LSM Tree 实验报告

沈玮杭 519021910766

5 月 26 日 2021 年

1 背景介绍

LSM Tree (log-structured merge-tree) 是一种具有较高写性能的数据存储系统。LSM Tree 克服了 B+ 树在写操作时会产生大量随机 IO 的性能缺陷。相比 B+ 树，它牺牲了一部分读性能，而大大增加了写性能。对于以硬盘为主要存储媒介的大容量数据存储系统，LSM Tree 是一个非常合适的存储策略。LSM Tree 已在 Apache HBase 数据库使用，正在工业界推广。

LST Tree 主要由三个部分组成：MemTable、Immutable MemTable 和 SSTable (Sorted String Table)。MemTable 是一个跳表，储存在内存中，用于保存最近更新的数据。当 MemTable 达到设定阈值时，会转化为 Immutable MemTable，此时写操作由新的 MemTable 处理，该 Immutable MemTable 正在被转化成 SSTable 储存到磁盘上，这样可以使得在保存过程中的操作不会被阻塞。在本次 lab 中，由于没有这样的需求，因此 Immutable MemTable 不是必要的。SSTable 是存储在磁盘上的键值对集合。为了加快 SSTable 的读取，在本次 lab 建立了索引以及 BloomFilter（布隆过滤器）来加速查找。SSTable 和 LSM Tree 的结构如图 1和图 2所示。

2 挑战

内存中的 SkipList 和读写 SSTable 都很好实现，最具挑战性的是 SSTable 的合并，需要慎重地分析每一种情况，仔细地检查每一行代码才能保证正确。

在选择 SSTable 进行合并时需要特别谨慎，我一开始并没有考虑上一层的 SSTable 时间戳相同的情况，只是选择了固定数量的时间戳最大的

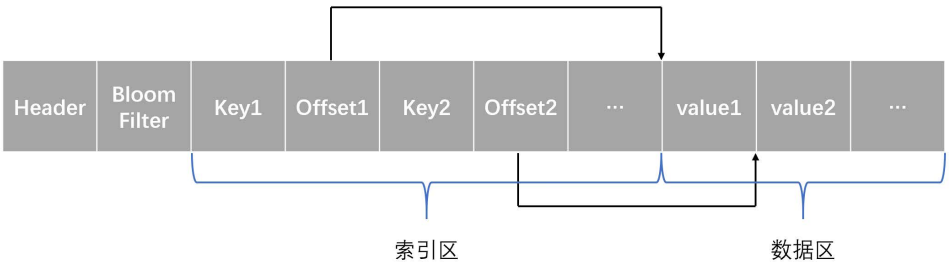

图 1: SSTable 结构

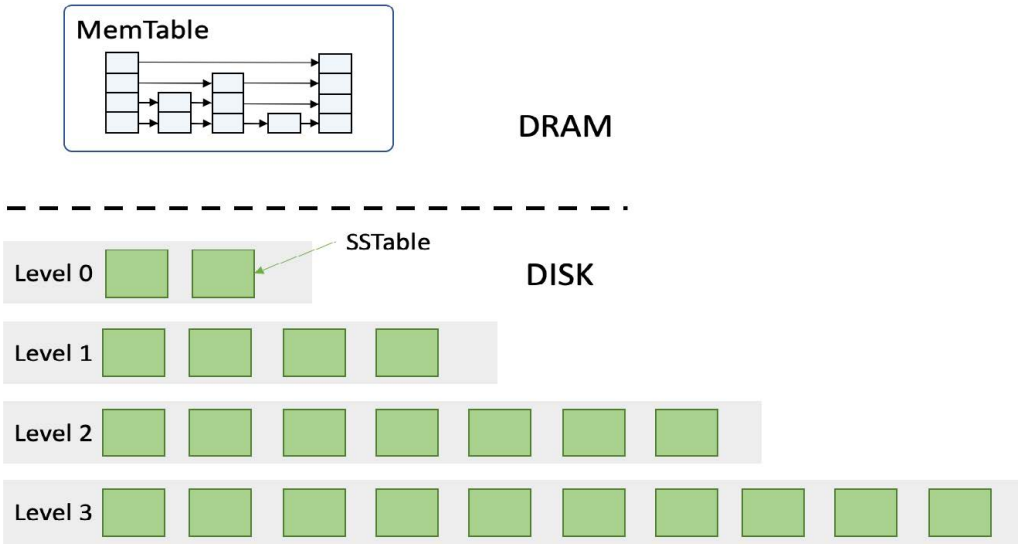

图 2: LSM Tree 结构

SSTable。结果导致在特殊情况下会出现 bug：如上一层有 10 个时间戳为 5 的 SSTable，从中选出了 5 个与下层合并，若下层与其合并的 SSTable 中有若干个与上层剩余的 5 个时间戳为 5 的 SSTable 有交集时，由于此次合并生成的时间戳是最大的，在下一次合并时，下一层中旧的值可能会覆盖上一层中新的值，会导致错误。这个问题在文档中没有细说，也很难发现，耗费了我很多时间。

