LSM-KV 项目报告

2023 年 5 月 1 日

1 背景介绍

Log-Structured Merge-Tree[OCGO96]（即 LSM 树）是一种面向磁盘、分层存储的数据结构。该数据结构被运用于 Bigtable[CDG+08]、LevelDB、RocksDB 等 NoSQL 数据库中。

对于一个典型的 LSM 树，其内存数据被称为 MemTable，跳表是一种常用的 MemTable 实现方法；磁盘中的数据文件被称为 SSTable，存储在不同的层级（Level）中。当某个层级的 SSTable 数量超过该层级限制，则触发 Compaction 将相邻层级中重叠的 SSTable 合并写入到下一层级，直到所有层级的 SSTable 数量满足限制。

相比 B 树、B+ 树等传统的面向磁盘数据结构，LSM 树的写入操作与顺序读取操作的延迟较小。其主要原因是：

- LSM 树在插入数据时无需像 B 树、B+ 树频繁通过磁盘 IO 更新索引信息，而是在 MemTable 或者某个层级已满时集中做磁盘操作，减少了磁盘 IO 系统调用的次数，提升了写入效率。
- LSM 树的每个 SSTable 都按键值顺序存储数据，且 LSM 树的 Compaction 机制减少了 SSTable 的键值重叠，这使得键值相邻的数据有大概率处在同一个 SSTable 中，提高了顺序读取效率。

2 代码实现

本文中实现的 LSM 树使用 C++17 结合模板编写，能够灵活切换索引缓存方法、MemTable 数据结构（默认为 SkipList，支持其他数据结构）、Level 配置、键值（Key）与值（Value）的类型、Value 的 IO 方法（支持自定义序列化或压缩算法），且不引入额外的运行时开销。

LSM 树实现了 Get、Put、Delete、Scan、Reset 方法，其中 Scan 的实现运用了二叉堆。

此外，代码实现中使用一个 LRU Cache 来缓存 std::ifstream 对象，显著减少了打开文件的系统调用开销。LRU Cache 的大小在运行时指定。

2.1 样例

代码 1定义了一个项目指定的标准 LSM 树，该 LSM 树使用 Murmur3 作为 Bloom Filter 的哈希算法。代码 2定义了一个使用 Snappy 压缩算法压缩 Value 的 LSM 树，且该 LSM 树不缓存键值。

```
#include <lsm/kv.hpp>
#include "MurmurHash3.h"

template <typename Key> struct Murmur3BloomHasher {
    template <std::size_t Bits, typename Array> inline static void Insert(Array &array, const Key
↪   &key) {
        uint32_t hashes[4];
        MurmurHash3_x64_128(&key, sizeof(Key), 1, hashes);
        array[hashes[0] % Bits] = true;
```

```
        array[hashes[1] % Bits] = true;
        array[hashes[2] % Bits] = true;
        array[hashes[3] % Bits] = true;
    }
    template <std::size_t Bits, typename Array> inline static bool Exist(const Array &array, const
↪   Key &key) {
        uint32_t hashes[4];
        MurmurHash3_x64_128(&key, sizeof(Key), 1, hashes);
        return array[hashes[0] % Bits] && array[hashes[1] % Bits] && array[hashes[2] % Bits] &&
↪   array[hashes[3] % Bits];
    }
};

template <typename Key> struct StandardTrait : public lsm::KVDefaultTrait<Key, std::string> {
    using Compare = std::less<Key>;
    using Container = lsm::SkipList<Key, lsm::KVMemValue<std::string>, Compare,
↪   std::default_random_engine, 1, 2, 32>;
    using KeyFile = lsm::KVCachedBloomKeyFile<Key, StandardTrait, lsm::Bloom<Key, 10240 * 8,
↪   Murmur3BloomHasher<Key>>>;
    constexpr static lsm::size_type kMaxFileSize = 2 * 1024 * 1024;

    constexpr static lsm::KVLevelConfig kLevelConfigs[] = {
        {2, lsm::KVLevelType::kTiering},
        {4, lsm::KVLevelType::kLeveling},
        {8, lsm::KVLevelType::kLeveling},
        {16, lsm::KVLevelType::kLeveling},
        {32, lsm::KVLevelType::kLeveling}
    };
};

using StandardKV = lsm::KV<uint64_t, std::string, StandardTrait<uint64_t>>;
```

