计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
[doi: 10.15888/j.cnki.csa.010237] [CSTR: 32024.14.csa.010237]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

Split-LSM-Tree: 基于动态树分裂的高性能键值存储系统①

曹宇昂[1], 郭　帆[2], 朱文喆[1], 李永坤[1]

[1](中国科学技术大学 计算机科学与技术学院, 合肥 230027)
[2](闽江学院 计算机与大数据学院, 福州 350108)
通信作者: 郭帆, E-mail: lps56@mail.ustc.edu.cn

摘　要: 基于日志结构合并树 (log-structured merge-tree, LSM-tree) 的键值存储系统采用分层架构, 层级越多, 读写放大问题越严重. 针对这一问题, 本文提出一种优化的键值存储引擎 Split-LSM-Tree. 该引擎将单个 LSM-tree 动态分割为子树森林, 从而有效降低子树深度. 系统通过负载感知的自适应机制精准触发分裂, 并结合非阻塞分裂算法与 I/O 优化策略, 在实现子树分裂的同时, 可以保障前台读写操作的连续性. 实验结果表明, 与广泛应用的 LevelDB 和 RocksDB 相比, Split-LSM-Tree 的写放大系数分别降低了 25.7% 与 18.1%, 写吞吐量分别提升至 2.9 倍与 2.4 倍; 在 YCSB 典型负载下, 其综合吞吐量较键值分离引擎 WiscKey 最高提升 30%.

关键词: 存储系统; 键值存储; 日志结构合并树; 读写放大

引用格式: 曹宇昂,郭帆,朱文喆,李永坤.Split-LSM-Tree: 基于动态树分裂的高性能键值存储系统.计算机系统应用. http://www.c-s-a.org.cn/1003-3254/10237.html

Split-LSM-Tree: High-performance Key-value Storage System Based on Dynamic Tree Splitting

CAO Yu-Ang[1], GUO Fan[2], ZHU Wen-Zhe[1], LI Yong-Kun[1]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
[2](School of Computer Science and Big Data, Minjiang University, Fuzhou 350108, China)

Abstract: Key-value (KV) storage systems based on log-structured merge-tree (LSM-tree) architectures are organized hierarchically, where an increasing number of levels exacerbates read and write amplification. To address this issue, this study proposes Split-LSM-Tree, an optimized KV storage engine that dynamically partitions a single LSM-tree into a forest of sub-trees, thus effectively reducing the sub-tree depth. Specifically, a load-aware adaptive mechanism is designed to precisely trigger splitting. Furthermore, a non-blocking split algorithm combined with I/O optimizations ensures uninterrupted front-end read and write operations during splitting. Experimental results show that Split-LSM-Tree reduces write amplification by 25.7% and 18.1% compared to LevelDB and RocksDB, respectively, while increasing write throughput by 2.9× and 2.4×. Under typical YCSB workloads, its overall throughput is improved by up to 30% compared with the KV-separated engine WiscKey.

Key words: storage system; key-value storage; log-structured merge-tree (LSM-tree); read/write amplification

1　引言

键值存储系统为现代数据密集型应用的高效数据存取提供了关键支撑, 广泛应用于电子商务[1]、社交网络[2]、图像存储[3]及云存储[4]等诸多领域. 在云计算、物联网及人工智能飞速发展的今天, 数据密集型应用正面临着前所未有的挑战. 例如, 现代电子商务平台的

① 收稿时间: 2026-01-25; 修改时间: 2026-02-14; 采用时间: 2026-02-28; csa 在线出版时间: 2026-07-17

实时推荐系统需要在毫秒级内处理数百万次的用户行为日志写入; 物联网监控系统需应对海量传感器数据的持续高并发注入; 在 Facebook 的生产环境中, 底层存储引擎支撑了超过 30 个不同的核心业务应用, 管理着总计数百 PB 的海量数据[5]. 作为这些应用的核心数据存储底座, 键值存储系统必须具备良好的读写吞吐能力与可扩展性.

目前, 主流键值存储系统 (如 Cassandra[6]、RocksDB[7]、TiKV[8]、WiredTiger[9]等) 普遍采用日志结构合并树 (log-structured merge-tree, LSM-tree) [10]架构. 该架构的核心优势在于其将用户对磁盘的随机写请求转化为了高效的顺序写操作, 同时针对读操作优化了其存储结构设计, 从而实现了较强的读写性能. 如图 1 所示, 在存储结构上, LSM-tree 采用分层设计, 每层由多个存储有序键值对的文件 (SSTable) 组成, 且各层容量随层级深度的增加呈指数级增长 (下层容量通常为其上层的 10 倍), 形成了典型的树状层次结构. 其中, 最上层 (L_0) 直接接收来自内存的数据刷写, 其层内 SSTable 间的键值范围可能会重叠; 而后续层级 (L_1–L_n) 则严格保持层内 SSTable 间的键值范围互不重叠且有序排列, 但不同层级之间所维护的键值范围仍可能存在交叉.

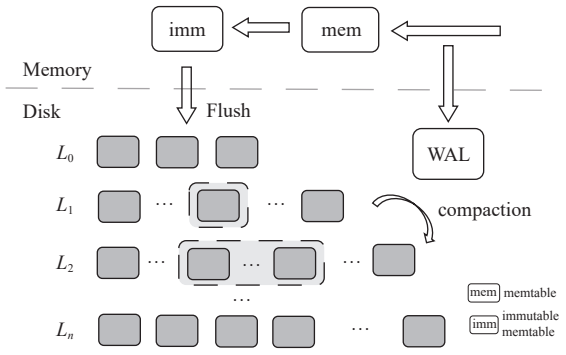

图 1 LSM-tree 存储架构

在此分层架构下, LSM-tree 遵循特定的数据处理流程: 写入请求首先缓存在内存表 (memtable) 中; 当 memtable 达到容量阈值时, 缓存的键值对将以文件形式 (SSTable) 刷写至磁盘中的最上层, 即 L_0 层. 随后, 系统通过后台的 compaction 操作将数据逐步向深层迁移, 该操作利用归并排序对不同层级中的 SSTable 进行数据去重与整理, 以维持数据读取效率并回收存储空间. 然而, 这种分层数据移动机制会引发严重的写放大与读放大效应. 写放大源于数据在不同层级中的重复写入, 现有研究表明 LSM-tree 的写放大系数可达 50 倍以上, 即每写入 1 GB 的数据可能产生超过 50 GB 的磁盘 I/O 负载[11,12]. 读放大则由于单次查询可能需要探测多个层级的 SSTable, 在大规模数据集下, 系统的读放大系数最高可达 300[11]. 严重的读写放大会占用大量磁盘 I/O 带宽, 加速 SSD 的损耗, 也会增加数据读写的延迟, 读写放大已成为键值存储系统在数据密集型场景下的关键瓶颈[11]. 实验数据显示, 当数据集规模从 10 GB 扩展至 400 GB 时, 受上述放大效应的影响, 系统的写入延迟增加了 154%.

