LSM-KV 项目报告

沈哲赟 520021910933

2022 年 5 月 8 日

1 背景介绍

LSM Tree (Log-structured Merge Tree) 是一种可以高性能执行大量写操作的数据结构。它于 1996 年，在 Patrick O'Neil 等人的一篇论文中被提出 [3]。现在，这种数据结构已经广泛应用于数据存储中。Google 的 LevelDB 和 Facebook 的 RocksDB 都以 LSM Tree 为核心数据结构 [2]。

LSM 树并不像 B+ 树、红黑树是一颗单一的严格树状“数据结构”，而是一种包括内存、磁盘及相关读写逻辑的存储结构。read/write tradeoff 是所有存储结构都需要做出选择的一件事 [1]，LSM 树选择牺牲小部分读性能来换取高性能的写。

此 Project 需要基于 LSM 树实现可持久化的键值存储系统，提供 put/get/del/scan 四个操作。

2 数据结构和算法概括

图 1展示了 LSM 树的存储结构，主要分为 MemTable 与 SSTable(sorted string table)。MemTable 是内存中维护的高性能查找结构，可由跳表、红黑树等实现，需要实现相同的接口 (此 Project 选择跳表)。SSTable 是数据持久化写入磁盘后的存储格式, 具体如图 2, 所有键按从小到大排序, 一旦生成后不可变。设定 SSTable 的大小不能超过一个阈值（此 Project 设置为 2M），当 memTable 所存放的数据量生成 SSTable 大于此阈值时需要转化为 SSTable。

SSTable 在磁盘上采用分级存储的方式，层数越小存放的文件数量越少，同时数据越热。文件数量超出规定个数 (此 Project 规定 level-n 上限为

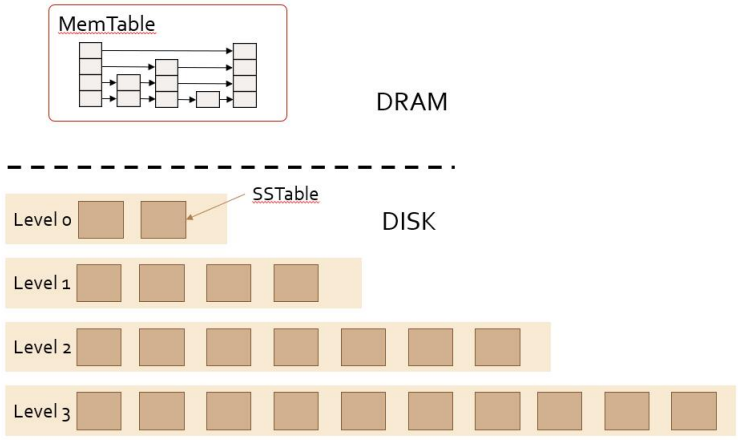

图 1: LSM-KV 存储结构

[2]

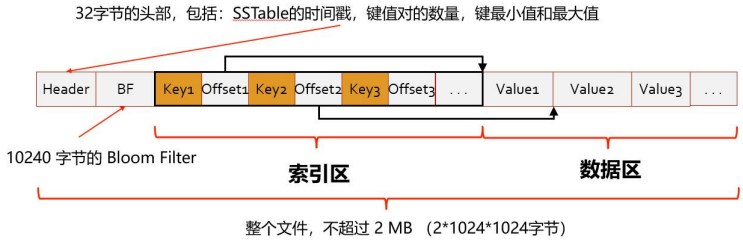

图 2: SSTable 存储结构

[2]

2^{n+1}) 的合并 (compactiom) 逻辑为：level-0 全部向下合并，其他 level 选择时间戳最小的若干个向下合并，并规定除 level-0 之外的层中必须满足每个 SSTable 保存的键范围均不相交。为了提高读取性能，在内存中保存一份相同结构的 SSTable 的缓存，即头部与索引区。

在实现的细节上，区间查询操作与某一层 SSTable 达到限制个数时触发的 compaction 操作都涉及到多路归并排序的算法；在索引区查找元素时使用二分查找的算法。

3 测试

3.1 性能测试

测试数据：key 范围 0 至 1000000 的随机数，value 固定长度 1000bytes

测试方法：生成给定长度个随机键值对，先连续插入所有键值对统计 put 延迟，再连续查找所有键值对及 $\frac{1}{5}$ 不存在的 key 统计 get 延迟，最后连续删除所有键值对及 $\frac{1}{5}$ 不存在的 key 统计 del 延迟。