3 测试

该部分对 LSM Tree 的性能进行了测试,主要分析了 PUT、GET 和 DEL 操作的吞吐量和时延，并修改了代码逻辑来对比分析内存中 BloomFilter 和索引缓存对 GET 性能的影响。

3.1 性能测试

3.1.1 预期结果

由于没有使用多线程加速，三种操作的吞吐量和时延应为倒数关系。由于 DEL 操作要返回 store 中是否存在，所以要先做一次 GET，再插入一个“~DELETED~”，因此 DEL 操作时延可能会比 GET 略高。而 PUT 操作可能会触发合并，而一次合并有较大的性能开销，因此在持续 PUT 时会有若干次时延相当高。又因为 DEL 插入的字符串很小，只有 9 个字节，很少会触发合并，因此字符串很长的 PUT 操作时延会显著高于 DEL。

BloomFliter 可以提前判断 SSTable 中是否有查询的 key，可以避免无用的查找，因此会提高很多 GET 性能。由于磁盘的速度比内存慢很多个数量级，在内存中缓存 index 可以显著地加快查找的速度。

3.1.2 常规分析

1. GET、PUT、DEL 操作的延迟：

   测试 GET 使用随机的 key 进行插入，插入的 Value 大小为 10000 字节。测试 PUT 和 DEL 使用随机的 key，并且该 key 在 store 中不存在的概率约为 0.2，以模拟真实的使用场景。每种操作的测试包含 100、1000、10000 个条目的三种规模的测试。由于篇幅原因，且代表性没有 10000 条目的强，100、1000 条目的时延图就不在此给出。

   在 100 条目规模的测试中，由于 MemTable 未达到阈值，故三种操作时延均很理想：GET 操作平均时延为 0.76μs，PUT 操作平均时延为 6.56μs，DEL 操作平均时延为 0.65μs。

   在 1000 条目规模的测试中，由于 MemTable 达到阈值，会产生磁盘读写，故时延变大：GET 操作平均时延为 25.74μs，PUT 操作平均时延为 34.08μs，DEL 操作平均时延为 17.76μs。

在 10000 条目规模的测试中，时延进一步变大：GET 操作平均时延为 33.34μs，PUT 操作平均时延为 173.19μs，DEL 操作平均时延为 23.00μs。

从图 3可以看出，10000 次 GET 操作中，大多数时延分布在 35.4μs 到 40.8μs 和 0.3μs 到 3.0μs 间，其中后者是直接在 MemTable 中直接访问的时延。

图 4中的尖峰表示此处触发了一次合并，可以发现合并操作非常耗时，因此也导致了 PUT 操作平均时延变长。

与预期结果不同，DEL 操作反而要略快于 GET 操作。从图图 5中可以看出，在 MemTable 命中的次数增加了，猜测可能是由于一次 DEL 操作至多插入 9 个字节，数据量很小，所以 MemTable 所能容纳的条目变多了。

图 3: 带缓存的 10000 次 GET 操作时延分布直方图

2. GET、PUT、DEL 操作的吞吐：

   经计算：

   在 100 条目规模的测试中，GET 操作吞吐量为 1315789IOPS，PUT 操作吞吐量为 152439IOPS，DEL 操作吞吐量为 1538461IOPS。

