化热点[11]。BlockDB[12]将压实的粒度由有序字符串表（sorted string table，SST）文件调整为数据块，仅读取和重组参与压实的数据块，不重写无关的数据块，从而降低写放大；WiscKey[13]采用键值分离的思想，使压实只需要重写键，而不涉及值的清除工作，从而有效降低写放大；低层驱动压实（lower-level driven compaction，LDC）[14]将低层中位于某个较小范围内的有序数据子集作为触发点，从上层拉取对应键范围内的小数据集进行压实操作，通过减少压实涉及的数据量降低写放大；PebblesDB[15]采用一种名为“守卫”（guard）的机制来管理一个限定范围内的SST文件，通过实现更细粒度的压实和更宽松的键值顺序来降低写放大。然而，以上对写放大的优化策略都牺牲了点读和扫描的性能，无法实现读

写性能的平衡。

本文提出一种新型的键值存储系统LooseKV，采用对键值有序性更宽松的Tired压实策略，有效降低写放大，大幅提升写性能。在键值有序性不如LevelDB的情况下，为了维持与LevelDB相当的读性能，LooseKV集成了3个新设计：①跳表索引优化点读性能；②有序段映射优化索引更新过程；③跳表迭代器优化扫描性能。本文首先介绍LSM-tree的整体框架和相关操作，之后介绍LooseKV的整体框架和3个设计点，最后对LevelDB、PebblesDB、LooseKV的微基准测试（micro-benchmark）和宏基准测试（macro-benchmark）进行对比。

## 1 日志结构合并树

LSM-tree是一种用于存储数据的多层树形数据结构。如图1所示，LSM-tree的结构分为两部分，一部分为内存组件，包括一个接受写的Memtable和多个只读Memtable，Memtable采用跳表作为基本数据结构，确保每个键值对在插入其中后保持有序；LSM-tree的另一部分持久化到磁盘上，为分布在多个层的SST文件，越底部的层能容纳的键值对越多，相邻层之间的容量比默认为1∶10。磁盘上的Manifest文件保存层与SST文件间的映射关系以及SST文件的元数据信息。

### 1.1 读写流程

当用户向数据库中写入键值对时，键值对首先被插入有序结构Memtable中，当其大小超出阈值时，将转为只读Memtable，等待随后被刷写（flush）进入第0层。刷写操作将一个只读Memtable中的键值对按序追加写到一个全新的SST文件中，并将最终生成的SST文件持久化到分层结构的第0层。当第0层的容量超出阈值时，将触发压实操作：首先，选择并读取第0层的$n$个文件和第1层键范围与其相交的所有SST文件，并将其作为输入文件；随后，对这些SST文件包含的键值对进行去重、排序和合并操作，生成多个键范围不重叠的SST文件，持久化到分层结构的第1层；同样地，当第$k$层的容量超出阈值时，也会触发与第$k$+1层之间的压实操作。刷写操作和压实操作使LSM-tree第0层的SST文件之间可能存在键范围的重叠，而非第0层则是整层有序的SST文件，一定不存在键范围的重叠。若将多个完全排序的SST文件构成的组合称作有序段（sorted run），则LSM-tree第0层中的每个文件就是一个有序段，而非第0层的每一层是一个有序段。

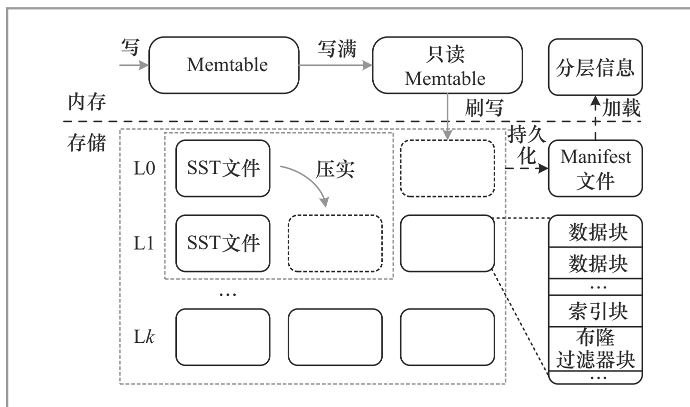



图1 LSM-tree框架

当需要从数据库中查找某个键值对时，首先依次查找Memtable和只读Memtable。如果没有从内存组件中查找到，则进入分层结构查找。在分层结构中查找时，需要从上层至下层逐层查找，直

到查找到所需键时才停止。在某一层查找时，首先通过分层结构信息获取该层包含的所有SST文件信息，并从中挑选出与所查键范围重叠的SST文件，接着在该SST文件中查找所需键；若没有找到，则进入下一层查找。

除了点读操作以外，LSM-tree还支持扫描操作。具体而言，数据库针对当前视图构造一个迭代器对象。该对象封装的是数据库内部全局的迭代器：先为每个Memtable构造迭代器，再为磁盘上的每个有序段构造迭代器，最后将这些迭代器加入迭代器堆。该迭代器堆以归并排序的方式正向或反向按序提供键值对。

### 1.2 压实策略

对Memtable的刷写操作（从内存到第0层）也可以认为是一种压实操作，其使用的压实策略为Tiered策略，而磁盘上采用的压实策略为Leveled策略。后文将介绍这两种压实策略的不同之处。

