LSM-KV 项目报告

罗世才 520021910605

2022 年 5 月 9 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。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 数据结构和算法概括

LSM-tree 是一种分层的、有序的、基于硬盘的数据结构，它的核心思路是首先将数据写入到内存中，不需要每次有数据更新就必须将数据写入到磁盘中，等到积累到一定阈值之后，再使用归并排序的方式将内存中的数据合并追加到磁盘。

所以该系统分为内存存储和硬盘存储两部分，两部分采用不同的存储方式。内存存储结构被称为 MemTable，其通过跳表或平衡二叉树等数据结构保存键值对。硬盘存储采用分层存储的方式进行存储，每一层中包括多个文件，每个文件被称为 SSTable（Sorted Strings Table），用于有序地存储多个键值对（Key-Value Pairs）, 每个 ssTable 由 Header,Bloom Filter, 索引区和数据区组成 [1]。SSTable 和 LSM Tree 的结构如图 1和图 2所示

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，修改了代码逻辑来对比分析内存中 BloomFilter

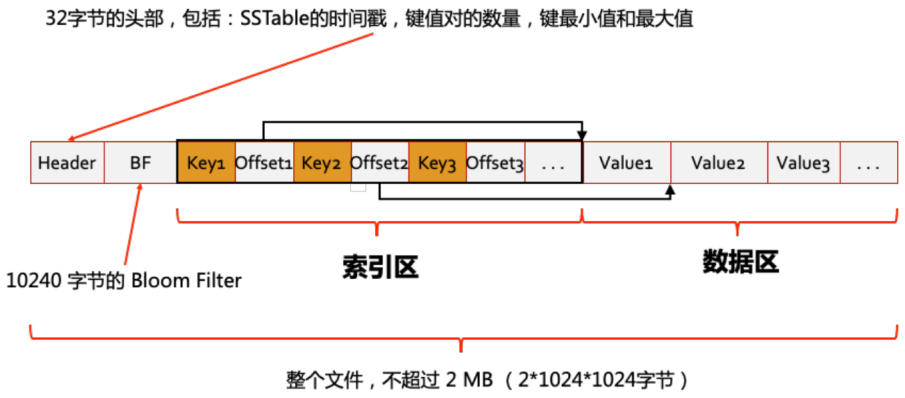

图 1: SSTable 结构

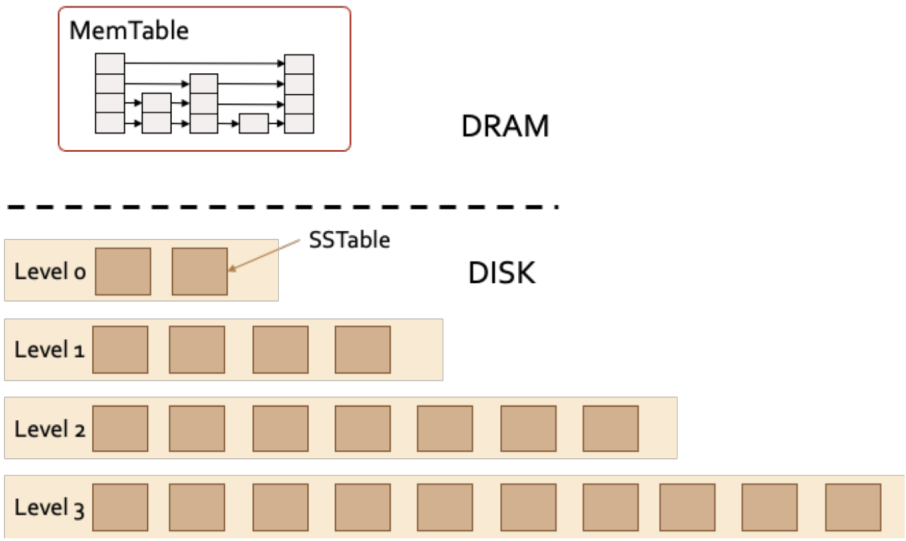

图 2: LSM Tree 结构

和索引缓存对 GET 性能的影响以及 Compaction 对性能的影响。最后，关于 memTable 的存储，选了 std::map 与项目中原选用的跳表做了对比试验，比较了两种实现对键值存储系统的性能影响。

