LSMKV 键值存储结构

朱涵 522031910213

1 背景介绍

日志化结构合并树 (Log-Structured Merge-Tree)[1] 是一种分层、有序、面向磁盘的数据结构，其核心思想是充分利用磁盘批量的顺序写远比随机写高效的特性，放弃部分读效率换取最大化的写操作效率。LSM 树是依据磁盘的存储特性，用以下的方式实现的: 假定内存足够，不需要每次有数据更新就将其写入磁盘，而是先将最新的数据驻留在内存中，等数据量积累到足够多之后，再使用归并排序的方式将内存中的数据与磁盘中的数据合并，批量追加到磁盘。合并过程因为所有待排序的数据都是有序的，所以可以通过归并排序快速合并。LSM 树结构的文件结构策略目前已经应用在多种 NoSQL 数据库，如 Hbase，Cassandra，Leveldb，RocksDB，MongoDB，TiDB，SQLite 等 [2]。

尽管 LSM 树结构已经十分完善，但仍有其不足之处。2016 年，Lanyue Lu 等人发表的论文 Wisckey 针对 LSM Tree 长期被人诟病的写放大（Write Amplification）问题提出了相应优化方法：**键值分离**，即通过把占用空间较小的键索引存储在 SSTable 文件中，把键值对存储在 vLog 文件中来缓解问题，必要时还可以把索引内容缓存在内存里减少读写开销。

2 代码实现

2.1 项目简介

本项目基于 LSM 树与键值分离技术实现了 **LSMKV 键值存储系统**，支持 GET、PUT、SCAN、DEL、RESET 操作，并且通过 GC 函数定期对 vLog 文件进行垃圾回收，通过 COMPACTION 函数对 SSTable 文件进行分层存储。LSM 树的结构被分为 **Memtable**（内存表）、**SSTable**（磁盘中的索引文件）以及 **vLog**（磁盘中的键值对存储文件）三种基本结构。其中 Memtable 通过跳表结构实现，SSTable 和 vLog 在磁盘中通过一定格式的 entry（条目）来存储和读取，在缓存中通过普通的数组结构来实现。

本文中实现的 LSMKV 结构是使用 C++20 类进行编写的，使用 fstream 库进行文件的读写，支持灵活切换键值类型、SSTable 层数、最大文件大小、缓存策略、Bloomfilter 大小等配置。

2.2 样例

如下图是 LSMKV 系统的**使用样例**，定义一个 KVStore 结构并进行了基本操作，所有配置类都封装在 config.h 文件中，只暴露了最简单的操作接口供使用。

```
void example()
{
    KVStore store("./data");
    store.reset();
    store.put(1, "1");
    store.get(1);
    scan_result_t result;
    store.scan(1, 1, result);
    store.del(1);
    store.gc(16 * MB);
}
```

图 1: LSMKV 系统样例代码

3 测试

本项目对于 LSMKV 键值存储系统主要进行了两部分的测试：**正确性测试**及**性能测试分析**。其中正确性测试较为简单，在这里只对测试内容作简单的介绍，不作结果的展示和分析，读者根据文档可以自行编译正确性测试程序并运行来校验结果。

3.1 正确性测试

此部分包含三个部分的测试：

1. **基本测试**：对于基本操作进行测试，数据量很小，只检验 Memtable 的实现正确性；
2. **复杂测试**：对于基本操作进行测试，数据量较大，检验 SSTable 以及 vLog 结构的正确性，并且检验垃圾回收（gc）功能；
3. **持久性测试**：对于 LSMKV 系统的持久性进行测试，主要通过手动终止程序并重新访问来检查键值存储的持久化是否正确。

3.2 性能测试

此部分包含四个部分的测试，主要对于 LSMKV 系统的基本性能以及不同参数配置下的影响进行测试和分析。在第一部分的基本性能测试中，如果没有说明，默认的配置是：**缓存策略 3**（缓存索引与 Bloomfilter），**BF 大小**为 8KB。