（1）Tiered策略

Tiered策略如图2所示。使用Tiered策略的层可以存在多个有序段。当第$k$层触发压实操作时，其上的所有有序段包含的文件都会被读入内存，排序、去重之后作为一个新的有序段加入第$k$+1层。

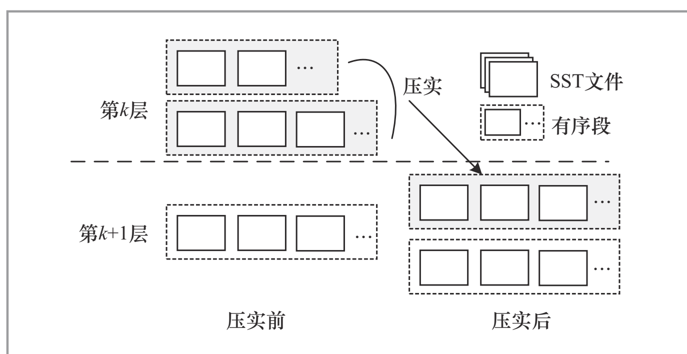



图2 Tiered策略

（2）Leveled策略

Leveled策略如图3所示。使用Leveled策略的层只可以存在一个有序段，即整层文件完全有序。当第$k$层触发压实操作时，第$k$层和第$k$+1层的有序段包含的部分文件会被读入内存，这些文件排序、去重之后会被写回第$k$+1层原来的有序段中，仍维持每层只存在一个有序段的限制。

显然，Tiered策略对数据库中键值对的有序性要求更宽松，但采用这种策略的层中包含的有序段数量较多。在最坏情况下，查询一个键需要在每个有序段中查找一个文件，因此采用Tiered策略时查询性能较低，扫描性能也较差。而Leveled策略虽然保证了数据库的有序性更强，但每次压实操作都需要将第$k$+1层与第$k$层重叠的文件一并读出排序，涉及的数据量较大。而且原本位于第$k$+1层的许多数据在排序完成后将被再次写回第$k$+1层，存在相同数据被反复读出和写回的情况，从而造成严重的写放大。

## 2 新系统设计

本文在LevelDB上提出一种新型的键值存储系统LooseKV，以降低LevelDB的高写放大，同时维持原有的读性能。本文将弃用原来的Leveled策略和无效读触发压实等优化，在LooseKV中选择更轻量、重写浪费更少的Tiered策略。为了弥补Tiered策略导致的读性能下降，LooseKV集成了以下3个设计，在降低写放大的同时实现读写性能的平衡。

• 跳表索引优化点读性能：为减轻有序段增加导致的点读性能下降问题，LooseKV引入跳表结构的内存索引记录键值对的最新版本位于的有序段编号，从而

将查询一个键值对时需要查找的文件数量降为至多1个，有效缩短读路径。

• 有序段映射优化索引更新过程：压实操作会导致参与的键值对在磁盘上的位置发生变化，为减少对跳表索引的频繁更新，LooseKV的跳表索引仅记录键刷入磁盘时的初始位置，其新位置和初始位置的映射由另外的内存映射表记录。

• 跳表迭代器优化扫描性能：在有序段数量较多的情况下，以归并排序的方式向用户提供键值对将导致较高时延。因此，LooseKV在有序段组成的迭代器堆之间不再采用排序的方式获得下一键值对，而是直接由跳表的迭代器指出下一个键所在的有序段，从而节省迭代器之间的比较时间，有效减轻Tiered压实策略带来的扫描性能下降的情况。

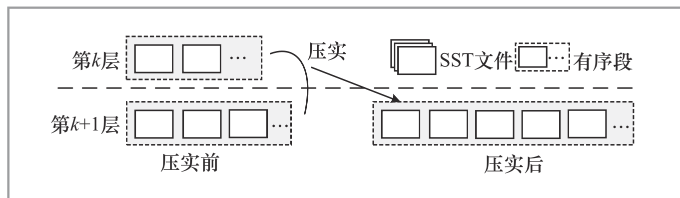



图3 Leveled策略

## 2.1 整体框架

如图4所示，LooseKV相比于LevelDB新增了两个内存组件：① 跳表索引；② 有序段映射。其中，跳表索引记录每个键值对刷写后所位于的第0层有序段编号，而有序段映射表记录该键值初始有序段编号与经过压实后所位于的新有序段编号的映射。在磁盘的分层结构上，LooseKV参与压实的粒度为有序段。当第$k$层的总大小超出限制时，将触发压实操作，后台线程选择本层最旧的$n$个（$n$由数据库参数决定）有序段，将其包含的所有SST文件进行排序去重，操作后的结果输出为单个或多个有序的SST文件，组成一个有序段，放置在第$k$+1层。Tiered压实策略保证多个旧有序段只会对应一个新有序段，为索引更新策略提供了可行性。