3.1.1 预期结果

LSM-KV 对于 del 的实现方式为 put 一个标记，因此猜测 del 的延迟与 put 相近，下面只讨论 put 与 get。

1. 数据量较小时，LSM-KV 可以直接通过内存中的 MemTable 完成所有的操作，无需对于磁盘的读写，此时 get 操作只需找到改元素即可，而 put 操作需要在找到改元素后维护数据结构，因此 put 操作会慢于 get; 数据量增大至出现 SSTable 后，put 操作大部分情况下只需要写入 MemTable，不进行磁盘读写操作，而 get 操作涉及到的数据可能存在磁盘中，因此需要进行性能较低的磁盘读写, 性能会低于 put。

2. 缓存使每次 get 操作仅需确定元素在某一个 SSTable 中时才会打开文件，将每次 get 操作需要读取的文件减少至 1 次；布隆过滤器提供了快速判断元素是否在集合中的方法，避免了不必要的查找导致的内存访问。二者都提高了 get 的性能。

3. compaction 操作涉及到大量的文件读写，会极大提高单次 put 操作的延迟，从而降低了 put 的平均延迟。而在大量的 put 操作没有进行 compaction 时，单次 compaction 的影响会被慢慢削弱。

4. 其他 memTable 的实现方式由于复杂度都是 O(logn)，在大规模的情况下应该与本 Project 采用的跳表相差不大。

3.1.2 常规分析

1. 图 3展示了三种操作平均延迟与数据量的关系。首先从结果可以看出 del 与 get 的结果相近而并非 put，原因是要求实现的 del 需要返回是否成功删除，即需要先进行 get 以判断该元素是否存在，因此 del 延

迟一定比 get 略大，而在元素存在时 del 需要 put 一个特殊标记却不如 put 的延迟之慢，原因是使用的标记为“~DELETED~”只有 9 个 bytes，比起常规 value 的 1000 个 bytes 更不易触发 compaction

另外可以发现在数据量较大时 put 的延迟迅速上升，超过了 get/del，这似乎违背了 LSM-KV 的设计理念。原因是数据量继续增大后，虽然一次 put 操作只有很小的概率会引发 compaction，但是一次 compaction 操作由于 level 增加可能会变得极其耗时，平均延迟也增加。相反，由于 get 操作使用缓存判断元素位置，至多只需要一次磁盘读写的操作，延迟较稳定。

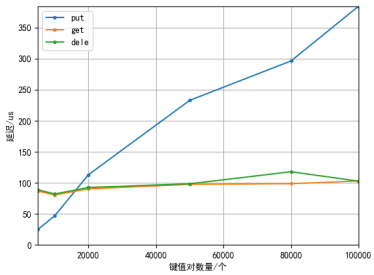

图 3: 三种操作平均延迟

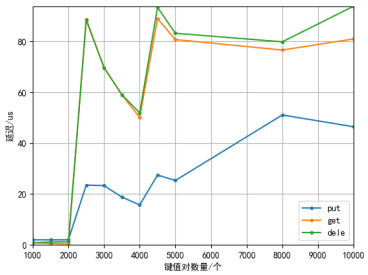

图 4: 数据量较小时三种操作的平均延迟

图 4展示了数据量较小时三种操作平均延迟与数据量的关系。仅在 memTable 中操作的部分只在键值对个数小于 2000 时发生。几个峰值表示了触发 compaction 的情况，每触发一次 compaction 会带来大量的延迟，而之后又会被慢慢分摊。get 需要读磁盘而 put 大概率不需要，因此 get 延迟高于 put。当 compaction 次数变多时，put 操作的延迟开始显著增加。

2. 图 5展示了三种操作吞吐量与数据量的关系。由于在数据量较小时吞吐量将达到十万量级以上，不方便在图上展示，因此只展示键值对数量大于 10000 的情况。可以看到 get/del 的吞吐随数据量增大而几乎不变，稳定在 10000tps，put 的吞吐量有一个迅速减小到逐渐平稳的过程，从优于 get/del 的 40000tps 降低至 5000tps 以下。