代码 1: 项目指定的标准 LSM 树定义

```
#include <lsm/kv.hpp>
#include <snappy.h>

struct SnappyStringIO {
    inline static lsm::size_type GetSize(const std::string &str) {
        std::string compressed;
        snappy::Compress(str.data(), str.length(), &compressed);
        return compressed.length();
    }
    template <typename Stream> inline static void Write(Stream &ostr, const std::string &str) {
        std::string compressed;
        snappy::Compress(str.data(), str.length(), &compressed);
        ostr.write(compressed.data(), compressed.length());
    }
    template <typename Stream> inline static std::string Read(Stream &istr, lsm::size_type length) {
        std::string compressed, str;
        compressed.resize(length);
        istr.read(compressed.data(), length);
        snappy::Uncompress(compressed.data(), length, &str);
        return str;
    }
};

template <typename Key> struct CompressedTrait : public lsm::KVDefaultTrait<Key, std::string> {
    using KeyFile = lsm::KVUncachedKeyFile<Key, CompressedTrait>;
    using ValueIO = SnappyStringIO;
};

using CompressedKV = lsm::KV<uint64_t, std::string, CompressedTrait<uint64_t>>;
```

代码 2: 使用 Snappy 压缩算法且不缓存键值的 LSM 树定义

3 性能测试

3.1 预期结果

根据 LSM 树的性质，在 Get、Put、Delete 三种操作中，只有 Put 与 Delete 操作可能触发磁盘写入和 Compaction；同时 Get 与 Put 涉及的磁盘 IO 数据量较大，Delete 至多只需写入一个不带 Value 的 Key。因此常规测试中的操作延迟从大到小应为 Put、Get、Delete，吞吐量则相反。

相比不缓存键值数据，理论上使用 Bloom Filter 能够在命中率较低时减少无效文件访问次数以降低 Get 延迟，但会增加高命中率时 Get 操作的延迟；索引缓存极大减少了 Get 操作的 IO 次数，Get 操作的延迟相比不缓存索引理论上会极大减少；缓存 Bloom Filter 和索引理论上不会比仅缓存索引有太大效率提升，因为内存中的二分查找效率已经足够高。

Put 操作中 Compaction 的开销与当前 SSTable 的数量有关。SSTable 数量占满的层数越多，则 Compaction 需要处理的平均层数越多，Put 的吞吐量越低。

Level 配置对吞吐量的影响有两个方面。首先，Tiering 层与较小的文件数目限制会减少 SSTable 的重叠，提升 Get 与 Scan 的吞吐量；然而这会增加 Put 操作中 Compaction 的频率，降低 Put 的吞吐量。

3.2 常规分析

本文分别测试了 2 KiB、4 KiB、6 KiB、8 KiB 数据大小的 Put、Get、Delete 操作的延迟与吞吐量，其中对 Get 操作分别测试了顺序和乱序的延迟与吞吐量，测试结果见图 1和图 2。