## 2.2 跳表索引优化读性能

对LooseKV的写入仍需要先通过

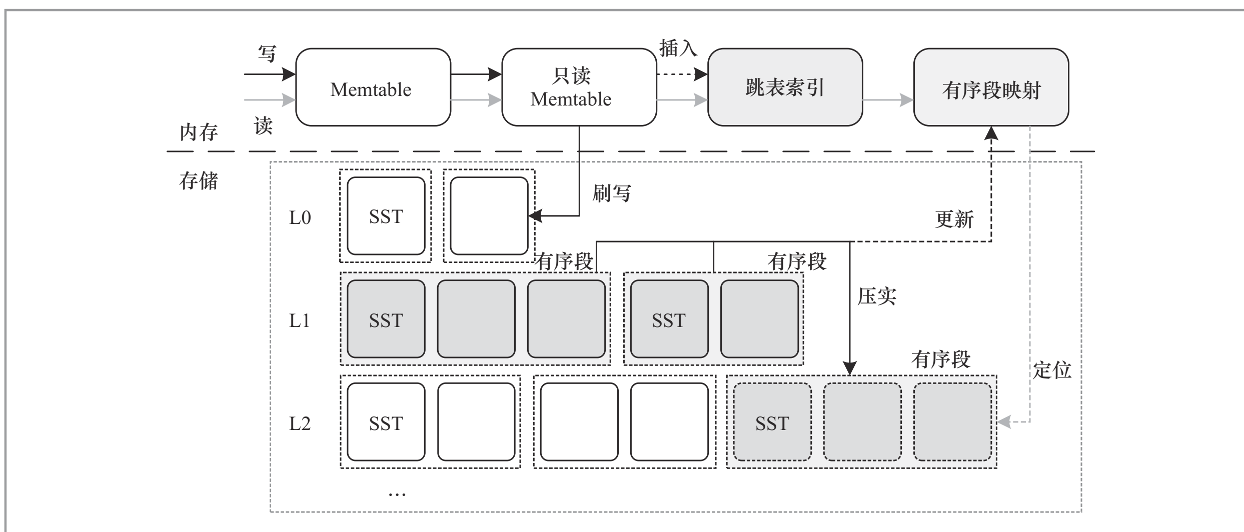



图4 LooseKV框架

Memtable，当Memtable写满之后转为只读Memtable等待刷盘。一个只读Memtable落盘时，将对应生成一个第0层的SST文件，单个SST文件也可以作为一个有序段，将其简称为L0-run（sorted run of level 0）。LooseKV在一个只读Memtable落盘、将其中的键值对写入SST文件的同时，会向跳表索引中插入或更新格式为<$k$(Key), $r$(L0-run: id)>的记录，标识键为$k$的最新条目位于ID为$r$的L0-run中。特别地，当该键的类型为删除时，将跳表中对应的条目删除。以100 GB不重复、键大小为16 B、值大小为1 024 B的键值对为例，跳表索引大约需要占用2~3 GB的内存：100 GB ÷ (16 B+1024 B) × (16 B+8 B) ≈ 2 GB（其中，8 B为ID的大小）。对于许多配备较大内存（如8 GB或更大）的现代边缘设备，这一内存开销是可接受的。

由于对跳表索引的更新仅发生在刷写过程中，且正在刷写的Memtable仍存在于内存中，因此对相关键值对的查询一定会先命中Memtable，而不会进行到查询跳表索引的步骤。当一个Memtable的刷写完成、SST文件已经生成并可见时，相应的跳表更新也已经完成，可以正确地服务点读。因此，跳表索引不需要额外的并发控制，刷写操作本身就可以保证点读操作不会访问到正在被更新的跳表索引条目。借助跳表索引能够有效定位所查询键值对所在的有序段，不再需要从上至下逐层查找，从而有效优化点读性能。

### 2.3 有序段映射优化索引更新过程

压实会导致键值对的磁盘位置发生改变，若在每次压实后更新跳表中的索引记录会带来极大的开销。因此，LooseKV以有序段的粒度来更新索引，该更新策略并不直接操作跳表中的记录，而是在另一个结构有序段映射中插入或更新<L0-run: id, 非L0-run: id>记录。

举例说明。如图5所示，假设第0层中两个ID为1和2的有序段被压实后，生成ID为10的有序段放置在第1层，则需要在有序段映射中加入条目<1, 10>和<2, 10>；ID为3和4的有序段生成ID为11的有序段同理。接着，第1层的ID为10和11的有序段进行压实生成ID为100的有序段放置在第2层，则需要将有序段映射中的条目更新为<1, 100>、<2, 100>、<3, 100>、<4, 100>，这意味着曾经位于ID为1、2、3、4的L0-run中且当前仍然存在的键值对现在位于ID为100的有序段。由于这一更新过程需要获取与参与压实的有序段相对应的L0-run的ID，若从有序段映射中遍历，开销较大。因此，在有序段的元数据信息中额外存放与这个有序段存在映射关系的L0-run ID，如ID为100的有序段，与其存在映射关系的L0-run ID则为1、2、3、4。

为保证有序段映射的并发一致性，在版本$k$上进行压实生成版本$k$+1的过程中，将会复制版本$k$的有序段映射，并在复制上进行更新，再由版本$k$+1持有。因此发生在版本$k$上的点读/扫描只会访问到该版本持有的有序段映射，而不会访问正在被更新的有序段映射。而版本$k$持有的有序段映射，在版本被释放时也会被删除。

