LSM-KV 项目报告

冯逸飞 520030910021

2022 年 5 月 30 日

1 背景介绍

LSM Tree 是一种可以高性能执行大量写操作的数据结构。它在 1996 年由论文《The Log-Structured Merge-Tree (LSM-Tree) 》[1] 首次提出，在年 Google Bigtable 论文之后得到了广泛应用。LSM-Tree 相比于传统的 B+ 树存储引擎，牺牲了部分读性能以具有更高的写性能。在本次 project 中，基于 LSM Tree，我实现了一个简化的键值存储系统 LSM-KV，实现了面向磁盘的键值对插入、查找、删除、范围查找等基本功能。

2 数据结构和算法概括

主要使用到的数据结构与算法

2.1 数据结构

1. SkipList

   负责项目中 MEMTable 的实现，处理内存的读写，保障了数据的有序性。

2. Bloom Filter

   用来初步判断数据是否存在于索引中，减少无效的内存读。

3. SSMsg

   用于存放 SStable 的索引信息和 BloomFilter，减少无效的内存与磁盘读写操作。

2.2 算法

1. Merge Sort

   用于在多个 SStable 合并时高效排列并生成有序的键值对。

2. Heap Sort

   用于 scan 操作时快速从多个 SStable 的索引中选出在需要范围内的键。

3. Compaction

   本项目最复杂的算法，用于在 SStable 数目超过本层上限时对本层和下层 SST 文件进行整合并生成新的 SST 文件。

3 测试

3.1 性能测试

3.1.1 预期结果

1. 对 PUT、GET、DELETE 的时延预测

   PUT：当多次独立地执行时，由于是直接写进 MEMTable，推测不同大小 Value 的写入时间应当接近；

   GET：由于大多数情况需要读取磁盘因此平均时延应当远高于 PUT，同时读磁盘的时间与 Value 的大小正相关，因此时延应当也会和 Value 的大小正相关；

   DELETE：需要先读缓存再执行 PUT 操作，且绝大多数情况无需读磁盘因此时延应当大于 PUT 但小于 GET。

2. 索引与 Bloom Filter

   无索引无 BloomFilter，每次 GET 都需要遍历所有 SST 文件的索引直到找到待查 Key 后读出 Value，进行了大量无效的磁盘读，性能最低；

   有索引无 BloomFilter 减少了无效的磁盘读，但是对每个索引都需要用二分法查找 Key，性能优于前者；

   有索引和 BloomFilter 一定程度克服了上述问题，性能最好。

3. Compaction 的影响

   进行连续的 PUT 时,早期由于不进行磁盘读写或只进行单次 MEMTable 到 SSTable 的转写，因此吞吐量必然大，但当 Compaction 触发瞬时吞吐量会下降到 0，且随着 SSTable 文件数的增大，Compaction 的时间会越来越长。