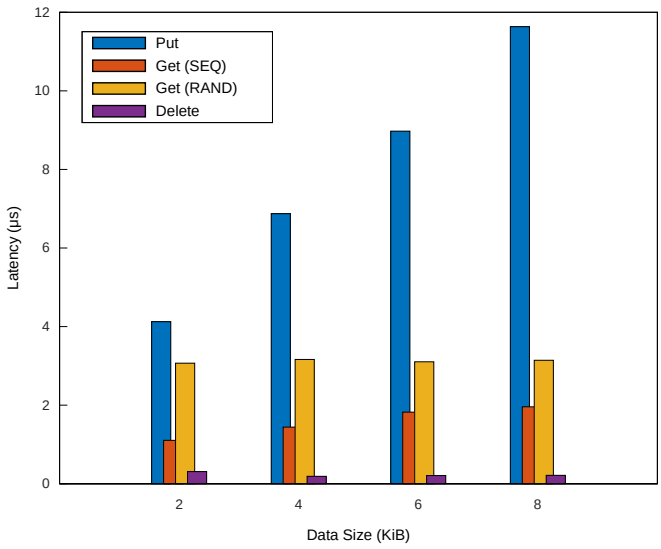

图 1: 标准 LSM 树在不同数据大小下 Put、Get、Delete 操作的平均延迟

测试结果与预期结果一致。操作延迟从大到小为 Put、Get、Delete，吞吐量则相反；且顺序 Get 操作的延迟小于乱序 Get 操作。

此外，Put 操作的延迟与数据大小基本呈现性关系，推测原因为数据越大，Compaction 的频率越高，导致延迟增加。顺序 Get 操作的延迟也与数据大小正相关，推测原因是数据越大，读取时切换文件的频率越高。

乱序 Get 与 Delete 操作的延迟与数据大小基本无关。对于乱序 Get，这是由于乱序时切换文件的频率基本与数据大小无关；对于 Delete，这是由于其写入数据量小且基本恒定。

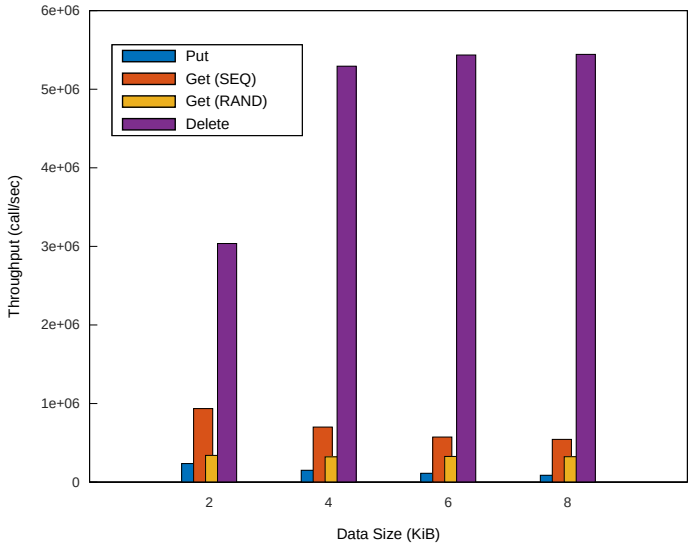

图 2: 标准 LSM 树在不同数据大小下 Put、Get、Delete 操作的吞吐量

3.3 索引缓存与 Bloom Filter 的效果测试

本文测试了不缓存、只使用 Bloom Filter、只缓存索引、缓存索引 +Bloom Filter 四种情况下 Get 操作的平均延迟。总共进行了两组测试，命中率分别为 100% 和 50%，其中 50% 的命中率通过 Put 偶数键值并在连续键值区间内 Get 实现，测试结果见图 3和图 4。