### 2.4 点读流程

在点查询过程中，结合跳表和有序段映射这两个索引结构，将保证一次查询在磁盘上至多检查一个SST文件。相比LevelDB，LooseKV在磁盘上的查询路径明显缩短。当查询某个键时，首先依次在

Memtable和只读Memtable中搜索，命中则返回结果，同时查询结束；没有命中，则搜索跳表索引。如果在跳表索引中没有找到该键对应的索引条目，则说明该键在磁盘上也不存在，直接返回失败。反之，则获得键到L0-run的映射。如果L0-run在磁盘中存在，则查找L0-run中有且仅有的一个SST文件；如果L0-run不存在，则利用该L0-run的ID在有序段映射中查找其对应的条目，获得所查键当前所在有序段的ID，最后在该有序段中仅查询一个范围包含该键的SST文件即可结束查找。

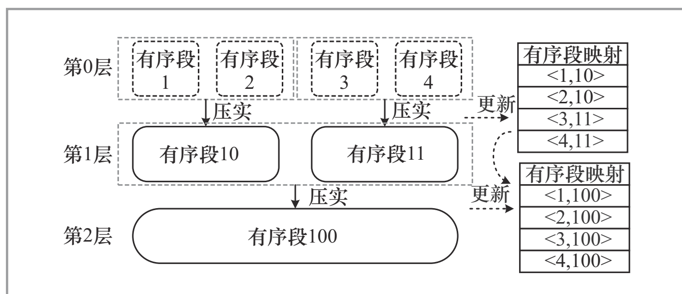



图5 有序段映射的更新

## 2.5 扫描优化

图6展示了扫描优化前的LooseKV磁盘组件部分的迭代器使用样例[16]。首先，需要在每个有序段上构建迭代器。为了从键67开始迭代，每个有序段上的迭代器都需要执行Seek(67)：采用二分查找法将迭代器指针移动到该有序段上不小于键67的最小键，在图6中，这些键以虚线指针标识。下一步，使用最小堆结构对这些键进行排序合并，从而选择第2层中ID为25的有序段迭代器所指向的键作为本次迭代向外提供的键。当还需要继续向后迭代时（Next操作），上次提供键的有序段25需要向后迭代一位，指向键71，再对所有的有序段迭代器指向的键进行合并排序，选择出有序段23迭代器指向的最小键68，作为磁盘全局迭代器向外提供的下一个键。

然而，如果每次都对所有有序段迭代器指向的键进行排序来获得下一个键，当数据库中的有序段数量较大时，上述比较过程就会导致Next操作时延较高。为了解

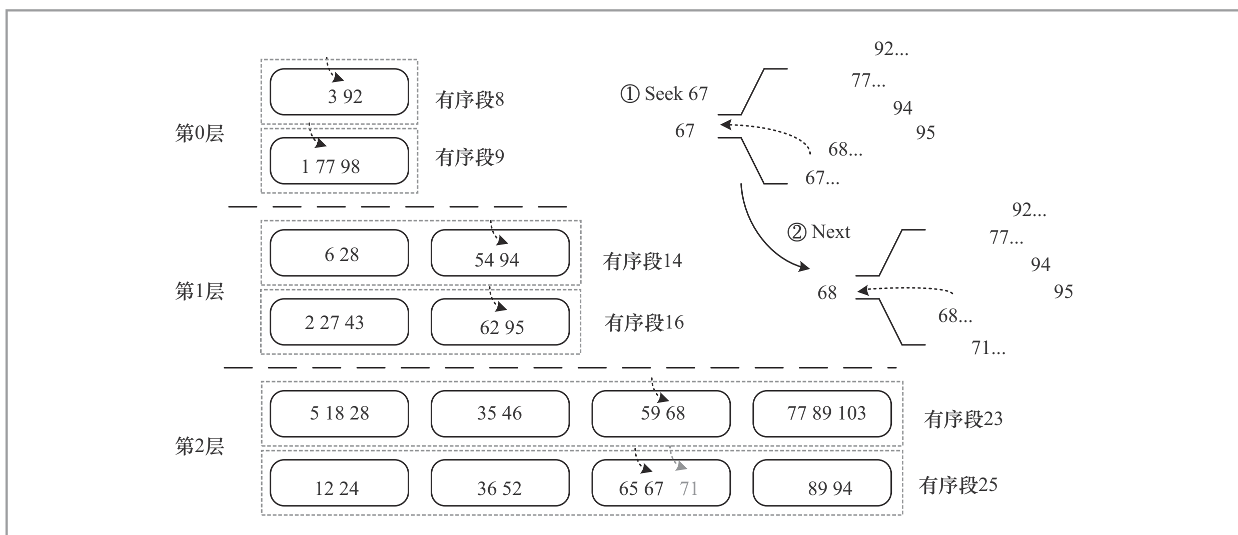



图6 优化前的迭代器[16]

决这一问题，LooseKV将跳表迭代器与磁盘组件的全局迭代器结合，如图7所示。根据跳表迭代器和有序段映射获得将要迭代的下一个键当前所在的有序段编号，将该有序段上的迭代器不停向后迭代，直到停在跳表指示的键上。该设计避免了在多个有序段中进行排序导致的高时延，有效提升了在Tiered策略下的范围扫描性能。