   在 1000 条目规模的测试中，GET 操作吞吐量为 38850IOPS，PUT 操作吞吐量为 29342IOPS，DEL 操作吞吐量为 56306IOPS。

   在 10000 条目规模的测试中，GET 操作吞吐量为 29994IOPS，PUT 操作吞吐量为 5774IOPS，DEL 操作吞吐量为 43478IOPS。

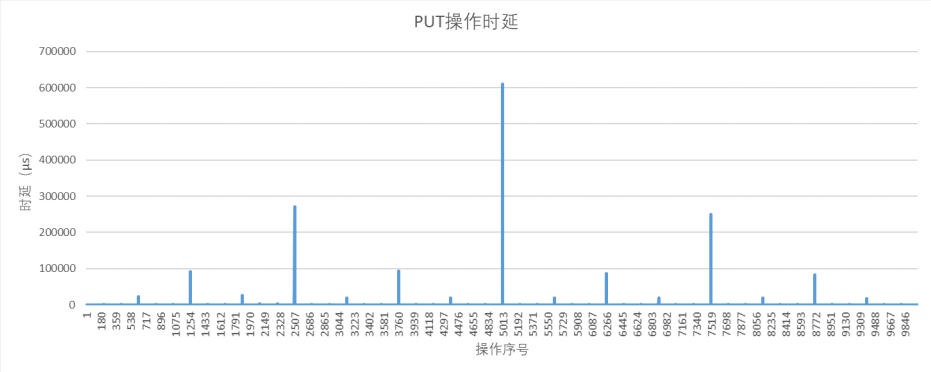

图 4: 带缓存的 10000 次 PUT 操作时延

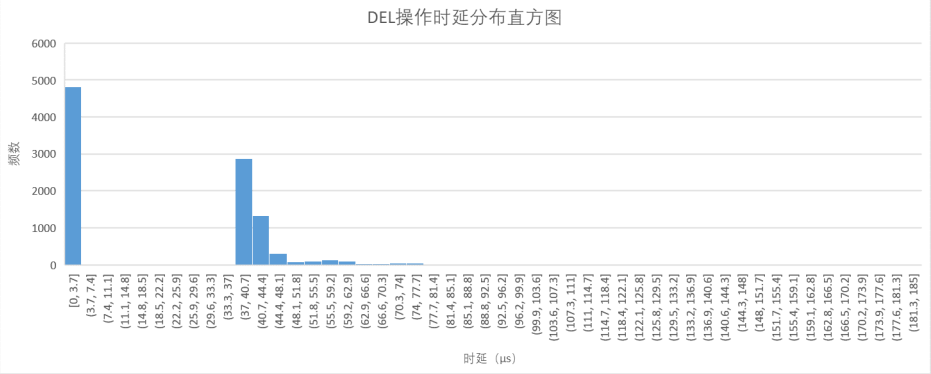

图 5: 带缓存的 10000 次 DEL 操作时延分布直方图

3.1.3 索引缓存与 Bloom Filter 的效果测试

本次 lab 对比了以下三种情况 GET 操作的平均时延：

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：

   在 100 条目规模的测试中，GET 操作平均时延为 0.60μs。在 1000 条目规模的测试中 GET 操作平均时延为 444.53μs。在 10000 条目规模的测试中，GET 操作平均时延为 1290.85μs。

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：

在 100 条目规模的测试中，GET 操作平均时延为 1.38μs。在 1000 条目规模的测试中 GET 操作平均时延为 32.26μs。在 10000 条目规模的测试中，GET 操作平均时延为 40.55μs。

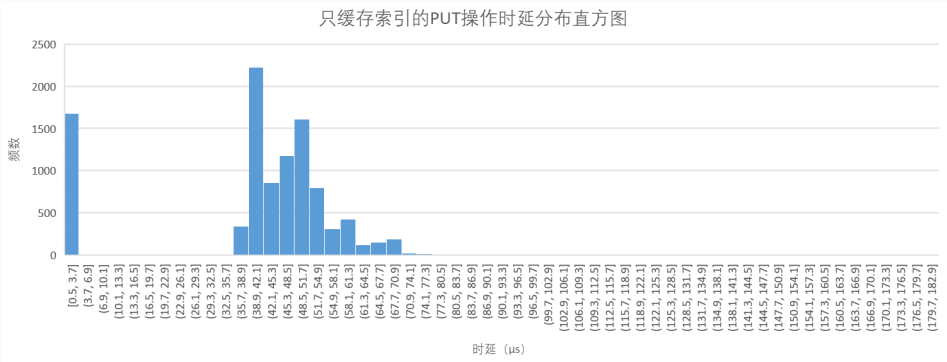

图 6: 只缓存索引的 10000 次 PUT 操作时延分布直方图

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：

   在 100 条目规模的测试中，GET 操作平均时延为 0.76μs。在 1000 条目规模的测试中 GET 操作平均时延为 25.74μs。在 10000 条目规模的测试中，GET 操作平均时延为 33.34μs。