现有研究主要通过键值分离[11,13]与弱化数据的有序性约束[14,15]两类策略来缓解读写放大. 键值分离架构将值存储在独立的值日志中, 仅在 LSM-tree 内存储键及其对应的值指针, 旨在通过减小树结构的存储规模来抑制读写放大效应; 弱化数据有序性约束方案则是允许同层的 SSTable 之间存在键范围重叠, 以此来降低 compaction 操作的频率和开销. 然而, 这两类方案均保留了传统 LSM-tree 的多层级树状架构, 因此在管理大规模数据集时, 系统仍需构建很深的层级结构, 进而引发严重的读写放大问题.

本文提出了 Split-LSM-Tree 架构, 该架构的核心在于通过自适应分裂触发机制、非阻塞分裂算法以及一系列 I/O 优化策略, 将传统的单体 LSM-tree 动态划分为多个容量受限的独立子树. 本文实现了 Split-LSM-Tree 的原型系统, 并将其与一些基准架构进行对比测试. 实验结果显示, 在写密集型负载下, 相较于 LevelDB[16]和 RocksDB[7], 本方案分别将写放大降低了 25.7% 与 18.1%, 并将写吞吐量提升至 2.9 倍与 2.4 倍. 此外, 在 YCSB 负载的测试中, 本系统相较于 WiscKey[11]实现了最高 30% 的性能收益. 本文的主要贡献如下.

(1) 设计从单体 LSM-tree 到子树森林的动态分裂机制, 当系统负载或规模达到阈值时自动执行分裂, 从而有效抑制层级加深带来的读写放大问题.

(2) 通过引入版本化指针重定向机制与基于元数据切换的后台文件重组机制, 实现近乎零阻塞的数据迁移, 保障了分裂过程中前台读写请求的连续性.

(3) 基于动态森林架构与非阻塞分裂算法实现了 Split-LSM-Tree 原型系统. 实验结果证明, 相较于 LevelDB、RocksDB 与 WiscKey, 该系统可以有效缓解读写放大, 提高吞吐率.

2 研究动机与技术挑战

2.1 研究动机

LSM-tree 的写入操作首先将键值对同步写入 memtable 与预写日志 (write ahead log, WAL) 中. 当 memtable 达到容量阈值时, 系统会将其刷写至磁盘, 转变为 L_0 层的 SSTable; 为实现数据的快速刷写, 允许 L_0 层内 SSTable 间的键值范围存在重叠. 当某层级超过其容量阈值时, 系统会触发 compaction 操作: 系统从当前层级 (L_i) 选取部分 SSTable, 并与下一层级 (L_{i+1}) 中与之存在键值范围重叠的文件进行归并排序, 最终生成新的 SSTable 存入 L_{i+1}. 该机制在保持各层级有序性的同时将数据向下迁移, 并将随机写请求转化为高效的顺序磁盘操作.

读操作遵循由新至旧的原则, 首先在包含最新数据的 memtable 中进行查找. 若未命中, 系统将自上而下逐层查找 (从 L_0 到 L_n) 磁盘中存储的数据, 并返回首个匹配到的结果. 由于 L_0 层存在键值范围重叠, 点查询需检查该层所有可能包含目标键值对的 SSTable; 而 L_1–L_n 层的层内数据严格有序, 因此系统定位每层中唯一的候选文件并进行查找即可.

Compaction 机制是写放大问题的源头. LSM-tree 将随机写请求转化为顺序追加操作, 实现了较高的写入吞吐量, 但这会导致数据呈现无序或弱有序状态, 进而使读性能降低. 为保证数据的高效读取, 系统通过 compaction 操作强制实现数据有序化, 写放大便在此过程中产生. 在 compaction 过程中, 当 L_i 层的一个 SSTable 合并至 L_{i+1} 层时, 需识别 L_{i+1} 层中所有与之存在键范围重叠的文件. 考虑到层级间呈指数级增长的存储容量 (通常为 10:1), L_i 层的单个 SSTable 可能需要与 L_{i+1} 层的多达 10 个 SSTable 进行合并. 因此, 单次 compaction 操作可能产生接近 10 倍的写放大. 随着数据在多层级间的逐层下移, 这种放大效应将持续累积, 使 LSM-tree 的实际写放大系数远高于 10.

读放大源于 LSM-tree 部分有序的结构设计, 点与范围查询可能需要进行跨层级的多次磁盘读取: 最坏情况下需检测 L_0 层的全部 SSTable, 以及后续每个层级 (L_1–L_n) 中的一些候选文件. 每次 SSTable 查询又可能引发多次数据块读取. 例如, 若 L_0 层包含 6 个 SSTable 且 LSM-tree 共包含 7 个层级 (L_0–L_6), 则单次点查询最多可能访问 12 个 SSTable.

文献[11,17]研究表明, 读写放大是制约键值存储系统性能的关键因素. 本文主要关注数据规模增长导致读写放大加剧和系统吞吐量下降这一核心问题. 我们在 LevelDB 上的实验结果 (见图 2) 表明: 随着数据集规模的增长, 写放大系数呈单调上升趋势, 同时系统吞吐量显著降低. 这种性能下降源于 LSM-tree 层级加深引发的更频繁、涉及数据量更大的 compaction 操作. 这些操作产生的后台 I/O 与 CPU 开销会加剧耗尽磁盘带宽与计算资源, 进而影响前台操作的执行, 导致系统整体性能的下降.