3.2.1 预期结果

- 对于 LSM 树的基本操作性能，推测时延顺序从大到小为 DELETE、PUT、GET，因为测试中的情景下所有操作都会命中，而 PUT 操作在较大的数据量下涉及到合并（compaction）操作，时延较大于 GET，DELETE 操作则由一次 GET 和一次 PUT 组成，显然应该最大。而 SCAN 的时延应该与扫描的数据范围成正比，因此在数据量变化但扫描范围不变的情况下，应该不会有太大的差异。对于数据量变化带来的性能影响，DELETE、PUT、GET 三者的时延应该与数据量成线性，而 SCAN 则不受影响；

- 对于不同缓存策略下的 GET 性能，推测不缓存任何索引的策略的时延是最大的，而且远远大于另外两种策略。至于只缓存索引和缓存索引与 Bloomfiter 两种策略，在理论上后者的时间复杂度更小，但是实际上可能会因为内存中的二分查找效率足够高而使得没有什么差异；

- 对于 Compaction 开销导致的 PUT 性能变化，推测在数据量累积到某一个值后 PUT 的时延显著上升，原因是在 Level0 文件超出阈值之后的 PUT 需要进行合并操作，大大提升了时延。

- 对于不同 Bloomfilter 大小下的 GET、PUT 性能变化，推测 BF 的大小越大，PUT 的平均时延越大，而 GET 的平均时延越小。原因是 BF 的大小越大，SSTable 中索引所占的空间就越小，也就间接导致 compaction 的操作次数变多；而同时 BF 的大小越大，又会导致哈希的冲突率越大，间接导致了 GET 时的错报率变大，也就使 GET 的时延越大（访问了更多的索引）。但是实际上可能 GET 的性能并不会有显著变化，甚至还会变大，因为我们采用了 sst 缓存的策略，对于索引访问的开销是访问内存的开销，相比于因为 BF 过大而间接导致 sst 文件更多的开销，可能没那么明显。

3.2.2 常规分析

第一部分测试的主要内容为：在数据量（k）为 20000 到 80000 之间对 LSMKV 的基本操作进行了测试，顺序插入 k 个字符串（内容为"test!"），顺序读取 k 个键值对，扫描 k 次（扫描范围为 1-10），最后顺序删除 k 个键值对。测试数据的 Value 大小为 5 字节，Key 的范围从 1 到数据量 k。

测试结果如下图。可以看到测试结果与预期结果一致，操作延迟从大到小为 DELETE、PUT、GET，吞吐量则相反，三者的时延也与数据量成正相关，推测原因是数据量越大，需要访问的 sst 文件就越多。

另外，SCAN 操作的时延与预期一样，基本与数据量无关，这是因为扫描缓存中的索引的开销很低，主要开销在于对于 vLog 文件的访问，而数据量大小与此无关，扫描范围越大导致命中的索引越多才会使得访问变多。