3.1.2 常规分析

1. PUT 的平均延迟

   本测试中，前后两次插入不连续，仅统计单次 PUT 操作的时延, 可以得到下图。

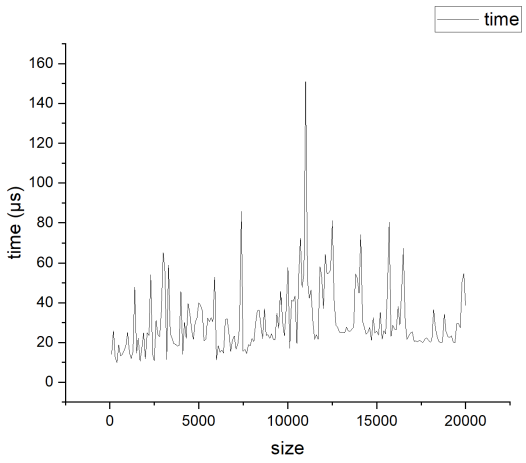

图 1: 插入时延

   可见 PUT 时，Value 的大小虽对时间有影响，但并不明显，平均 PUT 时延为 30.52 微秒。

2. GET 的时延

   通过对不同大小的 Value 进行测试得到的 GET 时延总体来看与数据大小成正相关，但存在较大抖动。平均时延为 587.45 微秒。

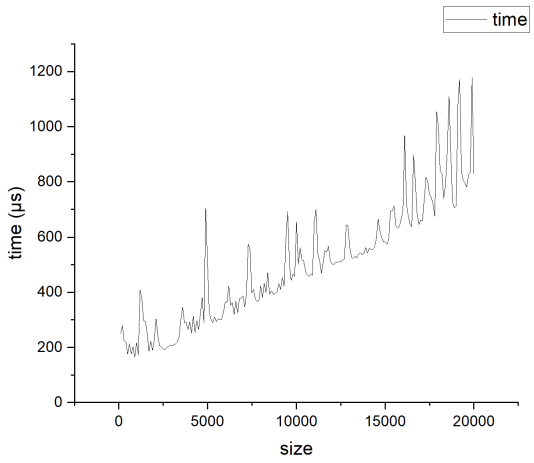

图 2: 查询时延

3. DELETE 的时延。

   通过对不同大小的 Value 进行测试得到的 DELETE 时延总体来看与数据大小成正相关，但存在较大抖动。平均时延为 520.42 微秒。

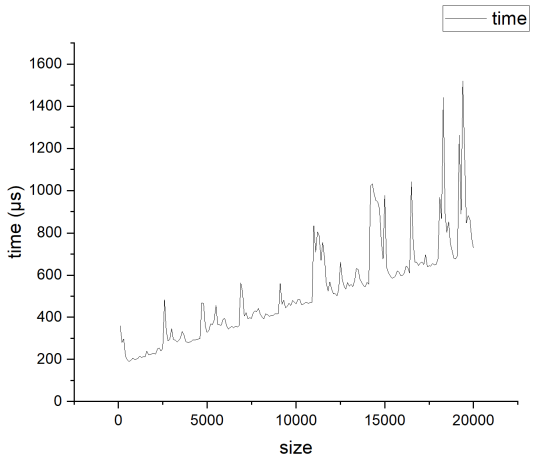

图 3: 删除时延

4. PUT 的吞吐量

   固定 Vlaue 为长度 50 的字符串，Key 从 0 开始递增，连续插入，得

到在 PUT 吞吐量为 100977。

5. GET 的吞吐量

   在已插入 500,000 个长度为 50 的字符串后，Key 从 0 开始递增，连续查找，得到在 GET 的吞吐量为 13142。

6. DELETE 的吞吐量

   在已插入 500,000 个长度为 50 的字符串后，Key 从 0 开始递增，连续删除，得到在 DELETE 的吞吐量为 24084。

3.1.3 索引缓存与 Bloom Filter 的效果测试