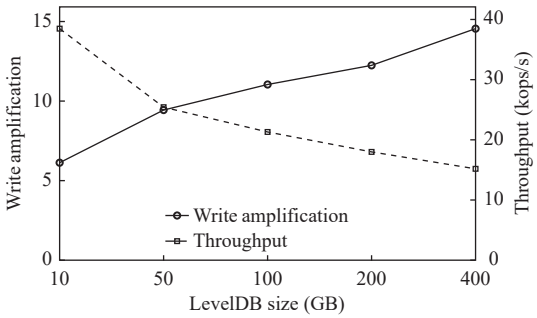

图 2 写放大与吞吐率

2.2 技术挑战

面对上述因层级加深导致的性能衰减, 一种直观且在工业界被广泛使用的工程解决方案是采用静态分区架构[18,19]. 即预先根据键值范围将数据集强行划分为多个固定分片, 每个分片由一个独立的小型 LSM-tree 实例管理. 尽管这种朴素方案在理论上能够限制单棵树的规模并降低树高, 但在缺乏工作负载先验知识的实际应用中, 它面临着难以克服的缺陷. 由于真实场景下的键值分布难以预测且可能存在数据倾斜, 静态分区大概率会导致子树之间负载不均衡, 热点分区的树高会迅速增长并成为系统性能瓶颈, 而冷分区的资源则被浪费. 此外, 在数据规模较小的场景下, 维护多个独立 LSM-tree 会引入额外的内存开销 (如每棵树需维护独立的 memtable), 且可能在并发 compaction 的执行过程中引起 I/O 竞争.

为了克服静态分区方案的缺点, 本文提出基于动态分区机制的架构 Split-LSM-Tree. 系统初始时以单体 LSM-tree 运行, 当存储容量超过预设阈值时, 自动分裂机制会将其划分为容量均衡的子树森林. 通过对子树容量进行约束, 该设计有效抑制了读写放大效应, 提高系统性能, 同时该设计也能有效避免静态分区方案的负载不均衡与资源浪费的问题. 然而, 该设计也带来了 3 项关键挑战: 首先, 分裂操作会产生大量数据拷

贝开销且可能会中断前台的数据读写请求, 因此如何设计用户无感知的轻量化分裂机制是一个重要问题; 其次, 树结构重组的过程必然会影响读写性能, 因此如何在维持性能稳定的同时, 兼顾范围查询的数据局部性也是一个待解决的挑战; 最后, 过早或过迟分裂都会导致负载不均衡与系统资源浪费, 如何构建自适应阈值模型以精准且及时地触发分裂也是一个重要的挑战.

3 系统设计

3.1 总体架构

Split-LSM-Tree 采用基于动态范围划分森林的架构 (如图 3 所示). 该架构的核心设计思想是通过自动分裂机制, 将传统的、不断纵向加深的单体 LSM-tree 演化为由多个存储规模受限的子树组成的森林. 每个子树负责管理一段连续且互不相交的键值范围.

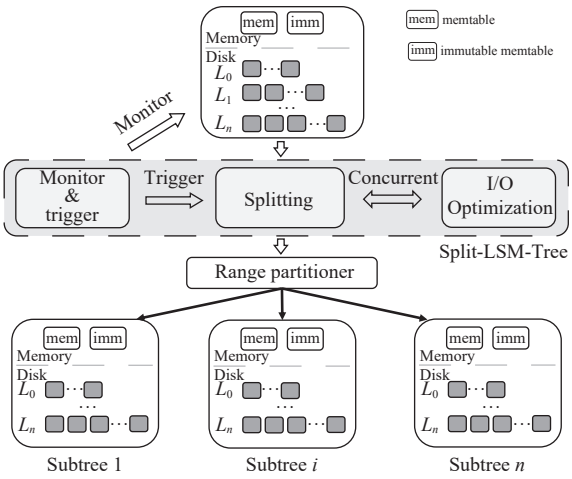

图 3 Split-LSM-Tree 系统架构

为保障与现有应用的兼容性, Split-LSM-Tree 实现了与传统 LSM-tree 一致的访问接口. 系统内部通过一个轻量级的全局路由表进行请求分发, 该表记录了各子树管理的键值范围. 当用户发起读写请求时, 路由层通过高效的二分查找定位目标子树, 并将操作转发至对应的子树实例. 这种设计在逻辑上对用户隐藏了底层森林架构的实现, 从而实现了访问的透明性, 使用户无需感知内部子树的分裂与分布细节.

该架构包含以下 3 个核心功能模块: (1) 自适应分裂触发机制: 该模块动态监测各子树的存储容量与性能指标 (如读写请求延迟), 当相关指标达到阈值时触发分裂, 通过自适应阈值确保分裂操作能及时抑制读写放大, 同时又能有效避免频繁分裂产生的资源开销. (2) 高效子树分裂算法: 该模块负责执行子树分裂操作, 以减小子树中每层的数据量, 从而减少参与 compaction 操作的数据量, 最终降低写放大. 算法首先采用贪心策略选取最优划分键, 随后利用元数据更新等技术最大限度地规避物理数据迁移, 从而最小化分裂过程中的 I/O 开销. (3) I/O 优化模块: 该模块负责在分裂期间保障前台读写请求的持续处理. 通过引入轻量化的 memtable 切换机制和版本化元数据快照机制, 系统实现了非阻塞的前台读写操作, 从而确保了分裂过程中系统的高可用性.

系统工作流程如下: 分裂触发模块实时监控系统性能指标, 一旦监测到相关指标达到阈值, 系统便会启动分裂算法. 在分裂执行期间, I/O 优化模块会保证前台读写操作不被阻塞, 多模块协同工作以实现较高的系统整体性能.

3.2 自适应分裂触发机制

确定分裂时机是一个关键设计挑战: 过早分裂会造成内存资源和 CPU 资源的浪费, 而过晚分裂则会导致 LSM-tree 层级过深, 进而引发读写性能下降. 为此, 本文设计并实现了基于存储规模与性能指标双重监测的自适应分裂触发机制, 确保仅在必要时触发分裂.

系统通过持续追踪各子树的存储容量 S 与预设容量阈值 C_{limit} 的关系来管理分裂状态. 当 $S<C_{\text{limit}}$ 时, 分裂不会被触发, 此时系统会持续采集子树在这一阶段的读写延迟数据, 作为基准性能指标 L_{base}. 当存储容量超过 C_{limit} 之后, 系统会以固定的时间窗口监测性能, 计算每个窗口内的平均读写延迟 L_{curr}, 当满足条件 $L_{\text{curr}}>L_{\text{base}}\times\sigma$ 时 (其中 σ 为可配置的分裂敏感度参数), 系统触发分裂. 这种双条件触发机制确保分裂操作仅在存储压力真正转化为性能瓶颈时发生.