3.1 性能测试

3.1.1 预期结果

就吞吐量和时延来说，三种操作的吞吐量和时延应为倒数关系。从算法实现的角度来具体的分析每个操作的时延的话，首先，考虑到 compaction 的问题，由于 Put 操作可能会触发写入内存和合并，会造成很大的性能开销，所以持续 PUT 时会有若干次时延特别高,Put 操作的时延也应该高于 Get 和 Del 操作。其次，每次 Del 操作都需要检查被删除的值在存储中是否存在，相当于先进行一次 Get 操作，进行判断后再视情况插入删除的标记，虽然 Del 插入的标签是 "~DELETED~"，所占空间很小，很少会触发写入磁盘的操作，但猜测 Del 的延迟应该略高于于 Get 操作。

就索引缓存和 BloomFileter 对性能的影响来说，很明显，这两种方式都可以提高性能，但是提高性能的原因不同。索引缓存是因为读写磁盘比读写内存慢很多个数量级，它可以减少读写磁盘的次数从而提高性能，而 BloomFilter 是因为可以减少无用的查找来提高性能，分析下来考虑到本数据结构如果没有缓存, 会大量增加读写磁盘的次数，因此可能索引缓存对性能的提升更大一点。

就 compaction 对性能的影响来说，由于合并会大量的读写磁盘而且一次合并可能会引发多次的合并，所以可以预见发生 compaction 后时延应该显著上升。

3.1.2 常规分析

1. Get、Put、Delete 操作的延迟

   在这次测试中，随机生成 key 值进行 Put,Get,Del, 在 Put 时，value 的大小为 10000 字节。在 Get 和 Del 时，为了模拟真实的使用场景，随即生成的 key 在存储中存在的概率为 0.5。对于每种操作分别执行 10000 次，进行测试，得出结果如下所示：

操作	Get	Put	Del
平均时延 (us)	45.0	141.4	33.9

   从上表可以看出，实验结果比较符合预期，Get 操作时延最高，Put 与 Del 操作相近，但令我感到意外的是 Del 操作的时延实际上是低于 Get 的，猜测可能是因为 Del 操作一次最多插入 9 个字节，数据量很

小，导致 memTable 中的条目增加。

2. Get、Put、Delete 操作的吞吐
   经过计算，Get 操作的吞吐量为：22222 次/秒
   Put 操作的吞吐量为：7072 次/秒
   Delete 操作的吞吐量为：29499 次/秒

3.1.3 索引缓存与 Bloom Filter 的效果测试

本次项目对比了下面三种情况 GET 操作的平均时延，输入数据是 10000 个长度为 10000 个字节的字符串，为了模拟实际使用的情况，Get 操作随机生成的 key 值有 0.5 的概率在存储中，0.5 的概率不在存储中。

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

   测得 Get 操作的平均时延为：1290.3 us

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

   测得 Get 操作的平均时延为：46.5 us

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

   测得 Get 操作的平均时延为：45.0 us

   从上面的数据可以看出，在内存中缓存 SSTable 的索引信息以及使用 BloomFilter 均可以降低 Get 操作的时延，但是缓存索引极大的减少了访问磁盘的速度，显著降低了时延，BloomFilter 减少了不必要的查找，虽然也降低了时延但是效果没有缓存索引显著。