## 2.6 重建

LooseKV上的数据库实例关闭后，再次打开需要进行的重建操作如下。

（1）有序段映射的重建

LooseKV将有序段映射持久化到Manifest日志文件中。当一个新的Manifest日志文件开启时，数据库当前的分层信息会作为一个完整的版本对象铺平写入Manifest日志中。之后，直到该日志文件写满之前，每次刷盘或压实生成的版本更新信息都同样被铺平写入其中。不同于LevelDB，LooseKV的版本更新信息以有序段为单位记录增删，版本对象中记录的则是层与有序段的映射。此外，LooseKV还将有序段映射的变化记录在版本更新信息中。因此，当重新打开LooseKV，通过Manifest文件恢复分层结构的同时，有序段映射也完成了重建。

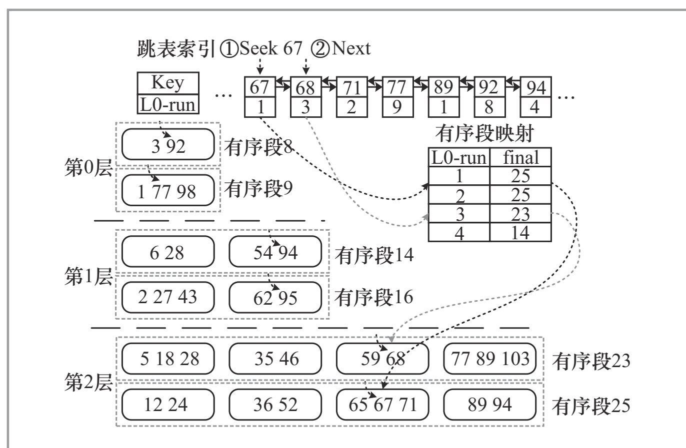



图7 跳表迭代器优化扫描性能

（2）跳表索引的重建

LooseKV并没有对跳表索引进行持久化，当数据库关闭后，跳表索引将丢失。然而，在步骤（1）完成后，层与有序段、有序段和SST文件的映射关系已经恢复，只需要扫描现存的所有有序段就可重建跳表索引。具体步骤如下：对于数据库中的每个有序段，从最下层最旧的有序段开始，按序读取组成该有序段的SST文件中的键值对，并从该有序段的元数据信息中获取与该有序段之间存在映射关系的所有L0 run的ID，在其中随机选取一个L0-run ID，将<Key, L0-run: id>插入跳表中，从而实现索引的重建。虽然跳表索引重建后与关闭时并不完全相同（Key对应的L0-run不完全相同），但仍能保证索引的正确性，实现与上次关闭前一致的索引效果。

对于某些边缘应用，数据库需要一直保持打开状态，不能频繁关闭或重启。因此，即使LooseKV在重启后需要进行索引重建，所需的时间对边缘设备的实时性影响较小。这些场景中的边缘设备通常会具有长时间稳定运行的能力，且重建过程可以在设备空闲时完成，不会显著影响实时应用的响应性能。

# 3 实验结果与分析

本节将对LooseKV进行微基准测试和宏基准测试。其中，微基准测试使用LevelDB的性能压测工具db_bench进行测试；宏基准测试使用模拟真实世界负载的YCSB基准测试[17]。其不同负载类型包含的操作类型和比例见表1。

所有实验均运行在装有Intel(R) Xeon

(R) Gold 6530 2.10 GHz处理器、256 GB DDR5 内存、3.5 TB Samsung Pcie 5.0 SSD 的服务器上，该服务器上安装 Ubuntu 24.04 LTS操作系统、6.8.0-48-generic Linux内核以及ext4文件系统。

实验选用的对比对象为LevelDB[6]、PebbelsDB[15]。其中，LevelDB为经典的基于LSM-tree构建的键值存储，第0层以上采用Leveled策略；PebblesDB采用的基于“守卫”的压实策略类似于Tiered策略，能够有效减少对输出层的大量重写，从而降低写放大。

表1 YCSB负载描述

| 负载类型 | 插入 | 查询 | 扫描 | 更新 |
|---|---|---|---|---|
| A | / | 50% | / | 50% |
| B | / | 95% | / | 5% |
| C | / | 100% | / | / |
| D | 5% | 95% | / | / |
| E | 5% | / | 95% | / |
| F | / | 50% | / | 50% |

## 3.1 微基准测试

本节实验使用了db_bench工具提供的3种负载进行测试，包括随机写入（fillrandom）、随机读取（readrandom）以及顺序读取（readseq）。对于每个负载，首先向LevelDB、PebblesDB、LooseKV对应的空数据库中载入40 000 000条键值对，再发出各个负载对应的请求40 000 000条。对于每个负载，测试实验所用键值对键大小为16 B，值大小分别为256 B、512 B、1 024 B、2 048 B时的每秒千操作完成数（kilo operations per second，KOPS）。