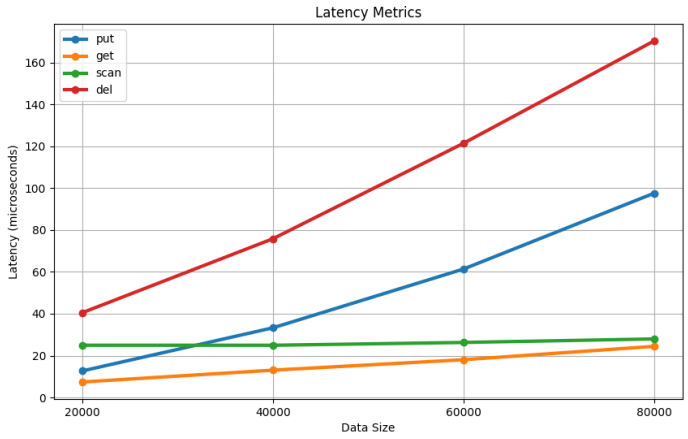

图 2: LSMKV 系统在不同数据量下的基本操作平均时延

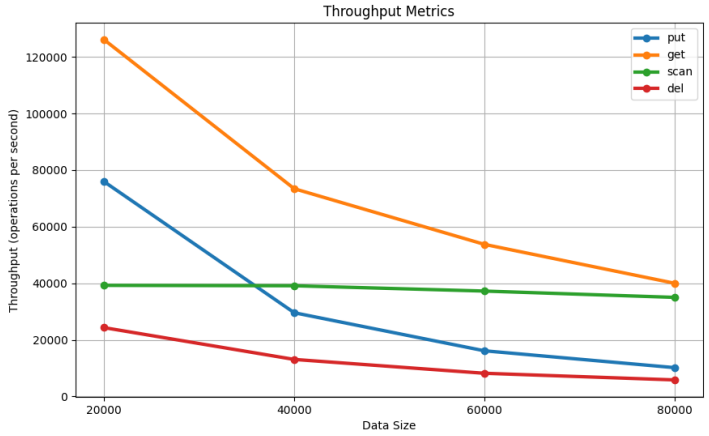

图 3: LSMKV 系统在不同数据量下的基本操作吞吐量

3.2.3 索引缓存与 Bloom Filter 的效果测试

第二部分测试的主要内容为：在三种缓存策略（1. 不缓存任何信息 2. 只缓存索引 3. 缓存索引以及 BF）下进行数据量为 1000 的 PUT 和 GET 操作，并计算 GET 操作的平均时延和吞吐量。

测试结果如下图。测试结果与预期基本一致，第一种缓存策略的时延远大于另外两种策略，而是否缓存 Bloomfilter 对于 GET 的性能基本没有影响，推测原因是虽然 Bloomfilter 可以减少查询索引，但是哈希的计算开销比较大，不足以抵消节省的内存中二分查找开销。

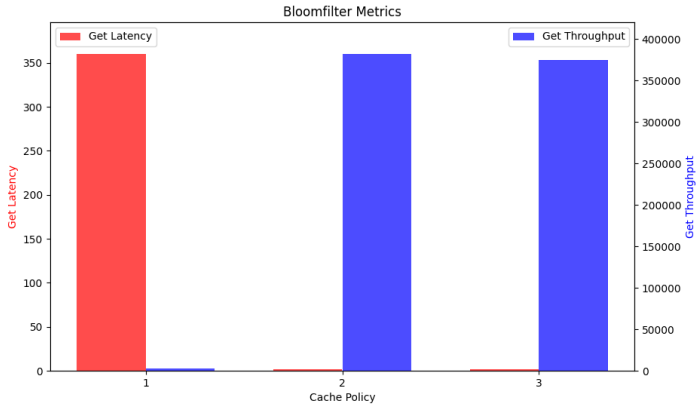

图 4: LSMKV 系统在不同缓存策略下的 GET 性能

3.2.4 Compaction 的影响

第三部分测试的主要内容为：连续插入 4000 次数据（值为"test!"），记录每一次 PUT 操作的时延，并绘制出随时间变化的曲线图，观察并分析。

测试结果如下图，因为数据的方差过大，取了窗口大小为 100 的移动平均。可以看到测试结果和预期基本一致，在数据量小于 500 时，平均时延非常小，因为没有触发 compaction 机制；当数据量大于 500 后，每 400 次插入就会有一次开销巨大的 put 出现，这是因为大概在 400 次左右的插入后，0 层就会被重新填满而触发 compaction。同时随着数据量变大，compaction 的开销也会因为需要合并的层数逐渐变大。

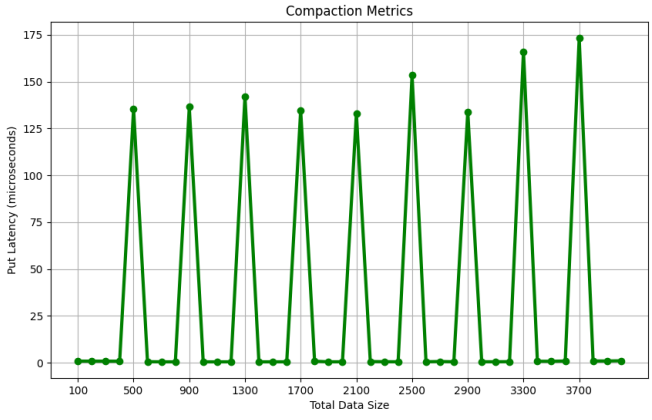

图 5: LSMKV 系统 PUT 操作时延随累积数据量变化曲线

3.2.5 Bloom Filter 大小配置的影响

第四部分测试的主要内容为：在不同的 BF 大小下（从 1KB 到 15KB，步长为 1KB）进行 10000 次 PUT 和 GET 操作，计算平均时延并绘图分析。