Split-LSM-Tree 为确保子树间的性能隔离, 为每个子树分配了独立的内存组件, 包括 memtable、immutable memtable 与数据块缓存. 虽然这种子树独立化管理策略增强了数据局部性, 但也带来了与子树数量成正比的内存压力. 为防止出现内存溢出 (out of memory, OOM) 问题, 系统构建了全局资源控制器, 设定内存占用上限. 当所有子树累计占用的内存超过服务器总可用内存的预设百分比时, 系统将强制暂停新的分裂请求. 此阶段中现有子树仍维持正常运行, 同时系统会定期重新评估内存可用性, 只有当内存用量回落至阈值以下时, 分裂触发机制才会恢复.

3.3 子树分裂算法

Split-LSM-Tree 的子树分裂是一个涉及多层级组件协同的重组过程. 该机制需要对原子树 (即待分裂的子树) 内的所有核心组件进行迁移或拆分, 涵盖易失性的内存结构 (memtable、immutable memtable 和元数据) 以及持久化存储在磁盘各层级中的 SSTable 文件. 该算法的核心设计目标是在保证数据一致性的前提下, 以尽量小的物理 I/O 开销与计算资源消耗来实现从单体结构向森林架构的转化.

3.3.1 划分键选择策略

分裂过程的首要环节是确定一个最优划分键. 为便于描述, 本文将被分裂的子树称为“原子树”, 其管理的键值范围记为$[k_{start}, k_{end})$, 其分裂后产生的两棵子树分别称为“左子树”和“右子树”. 分裂的目标是根据选定的划分键 k_{split}, 将原子树的键空间切分为两个连续且互不相交的区间, 左子树接管$[k_{start}, k_{split})$, 而右子树接管$[k_{split}, k_{end})$. 为在分裂产生的子树间实现均衡的数据分布, 本文设计了基于元数据采样的贪心选择策略, 该方法不依赖静态的数据中值计算, 能够适应动态的工作负载.

该策略的核心在于利用系统为每个 SSTable 维护的元数据信息 (如文件大小、键值范围边界等) 来构建数据分布的近似直方图. 算法流程如算法 1 所示, 首先选取与键空间中值 (即键范围上下界的中点) 最为接近的 SSTable 边界键作为初始候选划分点, 随后进入迭代流程: 通过累加当前候选点两侧的 SSTable 文件大小来估算左右子树的容量. 若两子树的预估容量之差未能满足预设的负载均衡约束 (即差距不超过原子树数据量的 5%), 算法将执行类似于二分查找的迭代逻辑, 计算当前候选点与容量较大一侧的子树边界键的中值, 选取与之最接近的 SSTable 边界键作为下一轮迭代的候选点. 为了加速算法收敛并防止搜索路径在目标值的左右两侧震荡, 当算法检测到连续两次迭代的候选点分别位于理想划分点的两侧时 (即一次偏左、一次偏右), 将选取这两个候选键的中值作为新的候选点. 该过程循环执行, 直至定位到满足均衡条件的 k_{split}.

算法 1. 划分键选择算法

输入: 原子树的键值范围边界 K_{max}、K_{min}, 原子树所有 SSTable 的元数据信息.

输出: 划分键 k_{split}.

1. $k_{left} \leftarrow K_{min}$, $k_{right} \leftarrow K_{max}$
2. $k_{curr} \leftarrow$ *FindClosestBoundrary*$((k_{left} + k_{right})/2)$ //初始选择接近中值的边界键
3. *History* ← [] // 记录历史候选点以检测震荡
4. **while** True **do**
5. $S_{left}, S_{right} \leftarrow$ *EstimateSize*(k_{curr}) // 计算左右子树的大小
6. $Diff \leftarrow |S_{left}–S_{right}|/S_{total}$
7. **if** $Diff < \epsilon$ **then return** k_{curr} **end if**
8. Append k_{curr} to *History*
9. **if** *IsOscillating*(*History*) **then** // 检测候选点是否在理想划分点的左右两侧震荡
10. $k_{curr} \leftarrow$ (*History*[*last*] + *History*[*last*–1])/2 // 取前两次候选点的中值
11. *History* ← []
12. continue
13. **end if**
14. **if** $S_{left} > S_{right}$ **then** // 左侧过大, 向左侧区间二分查找
15. $k_{right} \leftarrow k_{curr}$
16. $k_{curr} \leftarrow$ *FindClosestBoundrary*$((k_{left} + k_{curr})/2)$
17. **else**
18. $k_{left} \leftarrow k_{curr}$
19. $k_{curr} \leftarrow$ *FindClosestBoundrary*$((k_{curr} + k_{right})/2)$
20. **end if**
21. **end while**

3.3.2 Memtable 与 immutable memtable 的处理

为了保障子树分裂期间前台写入操作的连续性, Split-LSM-Tree 引入了写路径重定向机制. 如图 4 所示, 当分裂指令触发时, 系统首先将当前的活跃内存表 (memtable) 转化为不可变的 immutable memtable, 同时系统会分别为左右子树实例化独立的 memtable 和对应的预写日志, 将新的写入请求重定向至对应子树的 memtable 中, 从而确保分裂期间前台写入请求不被中断. 这种设计将子树分裂过程移出了前台写操作的关键路径, 保障了系统的持续可用性.

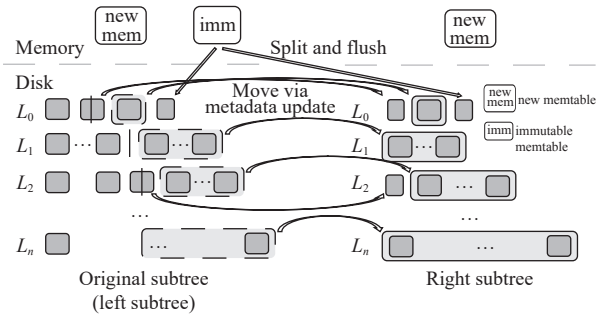

图 4 子树分裂过程

在数据持久化阶段, 系统实现了智能分裂逻辑以优化 I/O 开销并降低写放大. 系统会根据不同的 minor compaction (即 memtable 刷写至磁盘的过程) 执行状态

而采取不同的处理方式: 若分裂触发时 minor compaction 尚未启动, 系统将直接在内存中对 immutable memtable 执行分裂, 以 k_{split} 为界将其划分为两个 SSTable, 分别持久化到左右子树的 L_0 层, 避免了中间冗余的磁盘写入; 反之, 若分裂触发时, minor compaction 已经开始执行, 系统会暂停当前的分裂操作直至其执行完成.