在连续 PUT Value 大小 1 20000 的键值对后进行以下测试：

1. 内存中未缓存 SSTable 的任何信息

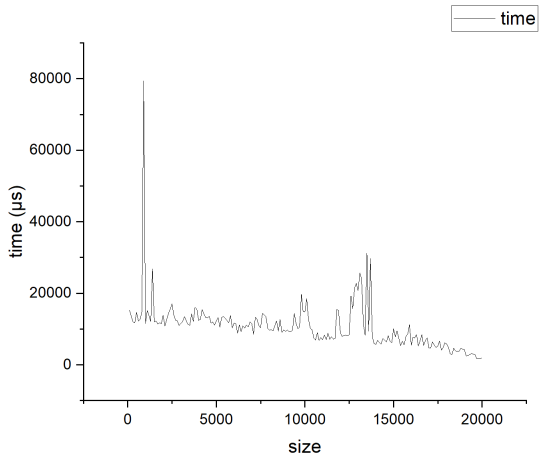

图 4: 无缓存

   GET 平均时延为 10659.3 微秒

2. 内存中只缓存了 SSTable 的索引信息

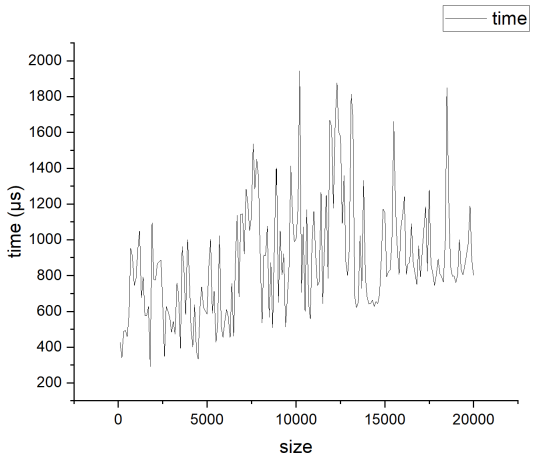

图 5: 缓存索引信息

GET 平均时延为 879.84 微秒

3. 内存中缓存了 SSTable 的 Bloom Filter 和索引

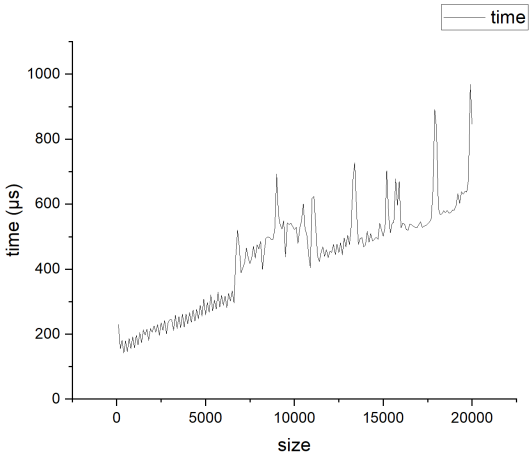

图 6: 缓存索引和 bloomFilte

GET 平均时延为 436.795 微秒。

3.1.4 Compaction 的影响

在进行连续 PUT 请求时，当一层的 SSTable 数量超过上限时可能会进行连续的 Compaction，导致某两次 PUT 之间有极大的时间差，这使得测得的吞吐量十分离散，往往在一秒内接受大量 PUT 请求，但更多时间是在等待 Compaction 执行完毕。实验中两次 Compaction 间的时间间隔记录如下。

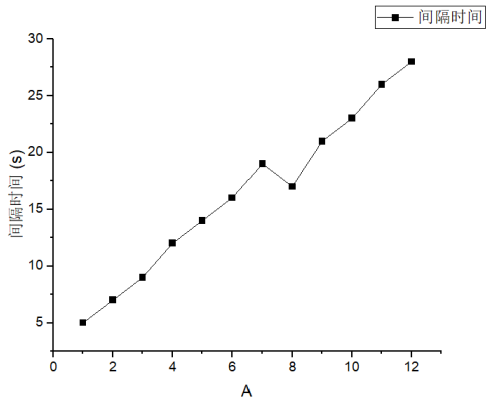

图 7: 相邻两次 Compaction 的时间间隔

由图 7 可知，compaction 操作所需的时间随 SSTable 的数量增加逐渐增长，为使这种变化更直观，将发生在一秒内的 PUT 数以相对平滑的方式分给了其余不进行 PUT 的时间，得到吞吐量变化如下。