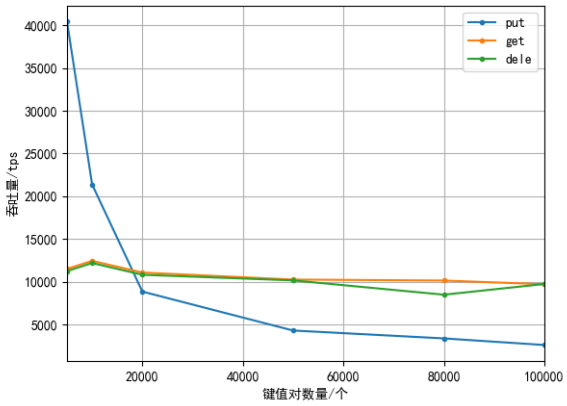

图 5: 三种操作平均吞吐量

3.1.3 索引缓存与 Bloom Filter 的效果测试

对比以下三种情况 Get 操作的延迟，结果如图 6

1. 内存中没有缓存 SSTable 的任何信息，从磁盘中访问 SSTable 的索引，在找到 offset 之后读取数据
2. 内存中只缓存了 SSTable 的索引信息，通过二分查找从 SSTable 的索引中找到 offset，并在磁盘中读取对应的值
3. 内存中缓存 SSTable 的 Bloom Filter 和索引，先通过 Bloom Filter 判断一个键值是否可能在一个 SSTable 中，如果存在再利用二分查找，否则直接查看下一个 SSTable 的索引 (此 Project 实现方式)

如图可得出以下三点结论：

1. 无缓存情况延迟巨大，且会随数据量增大而线性增加；有缓存情况的延迟在数据量增大后趋于稳定，符合预期。
2. 有无布隆过滤器对结果影响不大，不符合预期。笔者认为这与实验设置极度相关。在本实验参数下每个 SSTable 大约存 2000 个键值对，使用二分查找仅需 11 次内存访问即可；同时进行 get 时大部分的 key 是存在于存储系统中的，使用布隆过滤器反而增加了哈希函数的计算时间。

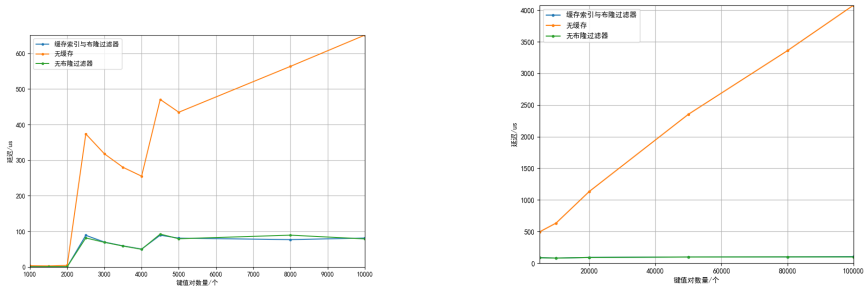

图 6: 三种情况 Get 操作平均延迟

3. 三种 get 的方式导致的不同延迟关系与数据量大小无关。均是情况 1 大于情况 2 等于情况 3 的关系。

3.1.4 Compaction 的影响

将每个 value 长度增加至 5000bytes，增加取样点，在数据量较小时更容易观察到 compaction 操作对于 put 延迟的影响，如图 7。

可见每次 compation 都会导致 put 的平均延迟迅速上升，随后下降，符合预期。原因已在 3.1.2 中进行阐释。

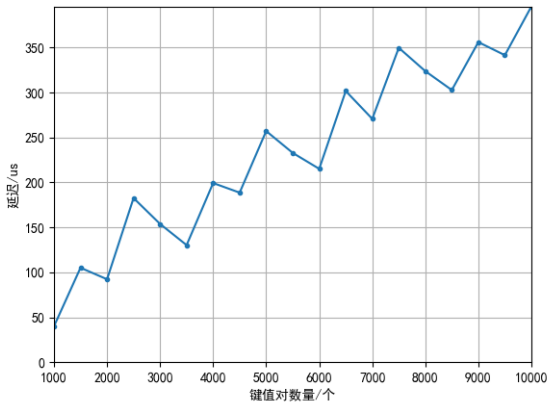

图 7: compaction 对 put 延迟的影响

3.1.5 对比实验

使用 std::map 实现 MemTable 的结果如图 8。可以看到图上只能看见 3 根完整的曲线，表明 skiplist 的所有曲线被 map 完全覆盖，二者的延迟差距可忽略不计。可见使用效率相同的两种数据结构实现 MemTable 对于 LSM-KV 整体的性能影响并不大。