这种双路径处理策略通过在内存阶段提前消除数据的重叠状态, 尽可能地避免了先将内存数据持久化至磁盘后再重新读出进行分裂而导致的冗余磁盘操作与额外的写放大. 此外, 通过将原子树与新子树的 WAL 进行解耦, 系统不仅简化了分裂期间的并发访问控制, 还确保了在发生故障时, 系统的数据一致性能够得到保障.

3.3.3 SSTable 与元数据的处理

针对 L_0 层与更深层 (L_1–L_n) 结构特性的差异, Split-LSM-Tree 采用差异化的处理策略以提升分裂效率. 对于 SSTable 间存在键值范围重叠的 L_0 层, 系统执行逐文件粒度的评估与划分. 对于键值范围完全位于划分键 k_{split} 单侧的文件, 系统仅需执行逻辑迁移, 即通过修改元数据来将其转移至对应子树, 该过程没有磁盘 I/O 开销. 若 SSTable 的键值范围跨越了 k_{split}, 系统则会对其进行物理分裂: 首先通过二分查找快速定位文件内的划分点, 随后在划分点处进行分裂, 生成新的 SSTable 时可以复用原文件的数据块和布隆过滤器, 仅需重新构建索引块, 从而避免了对数据进行重新编码的开销.

对于 L_1–L_n 层, 由于层内 SSTable 间没有键值范围重叠且有序排列, 系统通过批量元数据操作实现高效分裂. 每一层中至多存在一个 SSTable 跨越划分键, 系统首先定位并物理分裂该文件; 随后, 系统通过元数据更新将 k_{split} 左侧的所有 SSTable 整体划归到左子树, 右侧文件则划归到右子树. 这种元数据驱动的文件移动机制使得深层级中绝大部分的磁盘数据无需进行物理移动.

在元数据管理层面, 每棵子树会在内存中维护独立的元数据索引视图, 包括 SSTable 的文件路径、键值边界和层级统计信息等内容. 对于无需物理分裂的 SSTable, 其在不同子树间的重新分配完全在内存中的元数据结构内完成. 为了确保故障恢复一致性, 系统遵循先物理持久化、后逻辑提交的原则. 对于需要物理分裂的文件, 系统需首先确保新生成的 SSTable 完成磁盘持久化, 再更新元数据来完成子树视图的切换. 这种机制确保了即使在分裂过程中发生宕机, 系统重启后仍能依据完整的元数据日志恢复至稳定状态.

3.4 I/O 优化机制

Split-LSM-Tree 通过非阻塞的分裂算法与多维度的并行优化策略, 确保了系统在分裂过程中的前台响应能力与整体 I/O 效率.

在分裂过程中, 为了实现前台读操作的正常执行, 所有待迁移的 SSTable 文件及其关联的元数据均在原子树中保持有效, 直至分裂完成. 这种数据保留策略确保读操作可通过原子树的现有接口无中断执行, 在保持一致性的同时不产生性能损耗. 仅当新子树结构完全持久化, 且双方元数据更新完成后, 系统才会将查询请求重定向至对应子树. 随后系统通过后台的延迟垃圾回收机制异步清理过期的信息, 实现了读路径与分裂路径的解耦.

针对写入操作, 系统通过资源预分配与路径重定向确保了其在分裂期间的并发性. 在分裂启动时, 系统立即为两棵新子树实例化独立的 memtable 与预写日志. 输入数据流可直接写入目标子树的内存结构, 无需额外的协调开销, 这种架构级隔离确保了写入请求独立于后台进行的分裂操作, 使得写入路径几乎不受分裂过程的影响, 从而保障了系统在分裂期间依然能维持稳定的写性能.

针对森林架构中多子树并存的特性, Split-LSM-Tree 设计了基于子树解耦的多线程任务执行模型. 系统为每个子树分配专属的访问线程, 当某一子树因 compaction 压力出现写停滞时, 其他子树的读写操作仍能正常进行, 从而避免了单线程访问导致的系统性能瓶颈. 该设计能够在系统局部拥塞时保障系统的整体性能. 对于跨子树的范围查询, 系统采用分治策略, 根据各个子树的键值范围边界将其拆分为多个子范围查询, 由多个线程并发执行这些查询请求, 再对结果进行合并, 从而加速跨子树的数据扫描.

在多树架构下, 各子树会独立触发 compaction 操作. 为降低总体 compaction 时间并缓解其对前台操作的影响, Split-LSM-Tree 采用多线程执行机制, 为每个子树分配独立线程执行该操作. 同时, 系统通过基于优先级的策略来调度不同子树的 compaction 操作, 防止单个子树操作频次过高影响其他子树的性能, 在保障前台响应性的同时提升了后台数据组织的效率.

4 系统实现与性能评估

本节首先介绍了基于 Split-LSM-Tree 架构的原型系统的工程实现细节, 随后介绍了实验环境配置与测试方法, 接着展示不同测试下的实验结果, 并对其进行详细的分析与讨论.

4.1 系统实现

为了验证 Split-LSM-Tree 架构的可行性与性能优势, 本文基于 LevelDB 实现了系统原型. 为了实现多子树协同的森林架构, 本研究对 LevelDB 的核心引擎层 (包括 DBImpl 等关键类) 进行了重构与扩展. 在具体实现上, 我们将每个子树封装为一个完整的 LevelDB 实例, 各实例拥有独立的 memtable、预写日志及 L_0–L_n 层的文件集合.