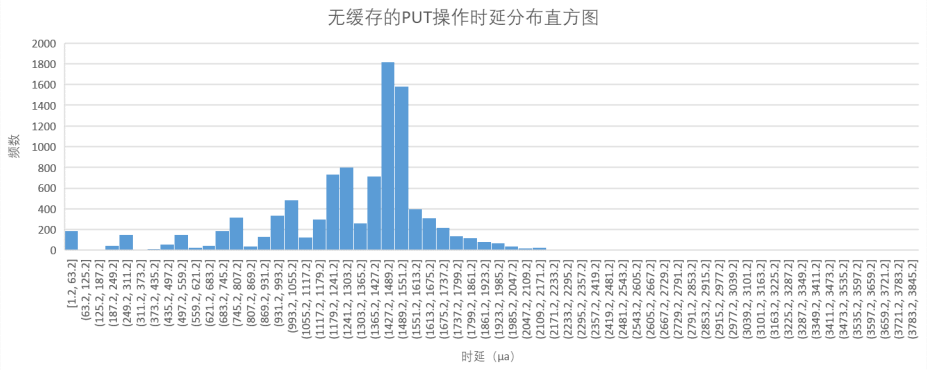

图 7: 无缓存的 10000 次 PUT 操作时延分布直方图

从结果可以看出 BloomFilter 对性能提升有一定作用，而缓存索引由

于可以大大减少对磁盘的随机读写，因此可以显著地提高 LSM Tree 的性能。由于篇幅原因，且代表性没有 10000 条目的强，100、1000 条目的时延图就不在此给出。

3.1.4 Compaction 的影响

本次 lab 统计了不断插入数据的情况下，统计每秒钟处理的 PUT 请求个数（即吞吐量），并绘制其随时间的变化图，如图 8所示。

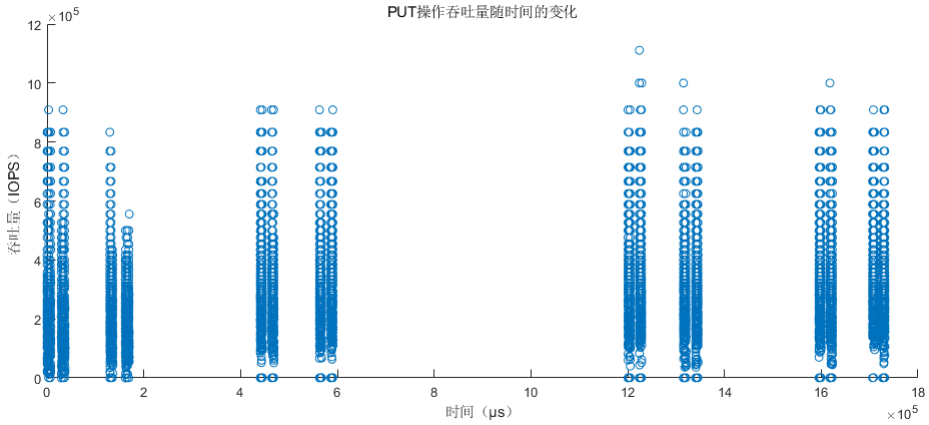

图 8: 10000 次 PUT 操作吞吐量随时间的变化

插入的数据为 10000 个 Key 随机 Value 大小为 10000 的键值对。图中间的空白表示在这段时间内都在执行合并，没有数据插入，因此没有吞吐。图 4也很好地展示了合并操作的影响，每一次合并表现为一次较长的时延，且合并层数越深，合并时间越长。

4 结论

通过常规分析，可以发现 DEL 操作略快于 GET 操作，并显著快于 PUT 操作。通过对比有无缓存的情况，可以得出 BloomFilter 对 GET 操作性能提升较小，而索引缓存对 GET 性能的提升很大。对 Compaction 的分析表明，PUT 操作慢的主要原因是合并时间很长。一般情况下 PUT 操作很快，而若需要合并，则操作的时延会高几个数量级。

LSM Tree 是一个根据计算机各存储结构设计的键值对存储系统，既通

过 SSTable 和索引缓存的方式规避了磁盘随机读写慢的问题，同时也通过 MemTable 的方式充分发挥了内存高速读写的优势。LSM Tree 也很好地利用了局部性原则，将最频繁使用的数据放在最快的地方，同时获得了近与内存的速度和近于磁盘的大容量。LSM Tree 的设计思路很巧妙，也非常值得我们学习。同时，这也告诉我们在设计系统时要充分考虑和利用计算机体系结构的特性和使用场景。

在本次 lab 中，我的代码能力也有所提高，也明白了动手之前要先想清楚，然后再写代码。