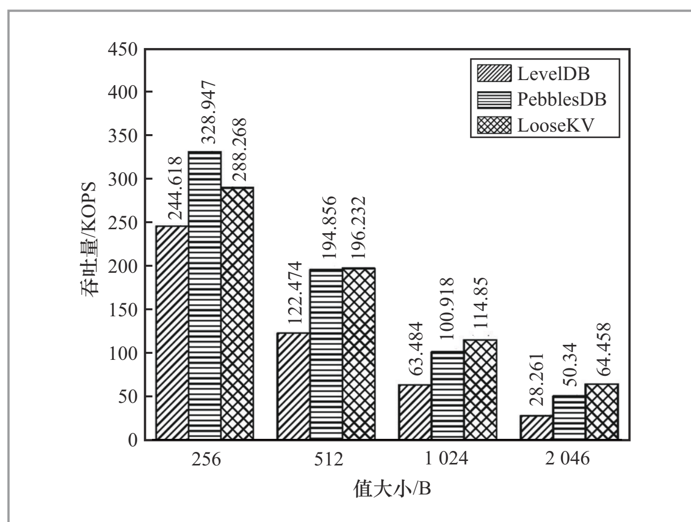



图8 随机写入测试

图8所示为各键值存储在随机写入负载下的每秒千操作完成数。相比于LevelDB，LooseKV的随机写入吞吐量提升了1.18~2.28倍。这是因为LevelDB采取的压实策略使后台压实频率较高，而且随着数据量的增大，压实越容易被触发，后台压实线程越会抢占前台写带宽，从而写时延增大。此外，在值大小为256 B时，LooseKV的随机写入吞吐量低于PebblesDB，但在值大小大于256 B时，LooseKV的随机写入吞吐量最高时为PebblesDB的1.28倍。这是因为在数据总量较少时，PebblesDB的守卫数量较少，进行压实的次数也更少，对前台写的影响更小，因此性能更高。PebblesDB应用的压实策略类似于Tiered策略，但其额外引入了“守卫”的设计以优化读性能，从而也在压实过程中引入了维护开销。而LooseKV采用最简易的Tiered策略，因此减小写放大的效果最显著。

图9所示为各键值存储在随机读取负载下的每秒千操作完成数。在值大小为256 B时，LooseKV的随机读取吞吐量低于LevelDB和PebblesDB。但在值大小大

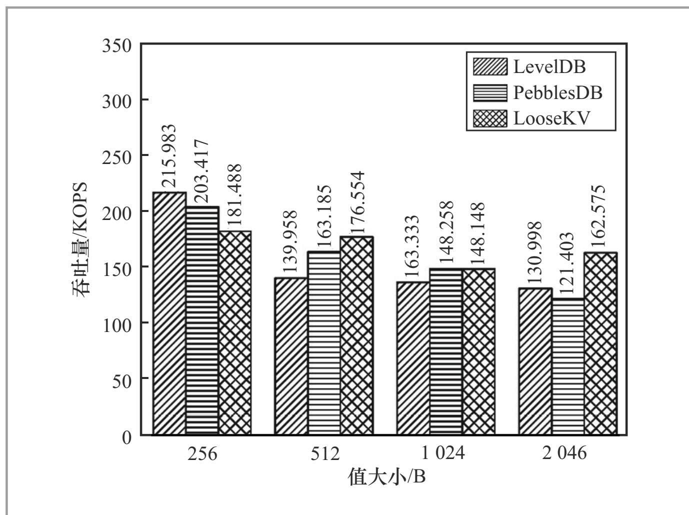



图9 随机读取测试

于256 B时，LooseKV的随机读取吞吐量为LevelDB的1.01~1.26倍，略高于PebblesDB，最高时为PebblesDB的1.34倍。显然，在值大小较小时，LooseKV的跳表索引无法发挥其优势。原因在于在同样的键值条数下，值大小越小，数据量就越少，从而LSM-tree中的层数越少，这使LevelDB和PebblesDB需要检查的SST文件数量都相应地减少。而对于

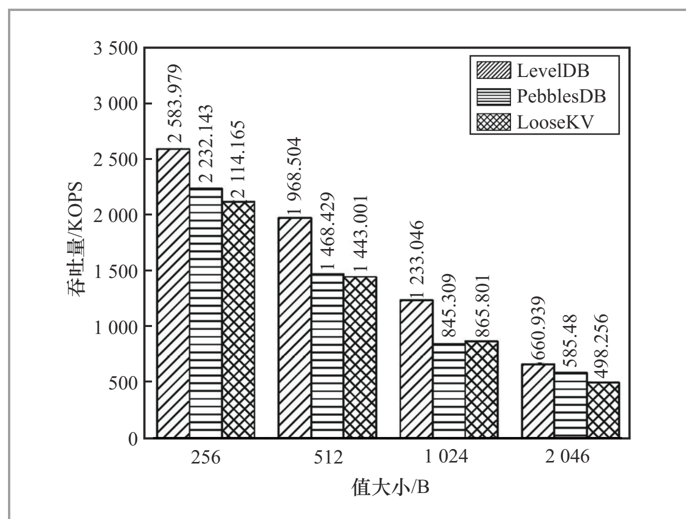



图10 顺序读取测试

LooseKV而言，由于采用了跳表索引优化点读操作，因此其点读操作固定由一次跳表索引查询、一次有序段映射以及在选定的有序段中进行至多一个的SST文件查询，并不因层数的减少而有所改善，因此相比于其他存储，在值大小较小时处于劣势。而在值较大时，LooseKV能够实现比LevelDB和PebblesDB更高的点读性能，证明了其在改善写放大的同时，并不以牺牲点读性能为代价，实现了读写性能的平衡。