3.1.4 Compaction 的影响

持续插入 10000 个字符串长度为 10000 字节的数据，记录每一个数据插入用时，绘制折线图如图 3所示。

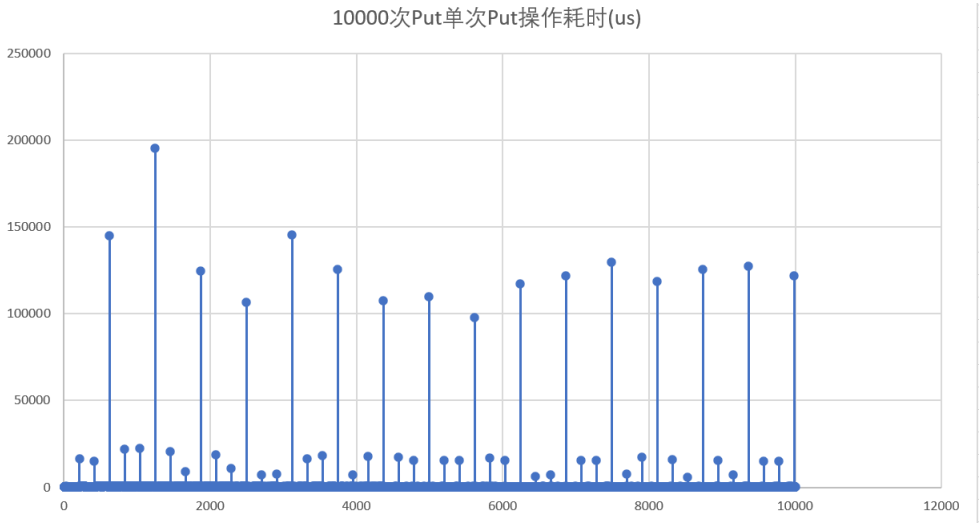

图 3: 持续插入过程中单次 Put 时延的变化

从图 3 中可以看出，10000 次插入中大部分 Put 操作所需时间极短，接近于 0，但是有规律的每隔一段距离后的某一个 Put，其所需时间会较显著的增加，在隔一段较长的距离后的某一个 Put，其所需时间会非常显著的增加。进一步分析数据，由于持续插入的是字符串长度为 10000 的数据，故大概每 200 次插入会生成一个新的 ssTable 写入内存，图中比较短的有规律增加便属于这一种情况。由于 level0 最多只能由两个 ssTable，所以每生成 3 个 ssTable，也即大概 600 次插入便会出发 compaction，造成时延异常的长，图中较长的有规律的增加便属于这种情况。理论可以解释实验结果，体现出了 compaction 的影响。

3.1.5 对比实验

在这次测试中，随机生成 key 值进行 Put,Get,Del, 在 Put 时，value 的大小为 10000 字节。在 Get 和 Del 时，为了模拟真实的使用场景，随即生成的 key 在存储中存在的概率为 0.5。对于每种操作分别执行 10000 次，进行测试，得出结果如下所示：

memTable 类型	Put	Get	Del
skipList	45.0	141.4	33.9
std::map	45.6	141.7	33.5

从上表可以看出，选用跳表还是选用 std::map 对三种操作的影响不大，分析后认为，由于数据的字节和数据量均很大，每次操作中的很多时间都消耗在了对 ssTable 的操作上，对于内存中的操作，跳表和内部用红黑树来实现的 std::map 其实都是 AVL 树的一种变体，增删改查性能差别不是特别大。

但值得一提的是，与自己设计跳表相比，使用 C++ 已有的 std::map 在 memTable 功能的实现上要简单很多。

4 结论

总的来说，LSM Tree 是一设计思路巧妙，实用性很强的数据结构。其既通过 memTable 的方式充分发挥了内存高速读写的优势，又通过 ssTable 和缓存解决了读写磁盘缓慢的问题。此外，LSM Tree 还很好的利用了局部性原则，将最近插入的结点放在内存中以便快速访问。在本次写 project 过程中，我的代码能力有所提高。以后在写项目时要注意变量命名以及可能需要多次使用的功能可用函数封装避免代码重复。

5 致谢

感谢 github,CSDN 等论坛、博客，让我学习了许多新的知识，比如二进制读写等，帮助我完成了此次项目。

6 其他和建议

建议学弟学妹们在写这个项目时一定要注意模块之间的区分，模块之内联系紧密，模块与模块之间仅通过接口连接，这样的代码易于修改和维护。在最后的对比实验中，要将 memTable 的实现方式改变，如果没有很好的组织代码，会造成很大的返工。

参考文献

[1] *Project: LSM KV 键值存储系统.*