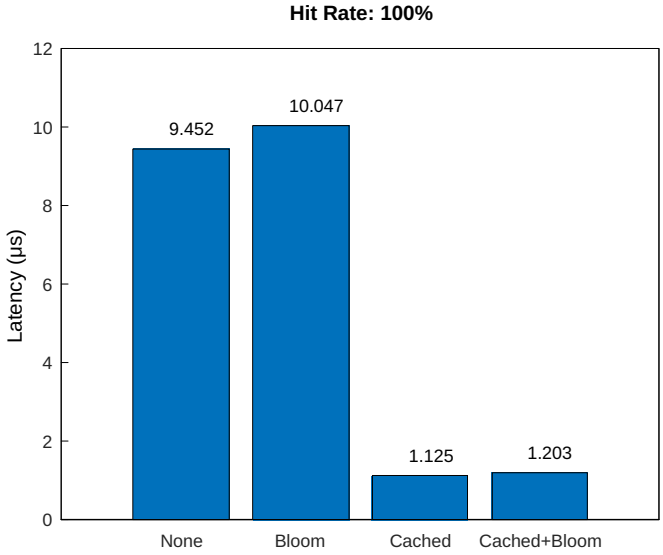

图 3: 不缓存、只使用 Bloom Filter、只缓存索引、缓存索引和 Bloom Filter 的 LSM 树在 2 KiB 数据大小、100% 命中率下 Get 操作的平均延迟

测试结果与预期基本一致。

- 相比不缓存，使用 Bloom Filter 在命中率为 50% 时降低了 Get 延迟，但在命中率为 100% 时增加了延迟
- 缓存索引的 Get 延迟在任何情况远小于不缓存索引
- 缓存索引 +Bloom Filter 相比仅缓存索引 Get 延迟略微增大，因为 Bloom Filter 的计算量较大且不足以抵消节省的内存二分查找开销

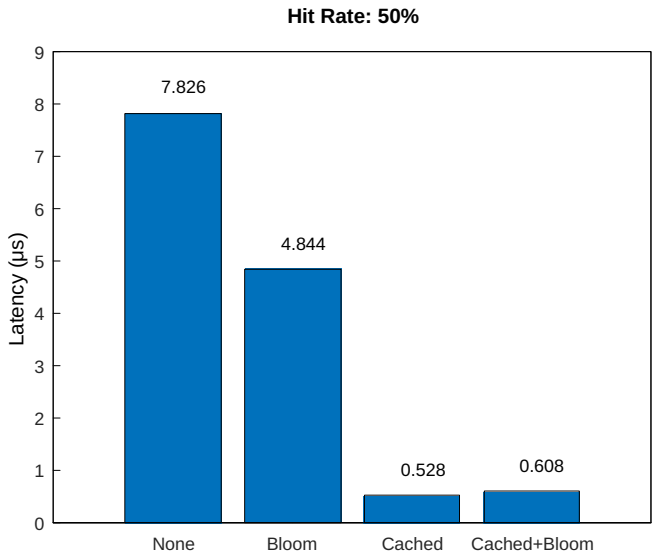

图 4: 不缓存、只使用 Bloom Filter、只缓存索引、缓存索引和 Bloom Filter 的 LSM 树在 2 KiB 数据大小、50% 命中率下 Get 操作的平均延迟

3.4 Compaction 的影响

本文测试了不断插入数据时 Put 操作的吞吐量变化。测试中每次 Put 插入的值大小为 8 KiB，共执行 16384 次 Put，总共插入 128 MiB 的值。由于数据方差过大，绘图前进行了窗口大小为 2048 的移动平均，实验结果见图 5。