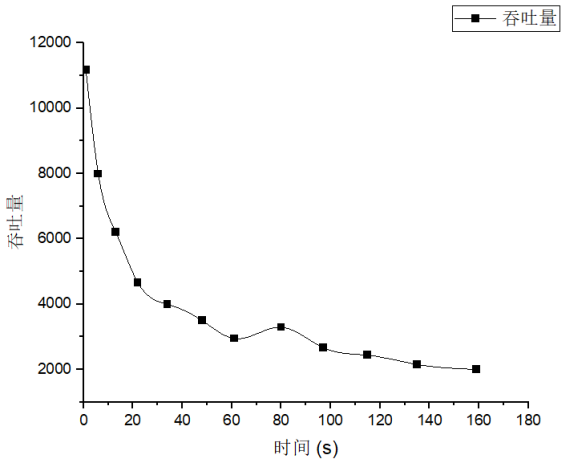

图 8: 吞吐量随时间的变化

可以明显看出随着时间的推移，PUT 的平均吞吐量逐渐下降。

3.1.5 对比实验

将 MEMTable 的实现由 SkipList 替换为 std::map 后，使用同样的方式测试并对比 PUT、GET、DEL 三个操作的性能，得到数据如下：

1. PUT 时延

   平均时延为 35.81 微秒，跳表: 30.52 微秒

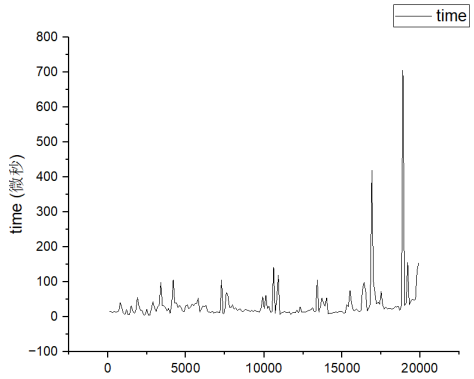

图 9: map 下的 PUT 时延

2. GET 时延

平均时延为 339.19 微秒，跳表: 587.45 微秒

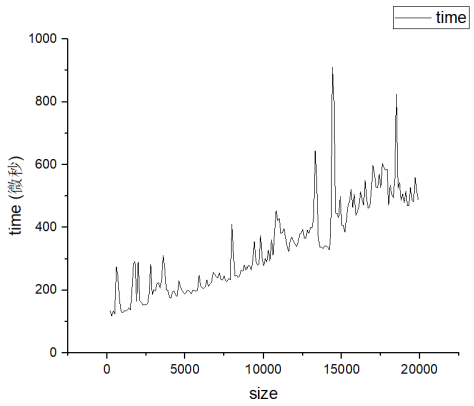

图 10: smap 下的 GET 时延

3. DELETE 时延

平均时延为 456.23 微秒，跳表: 520.42 微秒

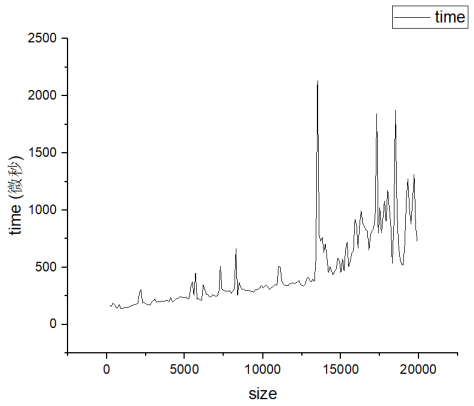

图 11: map 下的 DELETE 时延

可以看出，将 MEMTable 的实现改为 map 后，PUT 的时延略有上升，而 GET 和 DELETE 的时延有所下降。

由于 map 是无序存储，因此在生成 SSTable 前还需进行排序，但在测试时进行的是有序插入因此没有这部分时延，故 map 的 PUT 实际时延还会更高。

基于以上原因，当业务需要进行大量写操作时或写操作乱序程度高考虑使用跳表，若更偏向于查询或删除可以选用 std::map。

4 结论

4.1 实验结论

1. 对于 PUT、GET、DELETE 预期基本无误，意外的是在 DELETE 时 Value 的大小也对时延有影响，分析是由于在 DELETE 需要向 MEMTable 插入删除记号，而在测试时选用的键值对的 Vaule 大小与 Key 值正相关，因此更容易插在跳表的末端导致时延大，这说明了 Key 值的大小也会影响操作的时延。

2. 索引和 BloomFilter 的效果符合预期，都有效地减少了无效的磁盘/内存访问，大大提高了 GET 的效率，这说明了缓存在多级存储中的重要作用。

3. Compaction 对 PUT 操作的吞吐量有巨大影响，在 MEMTable 未达到上限时，PUT 的效率很高，但一旦触发了 Compaction 就会进入漫长的磁盘读写，且随着 SSTable 数量的增多该时间会越来越长。

4. 在内存中选择不同的数据结构来实现 MEMTable 有着不同的效果，由于 LSM 是一个强调顺序写的文件系统，因此可能顺序存储的数据结构会更加适合。

4.2 实验感想

本项目是我接触 coding 以来写过的最复杂，涉及知识面最广的代码，很好地锻炼了我数据结构与算法、文件读写、多文件管理等方面的能力。

另外，虽只是实现了基础的 LSM-KV, 但也让我初步了解到企业级项目开发的复杂性，大量测试也让我明白了想要写出一个正确性和健壮性良好的程序是十分不容易的。

在开发过程中遇到了很多 bug，大多数是由于对项目实现的细节不清楚或理解不深刻但匆匆动手导致的，例如前期将 kvstore 与 SStable 的功能大量耦合，导致后续进行了不少重构工作，或是出现缓存与磁盘在读写实现上的低级失误导致大量测试寻找 bug，这些都是 coding 之路上宝贵的经验。

5 致谢

感谢 github 用户 xtommy-1 的开源项目 LSM-Tree[2]，让我了解了如何有效地将内存信息写入磁盘文件；感谢知乎专栏文章《深入浅出分析 LSM 树》[3] 让我对 LSM 树的原理和优缺点有了更深入的认识；同时特别感谢卢天宇同学，和他的讨论让我提前规避了一些项目编码过程中的误区并学习了如何高效地利用 CLion 进行 c++ 程序的调试。

6 其他和建议

本项目对编程能力提升很大，就是漫长的 test 等待实在令人抓狂，但看着一个个 SST 文件在本地文件夹里创建又消失，莫名有种解压的感觉。

参考文献

[1] Patrick O' Neil; Edward Cheng; Dieter Gawlick; Elizabeth O' Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4)(351-385.), 1996.

[2] LSM-Tree. `https://github.com/xtommy-1/LSM-Tree`.

[3] 深入浅出分析 LSM 树. `https://zhuanlan.zhihu.com/p/415799237`.