为了实现动态分裂与并行处理, 系统引入了一系列关键调度与并发控制机制. 首先, 在后台监控方面, 系统为每棵子树配置了独立的后台监控线程, 该线程以固定周期对存储规模与 I/O 压力指标进行采样. 当这些指标触发预设的分裂阈值时, 系统会异步启动分裂执行线程. 我们采用了互斥锁机制来协调 memtable 的刷写、磁盘文件的重分配以及全局元数据的更新, 确保在分裂过程中数据的一致性. 此外, 在对外接口层面, 系统通过构建一层兼容性封装层来处理用户的读写请求, 并根据目标键值范围将请求精准路由至对应的子树实例. 这种保持与原生 LevelDB 接口兼容的设计对上层应用屏蔽了底层结构变化的复杂性. 这些设计赋予了系统动态扩展的能力, 有效抑制了读写放大效应并提升了整体性能.

4.2 实验环境设置

本节旨在对比评估 Split-LSM-Tree 与 3 种主流键值存储引擎 (LevelDB[16]、RocksDB[7]与 WiscKey[11]) 的性能表现. 实验平台是一台搭载了两颗 Intel Xeon Platinum 8360Y 处理器的服务器, 单颗处理器拥有 36 个物理核心, 基准频率为 2.40 GHz. 系统配备了 256 GB 的物理内存, 并采用 480 GB 的三星 MZ7KH480H AHQ0D3 SATA 固态硬盘作为底层存储介质. 系统运行 Ubuntu 20.04 LTS 操作系统.

在软件参数配置方面, 对于 Split-LSM-Tree, 单个子树的 memtable 容量配置为 64 MB, 数据块缓存容量设为 1 GB. 自适应分裂触发机制中, 分裂容量阈值设定为 20 GB, 分裂敏感度参数σ设为 1.1. 为进行公平对比, 测试中为 LevelDB、RocksDB 和 WiscKey 配置了 256 MB 的 memtable 与 4 GB 的数据块缓存.

4.3 Micro-benchmarks

本文首先评估不同数据集规模下的系统性能. 工作负载的键大小为 16 字节, 值大小为 1 KB, 实验通过执行均匀随机写入操作来填充键值存储, 数据集的总规模为 100–300 GB. 如图 5 所示, 在所有数据集规模下, Split-LSM-Tree 均表现出最低的写放大系数与最高的写入吞吐量. 在 100 GB 数据量下, 相较于 LevelDB 和 RocksDB, Split-LSM-Tree 分别将写放大降低了 23% 和 18%, 同时将写入吞吐量分别提升至 3.7 倍与 2.8 倍. 同时, 当数据集规模扩展至 300 GB 时, 其写放大抑制优势更加显著, 较 LevelDB 与 RocksDB 分别降低 25.7% 与 18.1%, 写入吞吐量则相应提升至 2.9 倍与 2.4 倍. 这一优势源于 Split-LSM-Tree 的动态分裂机制: 在数据加载过程中, 系统被划分为多个子树, 每个子树仅需管理较小规模的数据集, 从而显著减少了 compaction 引发的 I/O 开销并提升了写入效率.

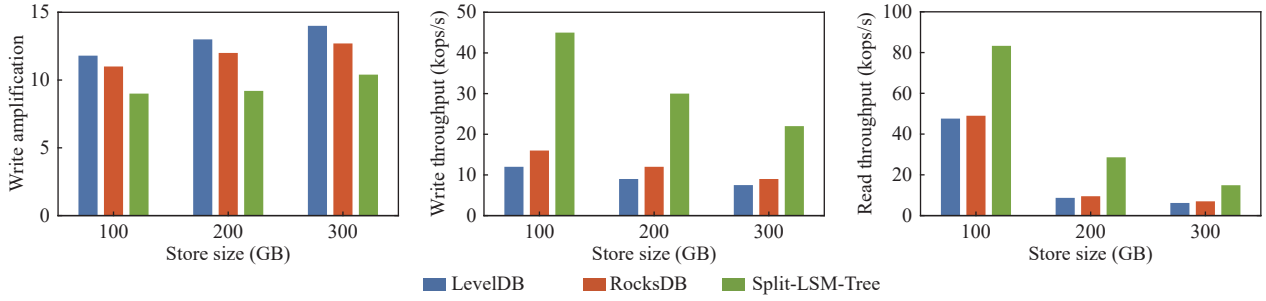

图 5 不同数据量下系统的读写性能

在读性能评估方面, 我们在初始数据加载阶段结束后, 针对数据集总量的 10% 执行了均匀随机读取测试. 实验结果 (见图 5) 表明, Split-LSM-Tree 在读吞吐量方面同样保持领先, 当存储规模为 300 GB 时, 其读性能分别达到了 LevelDB 和 RocksDB 的 2.4 倍与 2.1 倍. 这一提升得益于子树架构下层级深度的减小,

进而减少了读操作在查找过程中需要访问的文件数量, 从而降低了 I/O 操作次数并抑制了读放大效应.

4.4 YCSB 性能测试

YCSB (Yahoo! cloud serving benchmark)[20]是评估键值存储性能的行业标准框架, 其包含的多种工作负载可模拟不同的真实应用场景中的读写模式 (具体配置见表 1). 我们使用这些工作负载对 Split-LSM-Tree 与 LevelDB、RocksDB 和 WiscKey 进行对比测试. 实验使用预先加载的 100 GB 数据集, 其键大小为 16 字节, 值大小为 1 KB, YCSB 的数据访问模式遵循均匀分布, 每个工作负载执行 1 000 万次数据操作.

表 1 YCSB 主要测试负载

Workload	Characteristics
A (update heavy)	50% updates, 50% reads
B (read mostly)	5% updates, 95% reads
C (read only)	100% reads
D (read latest)	5% inserts, 95% reads
F (read-modify-write)	50% read-modify-write, 50% reads

图 6 展示了各系统在不同 YCSB 工作负载下的吞吐量表现. 实验结果表明, 无论是在读密集型还是写密集型场景下, Split-LSM-Tree 均保持性能领先. 在以读操作为主的工作负载 B、C、D 中, 其吞吐量较 LevelDB 分别提升了 86%、82% 与 106%, 较 RocksDB 提升了 72%、77% 与 88%, 较 WiscKey 提升了 9%、4%、5%, 这主要得益于更浅的子树层级结构有效降低了读放大. 在写密集型工作负载 A 与 F 中, Split-LSM-Tree 同样展现出明显的优势, 其吞吐量较 LevelDB 分别提升了 29% 与 73%, 较 RocksDB 提升了 7% 与 57%, 较 WiscKey 提升了 6% 与 30%. 尽管这些工作负载中的写入数据量较小 (约 5 GB) 导致系统间写放大差异不明显, 但 Split-LSM-Tree 较高的读取效率使其在 A、F 工作负载中仍能实现更优的整体性能.