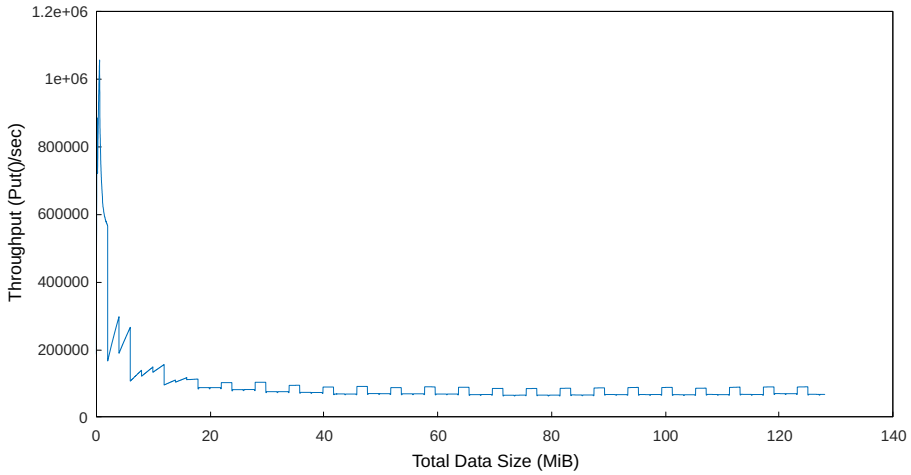

图 5: Put 操作吞吐量随累计数据量的变化曲线

测试结果与预期基本一致。当没有触发 Compaction 机制时，Put 操作吞吐量最大；随后第一层填满（约 4 MiB 的值），吞吐量骤降。在总插入大小低于 20 MiB 时，吞吐量的降低呈现阶梯状，之后则趋于稳定。这可以解释为填满标准 LSM 树的前几层时 Compaction 的文件合并次数增长速度较快，因此吞吐量阶梯状降低；随后由于 Leveling 层的键值范围不重叠，每次 Compaction 的文件合并次数基本维持稳定，于是吞吐量趋于稳定。

3.5 Level 配置的影响

在 LSM 树的 Compaction 机制中，Tiering 层在文件个数超出界限时需要将层级中所有文件与下一个层级合并，当 Tiering 层界限过大时，其开销远大于 Leveling 层。因此 Tiering 层通常在最上层且允许的文件数目最小，其他层级均为 Leveling 层。又由于 LSM 树在每相邻两个层级的最大文件数目比例相同时写入效率最高 [OCGO96]。因此本文主要测试分析最上层的 Tiering 层级的允许文件数量 TieringSize 以及每相邻两个层级的最大文件数目比例 SizeRatio 对 LSM 树整体效率的影响。

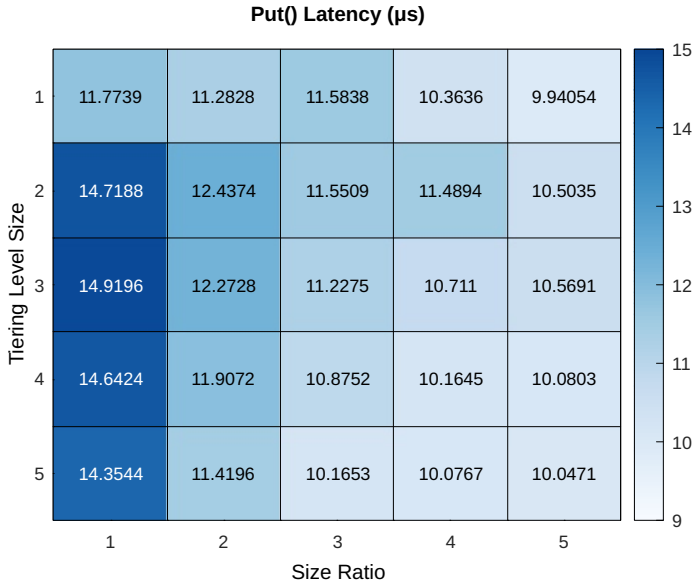

图 6: 指定 TieringSize 和 SizeRatio 的 LSM 树 Put 操作的平均延迟

测试中使用的层级数量为 5 的 LSM 树，分别在 $[1,5]$ 区间内调整 TieringSize 和 SizeRatio，测试 Put 与顺序 Get 操作的平均延迟。测试中每次写入的数据大小为 8 KiB，总共写入 128 MiB 的值。由于 Delete 操作的延迟显著低于 Put 与 Get，对 LSM 树性能构成的影响有限，不进行测试。测试结果见图 6、图 7。