图10所示为各键值存储在顺序读取负载下的每秒千操作完成数。由实验结果可知，PebblesDB和LooseKV的扫描性能均劣于LevelDB。原因是LevelDB采用Leveled策略，数据库中的有序段数量较少。而PebblesDB和LevelDB为采用Tiered策略的键值存储，数据库中存在大量过时数据和较多有序段。LooseKV加入了跳表迭代器进行优化，显著提高了Tiered策略下的扫描性能，解决了有序段数量过多导致的性能下降问题，基本与PebblesDB的顺序读取性能持平，但仍低于LevelDB。这是因为在相同数据量下，LooseKV的数据库中仍存在比LevelDB更多的过时数据，扫描整个数据库需要迭代的条目数更多。

### 3.2 YCSB

本节实验使用接近真实世界负载的YCSB基准测试对3个键值存储进行1个载入负载和6个包含不同操作负载的测试，负载的具体描述见表1。在进行实验时，首先使用载入负载Load-A填充40 000 000条键值对到空数据库中；接着依次进行Workload-A、Workload-B、Workload-C、Workload-D、Workload-F的测试，每个负载发出10 000 000条请

求，并在完成后重新使用载入负载Load-A填充空数据库；随后进行Workload-E的测试，该负载也发出10 000 000条请求。测试结果如图11所示。

由实验结果可知，对于完全由写入操作组成的载入负载Load-A，LooseKV的吞吐量为LevelDB的1.97倍、PebblesDB的1.21倍，体现了Tiered压实策略的写入优势。相比LevelDB的Leveled策略，LooseKV采用的Tiered策略不需要重写输出层的大量有效键值对，与前台写争用带宽的情况有所缓解，从而写性能有所提升。而PebblesDB虽然使用Tiered策略，但一次压实只涉及一个“守卫”内的一批文件，即压实回收的无效数据量小于LooseKV，压实效率低于LooseKV，触发频率更高，因此对前台写的影响比LooseKV更大。对于其他的写密集负载，如Workload-A，LooseKV相比于LevelDB提升1.20倍，相比于PebblesDB提升1.65倍；在写密集负载Workload-F下，LooseKV相比于PebblesDB提升1.16倍，而与LevelDB基本持平。可以发现，在读写混合的负载下，PebblesDB的优势不如在完全由写入操作组成的负载下明显，这是因为在读取的同时不断写入会使PebblesDB中有许多有序段来不及被压实合并，因此点读时需要查询的文件数量就会增多。而LooseKV由于使用跳表索引直接定位键值所在的有序段，在读写混合的负载下受干扰较轻。对于读密集型负载Workload-B、Workload-C、Workload-D，LooseKV的吞吐量能与另外两个键值存储持平或取得部分改善。然而，对于扫描为主的负载Workload-E，PebblesDB和LooseKV的吞吐量均低于LevelDB。原因在于PebblesDB和LooseKV中没有被压实清除的过时条目多于LevelDB，导致迭代过程中不可避免需要多读取和对比键值对。而PebblesDB在扫描过程中会因为Seek操作触发压实，导致随着扫描操作的进行，数据库中有序段的数量会减少，因此扫描性能优于LooseKV。

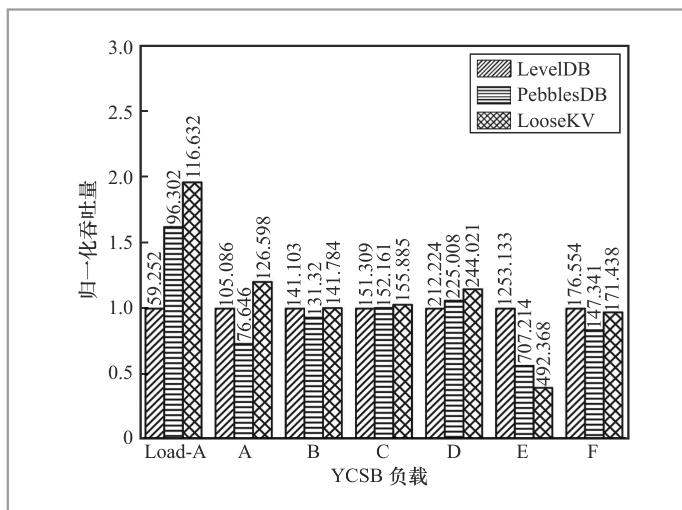



图11 YCSB测试结果

总而言之，相比于LevelDB和PebblesDB，LooseKV通过Tiered策略取得了更高的写性能，并借助跳表索引抵消改变压实策略引入的点读牺牲。与跳表索引集成的迭代器在一定程度上改善了LooseKV的扫描性能，但由于Tiered策略本身的限制，使扫描需要不可避免地迭代更多的键值对，因此扫描性能无法达到与LevelDB相近。