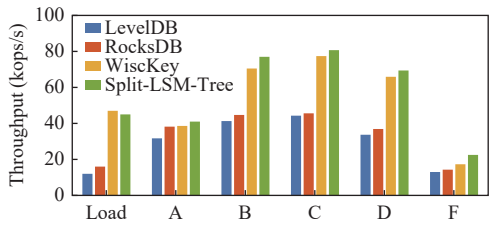

图 6 YCSB 性能测试实验结果

4.5 与静态分区架构的性能对比

为了验证动态分裂机制带来的性能收益, 本节对比评估了 Split-LSM-Tree 与静态分区森林架构的性能表现. 实验数据集规模为 300 GB, 首先测试数据随机写入的性能, 在数据加载完成后, 针对占数据集总量 10% 的数据执行均匀随机读取以测试读性能. 实验结果如图 7 所示, Split-LSM-Tree 的写性能和读性能分别比静态分区架构提高了 89% 和 62%, 这主要得益于其动态分裂机制和划分键选择算法能够利用已有的数据特征来选择划分点, 与静态分区方案相比, 这种自适应设计使得各个子树的负载分配更加均衡.

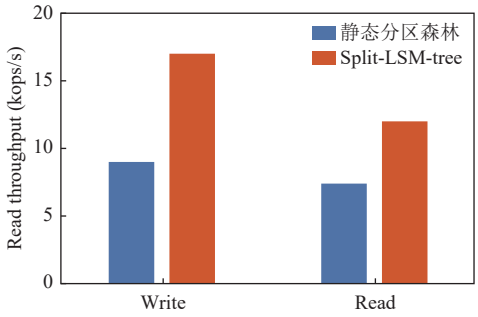

图 7 静态分区森林与 Split-LSM-Tree 的性能对比

4.6 CPU 与内存开销分析

为评估 Split-LSM-Tree 在数据加载过程中因子树专属 compaction 线程而引入的 CPU 开销, 我们向系统中写入 100 GB 的数据, 同时测量写入过程中各系统的平均 CPU 利用率. 实验结果如表 2 所示, Split-LSM-Tree 在数据加载过程中的平均 CPU 利用率为 52.6%, 略高于 LevelDB 与 RocksDB. 这一微小差异证实其计算开销处于可控范围, 原因在于每个子树的浅层级结构减少了跨层数据移动与 compaction 频率, 从而保证了高效的 CPU 利用率.

表 2 系统 CPU 开销 (%)

系统	LevelDB	RocksDB	Split-LSM-Tree
Average CPU overhead	50.97	48.3	52.6

此外, Split-LSM-Tree 引入的额外内存开销可以忽略不计, 在测试中我们将其内存上限约束至与传统 LSM-tree 相同的水平. 在实际部署中, 用户可根据需求灵活配置内存限额, 更高的内存限额将带来更显著的吞吐量提升.

5 相关工作

随着基于 LSM-tree 的存储系统在数据密集型应用中的广泛部署, 针对其性能瓶颈的优化已成为学术

界和工业界的研究热点. 现有工作主要聚焦于降低系统的读写放大, 提升整体性能, 具体可归纳为以下几种主流方向.

● 键值分离方向: WiscKey[11]提出了键值分离架构, 将值存储于独立的值日志中, LSM-tree 中仅保留键和值指针, 该设计有效消除了 compaction 过程中对值进行重写的开销. Scavenger[21]提出了一种 I/O 高效的垃圾回收机制与基于补偿大小的空间感知合并策略. 该方案有效控制了分离架构下的空间放大, 在提升写入性能的同时实现了更优的系统时空权衡. DumpKV[22]通过轻量级机器学习模型预测键的剩余生命周期, 在垃圾回收时对相似生命周期的数据进行聚合存储, 显著降低了数据重复搬运带来的写放大.

● 哈希优化方向: LSM-trie[12]基于哈希将数据组织成前缀树结构, 在 compaction 执行时, 通过将 SSTable 直接追加至下层以避免合并操作, 从而降低写放大. UniKV[13]融合了哈希索引与 LSM-tree 架构, 利用数据局部性特征, 对不同类型的键值对进行分类管理, 通过将新插入的键值对暂存于哈希索引并定期批量合并至底层 LSM-tree 的方式提升读写性能. Cai 等人[23]提出了一种基于可扩展哈希的 LSM-tree, 通过局部敏感哈希和新型 HTable 结构替代传统 SSTable, 提升了系统缓存利用率.

● 宽松排序方向: PebblesDB[14]放松了数据的严格有序性约束, 将一个层级划分为多个段, 且允许段内 SSTable 存在范围重叠, 该设计使其在 compaction 时仅需将 SSTable 追加至下层的对应段中, 从而降低了写放大. TRIAD[24]整合了针对内存管理、压缩策略与日志机制的三重优化, 减少了后台操作的 I/O 开销. FGKV[25]提出的 LSPM-tree 通过引入补丁实现细粒度合并, 仅局部重写更新数据以维持宽松有序, 减少了无关数据的重复搬运与写放大.

● I/O 调度方向: ADOC[26]将 LSM-tree 的写入停滞归因于数据溢出现象, 并设计了自适应参数调优框架, 通过动态调节数据累积速率来预防溢出, 从而确保系统的高性能运行. EcoTune[27]将 compaction 视为资源投资, 利用动态规划算法实时求解最优 compaction 策略, 在 compaction 开销与平均查询吞吐量间实现了良好的动态平衡. SuccinctKV[28]提出了一种基于扫描的合并机制, 将范围查询与合并操作中的归并排序开销相融合, 并结合热点感知的 I/O 统一调度器, 显著降低了混合负载下的 CPU 开销与突发 I/O 竞争.

需要指出的是, 上述解决方案均基于传统的单体树形基础架构. 随着键值存储规模的持续扩张, 单体架构中较深的层级结构仍不可避免地会引发显著的读写放大问题. Split-LSM-Tree 采用的森林架构正是针对这一根本性问题提出的解决方案.