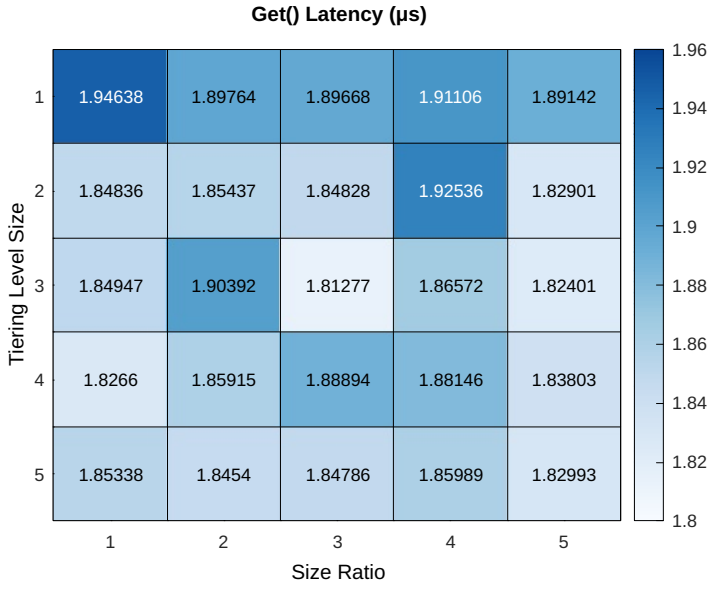

图 7: 指定 TieringSize 和 SizeRatio 的 LSM 树 Get 操作的平均延迟

测试结果与预期部分一致。由图 7可见，当 TieringSize 大于 2 时，Get 操作延迟相比 TieringSize 小于 2 时显著减小；继续增大 TieringSize 对 Get 延迟的降低不显著。

根据图 6，Put 的延迟随着 TieringSize 的增大先增加后减小，这与预期结果由一定差别，可能的原

因是当 TieringSize 较大时，每个层级的容量相应增大，抵消了合并较大的 Tiering 层的开销；同时 Put 的延迟与 SizeRatio 则呈现较强的负相关性，这可以解释为较大的 SizeRatio 增大了层级容量，减少了 Compaction 的频率。

综合以上测试结果，可以得出结论：选择较大的 SizeRatio 并选择大于 2 的 TieringSize 能同时降低 Put 和 Get 操作的延迟，然而这样显然也会导致整个数据库的总存储大小膨胀。因此本文给出的指导建议为：在存储空间允许的范围内尽可能提升 SizeRatio，同时设置大于 2 的 TieringSize。

4 结论

LSM-KV 项目使用 C++17 结合模版实现了可定制的 LSM 树，并对其进行了性能测试。

常规分析、缓存与 Bloom Filter 测试、研究 Compaction 影响的实验结果基本符合理论预期；Level 配置实验的结果能够用理论解释，且可以对 Level 配置给出指导建议。

总体而言，LSM-KV 项目的代码实现与实验较为成功。

5 致谢

本项目 LSM 树核心代码中的 LRU Cache 实现借鉴了 https://github.com/lamerman/cpp-lru-cache，除此之外均为独立编写，没有借鉴任何开源项目、论坛、博客。

本文的背景介绍部分参考了 https://zhuanlan.zhihu.com/p/415799237。

本文中缓存与 Bloom Filter 测试的思路受到了 https://medium.com/swlh/log-structured-merge-trees-9c8e2bea89e8 的启发。

6 其他和建议

我认为项目要求应当鼓励同学学习使用最现代的 C++，而非限制在 C++14。

C++17 相比 C++14 增加了 filesystem、constexpr if、更方便的 type traits、结构化绑定等特性。C++20 进一步引进了更多现代特性，例如 concept、span 等。

本项目使用 C++17 实现的是为了简化代码的编写、提升可读性与运行效率。这不代表本项目无法使用 C++14 实现，只是需要写更多没有思维价值的代码。

C++ 的最新特性很可能会在同学们的未来工作中使用，课程应当支持同学自学、运用。

参考文献

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[OCGO96] Patrick O' Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O' Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33:351–385, 1996.