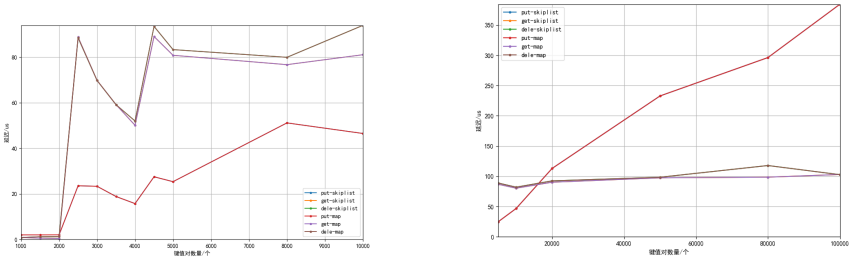

图 8: 不同 MemTable 下三种操作平均延迟

4 结论

LSM-KV 键值存储系统通过精妙的设计尽量减小对于三种数据操作的延迟。在中等数据量范围内 put 延迟会低于 get，在大数据量时 put 延迟增加，get 延迟稳定；缓存极大提升 get 的性能；compaction 会周期减弱 put 的性能；MemTable 的选择对所有操作性能影响不大。

此 Project 从零实现了一个前沿键值存储系统，对代码能力、存储结构理解都有较大的提升。实验结果大部分符合预期，put/get 操作延迟的变化关系也表明了即使某种键值存储系统的设计已经做出了对于某些操作的偏好，但对于不同的 workload 的表现也是不同的。

5 致谢

感谢同专业室友童楚炎在项目过程中的探讨交流，给了笔者很多启发。

感谢女友李荣洁在笔者无数次深夜 debug 时的鼓励。

6 其他和建议

写一个功能修 5 个 bug，运气不好一个 bug 得看 2 小时。在这个项目之后笔者对于 bug 已经看淡了，只要不出现以下三种情况都好说：

1. 每次测试错误的地方不一样
2. run 会出现错误 debug 错误消失
3. debug 报 segmentation fault 的时候跳到一个不是自己的代码的地方 (例如 ___atomic_add)

这些情况实在是太难 debug 了，不知道这些是不是和 IDE 有关，笔者使用 Clion。

犯过的愚蠢错误包括但不限于：复制一行大部分相同的代码的时候忘记把不同的地方改了，忘记 close 文件，变量没有初始化，数组越界，变量在处理的时候写错名字了，小于等于没加等于，没有想清楚逻辑导致合并的算法是错的……

debug 方法如下：

1. 如果出现每次运行的结果不同，或者指针错误，或者 run 和 debug 不同，考虑是内存或者文件的问题，因为这部分的问题 C++ 并不会发现，A 处的错误可能执行到了 B 处才会触发，除了肉眼看代码外并没有想到其他更好的方式。
2. 如果只是通不过测试，可以修改测试代码定位到错误的元素，然后在代码中追加对此元素的跟踪，例如 put/del/compaction 时都 cout。然后在错误的地方加 if 判断后打断点，进入 debug。在 debug 模式断点停下后，可以打上新的断点，Clion 按 F9 可以直接执行到下一个断点的位置 (但是此时新增代码是没用的)。

教训：

1. 写代码之前一定要先把所有的算法和逻辑都想想好，多花几分钟仔细一点把一行写对了可能节省几个小时的 debug 时间。
2. 涉及到文件、内存的时候要小心，这部分就算错了也很难被发现。
3. 测试很重要，很多 corner case 可能跑了测试才能意识到。如果测试不能覆盖某些代码那这段代码如果是错误的就不能被发现。

参考文献

[1] Brian F. Cooper. Benchmarking cloud serving systems with ycsb. 2010.

[2] Mingkai Dong. Project lsm-kv: 基于 lsm 树的键值存储系统. 2022.

[3] Patrick O'Neil. The log-structured merge-tree (lsm-tree). 1996.