6 总结

针对传统 LSM-tree 架构因层级过深导致的读写放大问题, 本文提出并实现了一种基于森林架构的键值存储引擎 Split-LSM-Tree, 并为其设计了自适应分裂触发机制、非阻塞分裂算法及相关 I/O 优化策略, 使其能高效地将单体 LSM-tree 动态分裂为子树森林, 有效缓解了传统架构的读写放大问题. 实验结果表明, 与传统单体架构相比, Split-LSM-Tree 在读写性能方面均实现了显著提升.

参考文献

1 DeCandia G, Hastorun D, Jampani M, *et al*. Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review, 2007, 41(6): 205–220. [doi: 10.1145/1323293.1294281]

2 Armstrong TG, Ponnekanti V, Borthakur D, *et al*. Linkbench: A database benchmark based on the Facebook social graph. Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. New York: ACM, 2013. 1185–1196. [doi: 10.1145/2463676.2465296]

3 Beaver D, Kumar S, Li H C, *et al*. Finding a needle in haystack: Facebook's photo storage. Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation. Vancouver: USENIX Association, 2010.

4 Lakshman A, Malik P. Cassandra: A decentralized structured storage system. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35–40. [doi: 10.1145/1773912.1773922]

5 Dong SY, Kryczka A, Jin YQ, *et al*. RocksDB: Evolution of development priorities in a key-value store serving large-scale applications. ACM Transactions on Storage (TOS), 2021, 17(4): 26. [doi: 10.1145/3483840]

6 Cassandra. https://github.com/apache/cassandra. (2026-03-05)[2026-03-10].

7 RocksDB. http://rocksdb.org/. (2026-03-07)[2026-03-10].

8 TiKV. https://tikv.org/. (2026-03-09)[2026-03-10].

9 WiredTiger. https://github.com/wiredtiger. (2026-03-09)

[2026-03-10].

10 O'Neil P, Cheng E, Gawlick D, *et al*. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996, 33(4): 351–385. [doi: 10.1007/s002360050048]

11 Lu LY, Pillai TS, Gopalakrishnan H, *et al*. WiscKey: Separating keys from values in SSD-conscious storage. ACM Transactions on Storage, 2017, 13(1): 5. [doi: 10.1145/3033273]

12 Wu XB, Xu YH, Shao ZL, *et al*. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. Proceedings of the 2015 USENIX Annual Technical Conference. Santa Clara: USENIX Association, 2015. 71–82.

13 Zhang Q, Li YK, Lee PPC, *et al*. UniKV: Toward high-performance and scalable KV storage in mixed workloads via unified indexing. Proceedings of the 36th IEEE International Conference on Data Engineering. Dallas: IEEE, 2020. 313–324. [doi: 10.1109/ICDE48307.2020.00034]

14 Raju P, Kadekodi R, Chidambaram V, *et al*. PebblesDB: Building key-value stores using fragmented log-structured merge trees. Proceedings of the 26th Symposium on Operating Systems Principles. Shanghai: ACM, 2017. 497–514. [doi: 10.1145/3132747.3132765]

15 Ren K, Zheng Q, Arulraj J, *et al*. SlimDB: A space-efficient key-value storage engine for semi-sorted data. Proceedings of the VLDB Endowment, 2017, 10(13): 2037–2048. [doi: 10.14778/3151106.3151108]

16 LevelDB. https://code.google.com/p/leveldb. (2021-02-24).

17 Li YK, Tian CJ, Guo F, *et al*. ElasticBF: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. Proceedings of the 2019 USENIX Annual Technical Conference. Renton: USENIX Association, 2019. 739–752.

18 LogsDB. https://logdevice.io/docs/LogsDB.html. (2022-01-08)[2026-03-10].

19 Sandeep. How we use RocksDB at rockset. https://medium.com/rocksetcloud/how-we-use-rocksdb-at-rockset-c500876fd6d4. (2019-09-05)[2026-03-10].

20 Cooper BF, Silberstein A, Tam E, *et al*. Benchmarking cloud serving systems with YCSB. Proceedings of the 1st ACM Symposium on Cloud Computing. Indianapolis: ACM, 2010. 143–154. [doi: 10.1145/1807128.1807152]

21 Zhang JS, Wang F, Qiu S, *et al*. Scavenger: Better space-time trade-offs for key-value separated LSM-trees. Proceedings of the 40th IEEE International Conference on Data Engineering. Utrecht: IEEE, 2024. 4072–4085. [doi: 10.1109/ICDE60146.2024.00312]

22 Zhuang ZT, Zeng XQ, Chen ZG. DumpKV: Learning based lifetime aware garbage collection for key value separation in LSM-tree. Proceedings of the VLDB Endowment, 2024, 18(4): 1223–1236. [doi: 10.14778/3717755.3717778]

23 Cai T, Huang QJ, Dai JF, *et al*. A cache friendly LSM tree based on extendible hash. Concurrency and Computation: Practice and Experience, 2026, 38(1): e70522. [doi: 10.1002/cpe.70522]

24 Balmau O, Didona D, Guerraoui R, *et al*. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. Proceedings of the 2017 USENIX Annual Technical Conference. Santa Clara: USENIX Association, 2017. 363–375.

25 Sun H, Chen GZ, Yue YL, *et al*. Improving LSM-tree based key-value stores with fine-grained compaction mechanism. IEEE Transactions on Cloud Computing, 2023, 11(4): 3778–3796. [doi: 10.1109/TCC.2023.3329646]

26 Yu JH, Noh SH, Choi YR, *et al*. ADOC: Automatically harmonizing dataflow between components in log-structured key-value stores for improved performance. Proceedings of the 21st USENIX Conference on File and Storage Technologies. Santa Clara: USENIX Association, 2023. 65–80.

27 Wang HR, Qiu JS, Yuan FZ, *et al*. Rethinking the compaction policies in LSM-trees. Proceedings of the ACM on Management of Data, 2025, 3(3): 207. [doi: 10.1145/3725344]

28 Zhang YN, Yang S, Hu HQ, *et al*. SuccinctKV: A CPU-efficient LSM-tree based KV store with scan-based compaction. ACM Transactions on Architecture and Code Optimization, 2024, 21(4): 90. [doi: 10.1145/3695873]

(校对责编: 张重毅)