## 4 结束语

LSM-tree的压实操作引入了过多的重复写入，从而造成键值存储的写性能瓶颈。现有的写放大优化往往以牺牲读性能为代价，难以实现读写平衡。本文在LevelDB的基础上提出LooseKV，目的是降低压实带来的写放大开销，从而提升写

性能，同时使查询性能不受压实策略变化的影响。LooseKV使用内存跳表索引快速定位键值对所位于的有序段，并使用轻量级的索引更新策略，在最小化压实开销的同时优化点读性能。此外，LooseKV将跳表索引与数据库迭代器的构建集成，去除排序过程，减轻有序段数量增多对扫描的性能负面影响。实验表明，相比于LevelDB，LooseKV的随机写入吞吐量最高可提升2.28倍，随机读取性能最高提升1.26倍，扫描性能有所降低，但接近PebblesDB，由此证明了LooseKV能够在提升写性能的同时维持读性能。

## 参考文献：


[1] MITTAL N, NAWAB F. CooLSM: distributed and cooperative indexing across edge and cloud machines[C]//Proceedings of the 2021 IEEE 37th International Conference on Data Engineering. Piscataway: IEEE Press, 2021: 420-431.

[2] ZHOU Y, ZHOU J, CHEN S, et al. Calcspar: a contract-aware LSM-store for cloud storage with low latency spikes [C]//Proceedings of the Conference on Annual Technical Conference. Berkeley: USENIX Association, 2023: 451-465.

[3] ZHANG T, TAN J, CAI X, et al. SA-LSM: optimize data layout for LSM-tree based storage using survival analysis[J]. Proceedings of the VLDB Endowment, 2022, 15(10): 2161-2174.

[4] 林清音，陈志广. 基于更新热点感知的LSM-Tree查询优化[J]. 大数据，2023, 9(1): 126-140.
LIN Q Y, CHEN Z G. A hot-update-aware optimization to the query of LSM-Tree[J]. Big Data Research, 2023, 9(1): 126-140.

[5] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable[J]. ACM Transactions on Computer Systems, 2008, 26(2): 1-26.

[6] GHEMAWAT S, DEAN J. LevelDB[Z]. 2023.

[7] HARTER T, BORTHAKUR D, DONG S, et al. Analysis of HDFS under HBase: a Facebook messages case study[C]//Proceedings of the Conference on File and Storage Technologies. Berkeley: USENIX Association, 2014: 199-212.

[8] Facebook. RocksDB[Z]. 2023.

[9] HUANG G, CHENG X T, WANG J Y, et al. X-engine: an optimized storage engine for large-scale E-commerce transaction processing[C]//Proceedings of the 2019 International Conference on Management of Data. New York: ACM, 2019: 651-665.

[10] 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 M, HUA W D, XIE P. Key value storage technology based on LSM-tree: a survey[J]. Computer Science, 2023, 50(8): 1-15.

[11] 杜云箫，陈珂，寿黎但，等. LazyStore: 基于混合存储架构的写优化键值存储系统[J]. 软件学报，2025, 36(2): 805-829.
DU Y, CHEN K, SHOU L, et al. LazyStore: Write-optimized Key-value Storage System Based on Hybrid Storage Architecture[J]. Journal of Software, 2025, 36(2): 805-829.

[12] WANG X L, JIN P Q, HUA B, et al. Reducing write amplification of LSM-tree with block-grained compaction[C]//Proceedings of the 2022 IEEE 38th International Conference on Data Engineering. Piscataway: IEEE Press, 2022: 3119-3131.

[13] LU L Y, PILLAI T S, GOPALAKRISHNAN H, et al. WiscKey[J]. ACM Transactions on Storage, 2017, 13(1): 1-28.

[14] 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, 2020, 34(6):

2595-2609.

[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. New York: ACM, 2017: 497-514.

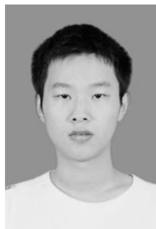
[16] ZHONG W, CHEN C. REMIX: efficient range query for LSM-trees[C]//Proceedings of the Conference on File and Storage Technologies. Berkeley: USENIX Association, 2021: 51-64.

[17] COOPER B F, SILBERSTEIN A, TAM E, et al. Benchmarking cloud serving systems with YCSB[C]//Proceedings of the 1st ACM Symposium on Cloud Computing. New York: ACM, 2010: 143-154.

## 作者简介

**郑宜湉**（2001-），女，厦门大学信息学院硕士生，主要研究方向为存储系统。

**张余豪**（1994-），男，博士，清华大学计算机科学与技术系博士后，主要研究方向为存储系统。



**霍志杰**（2001-），男，厦门大学信息学院博士生，主要研究方向为存储系统。

**舒继武**（1968-），男，博士，清华大学计算机科学与技术系教授，主要研究方向为存储系统、体系结构。

**收稿日期**: 2025-02-10

**通信作者**: 张余豪,yuhaopaul@163.com;舒继武,jwshu@xmu.edu.cn

**基金项目**: 国家自然科学基金青年科学基金项目(No. 62402204);国家自然科学基金联合基金项目(No. U22B2023)

**Foundation Items**: Young Scientists Fund of the National Natural Science Foundation of China (No. 62402204), Joint Funds of the National Natural Science Foundation of China (No. U22B2023)