测试结果如下图。可以看到测试结果和理论预期不完全一致，PUT 操作确实随 BF 的大小呈正相关趋势，但是 GET 也同样呈一定的正相关。推测原因和上面提到的缓存策略有关，也有可能是因为本项目中采取了使用 minKey 和 maxKey 来先判定 sst 文件是否存在 key 的策略，而导致 BF 的判定没有那么重要，因此 BF 的大小越大，sst 文件的数目就越多，反而导致了 GET 操作平均时延的上升。

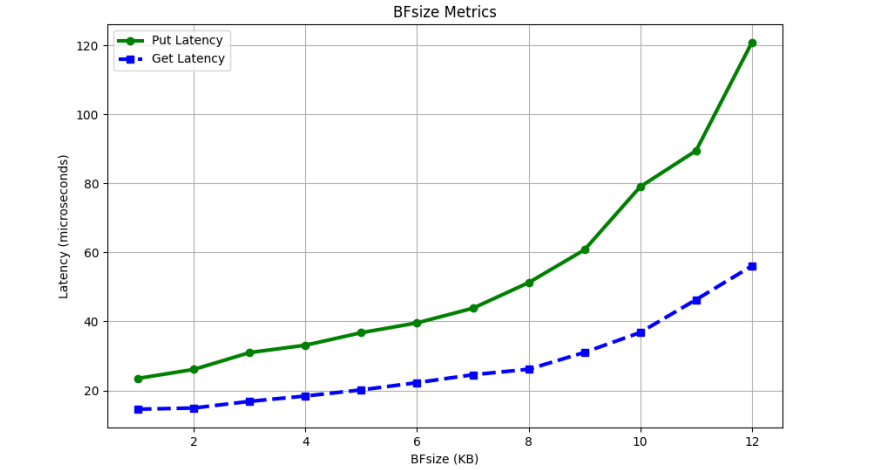

图 6: 不同 Bloomfilter 大小下 LSMKV 系统 GET 和 PUT 的性能

4 结论

本项目使用 C++20 标准实现了基于 LSM 树和键值分离技术的键值存储系统 LSMKV 结构，并进行了相应的正确性和性能测试分析，所有测试的结果都与理论预期较为符合，个别不符的结果也都可以通过分析得出合理的原因。

测试结果有几点或许对于优化 LSMKV 结构有用，比如在键值分离并采用缓存策略的情况下，**Bloomfilter 实际上并没有带来性能上的提升**，或许可以废弃使用。

总体而言，项目实现的 LSMKV 系统性能较优，且具备继续优化的潜力，整个实验较为成功。

5 致谢

本项目的代码实现没有参考任何开源项目、论坛、博客等，其中的跳表和布隆过滤器的实现均是本人之前在课程作业中自行编写的。

本人对于 LSM 树结构的了解参考了此文章：**深入浅出分析 LSM 树（日志结构合并树）**

6 其他和建议

本项目于 5.29 日开始编写，至今日 6.2 日完成，共耗时 5 天，此项目涉及的 LSM 树是当下数据库技术比较前沿的算法，做完整个项目还是收益颇丰的。对于这种不大不小的项目，从哪里入手确实是个很难的问题。一般而言我会先确定整个项目的代码结构之后再开始着手进行实现，而像课程建议的那样从 3 月份开始分阶段实现是我所不提倡的，拉长战线只会使自己遗忘上次实现的思路。当然老师和助教很善良，不要求大家强制按阶段实现，也就变相允许了像我这样的人在几天之内赶完项目，我也是赞成的。

对于本项目没有太多建议，大部分的 Lab 都是比较优质的。而对于此门课程吐槽点比较多，其中最忍俊不禁的就是学分课时安排，作为 3 学分的课代码量却比 ICS 还多上一倍，并且还包括大量的实验报告编写，实在是让我觉得这门课就是软工学生的专属“大物实验”。希望以后的教学计划要么多个这门课程分配课时，要么就减少一些无谓的作业，这学期的工作量已经相比上一届要好很多了，但是仍需改进，学生们并不是希望一味减少工作量，只是期望一个**更合理的课时安排**。

参考文献

[1] Patrick O’ Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’ Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 1996.

[2] 谈谈 1974. LSM 树 (Log-Structured Merge-Tree) 原理.