LSM-KV 项目报告模板

钱韦克 521021910760

2023 年 5 月 4 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。这个数据结构能够通过高效利用空间来实现众多键值对的操作，并且能够反复重启，具有一定的稳定性，非常适合用来存储大批量的、有长期使用需求的数据。

2 测试

2.1 性能测试

2.1.1 预期结果

对于 2.1.2 常规分析部分：对于延迟，应该是 put<get<delete，因为 put 仅需要在 memtable 里面操作，而且 delete 操作需要额外调用一次 get 操作。对于吞吐，get<delete<put。因为我的内存中并没有缓存 memtable 的 bloomfilter 的数据，因此 get 操作需要单独占用打开文件，从而吞吐量低于其他操作。而 delete 操作同理, 因此略慢于 put 操作。

对于 2.1.3 索引缓存测试：平均时长应为 3>2>1。应为打开文件和读取需要一定时间，而扫描文件也需要时间。

对于 2.1.4Compation 的影响，应当在层数越来越多时，Compaction 操作所耗费的时间将会越来越长，导致吞吐量的下降。

对于 2.1.5Level 配置的影响，应当在前几层的个数较少、后面几层的个数较多时，能够尽可能延后层数的增加导致的 Compaction 操作花费的上升，从而一定程度上提升吞吐量。而若前面的层数过少，反而会使前面的 Compaciton 次数增加；对于后面几层的个数过多的话，会导致在合并时扫描整层文件的开销增加，最终都会导致吞吐量的反降。因此，应当合理安排各

层文件的个数，例如为 2 的幂次。而对于模式而言，Tiering 模式过多将会导致合并过程中选取的文件增加，因此应尽可能减少模式为 Tiering 的层数，从而提升吞吐量。

2.1.2 常规分析

1. 以下数据内容均在依次进行长度为 1 到 10240 的字符串的操作时测量所得。由于数据的规模包含了多次的合并（计算在了 put 里面），因此 put 的时间延迟应为最长。

   测试结果（单位为秒）：put=0.001086 get=0.000865 delete=0.000873

2. 以下数据测试内容在 lsm 中已有 1 到 10240 的键值对及其对应的 memtable 和 sstable 时进行，测试方式为一直生成线程来执行对应操作，到时间为 1 秒时停止。其中，put 操作的对象为随机值的键值对，get 和 delete 操作的参数为 1 到 20480 的随机数。

   测试结果（单位为次）：put=27071 get=7939 delete=22250

2.1.3 索引缓存与 Bloom Filter 的效果测试

测试情况为 kvs 当中已经缓存好了 1 到 10240 长度的键值对及其 memtable 和 sst，并且每次查找时从 1 到 10240 依次查找，最后计算得到结果后取得平均值。

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

   time=0.003707

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

   time=0.001921

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

   time=0.001882

2.1.4 Compaction 的影响

在以下的测试内容中，采取的策略为每次插入长度为 100000 的键值对，并且每过一定时间测量出并计算得到插入操作的吞吐量。同时，也记录了该数据结构中当前的最大层数。

图 1

在该图中，横坐标为每次测量的序次，橙线为最大的 Layer 层数，蓝线为吞吐量。可以看出，当层数上升时，吞吐量会下降，且开头下降得最为明显。

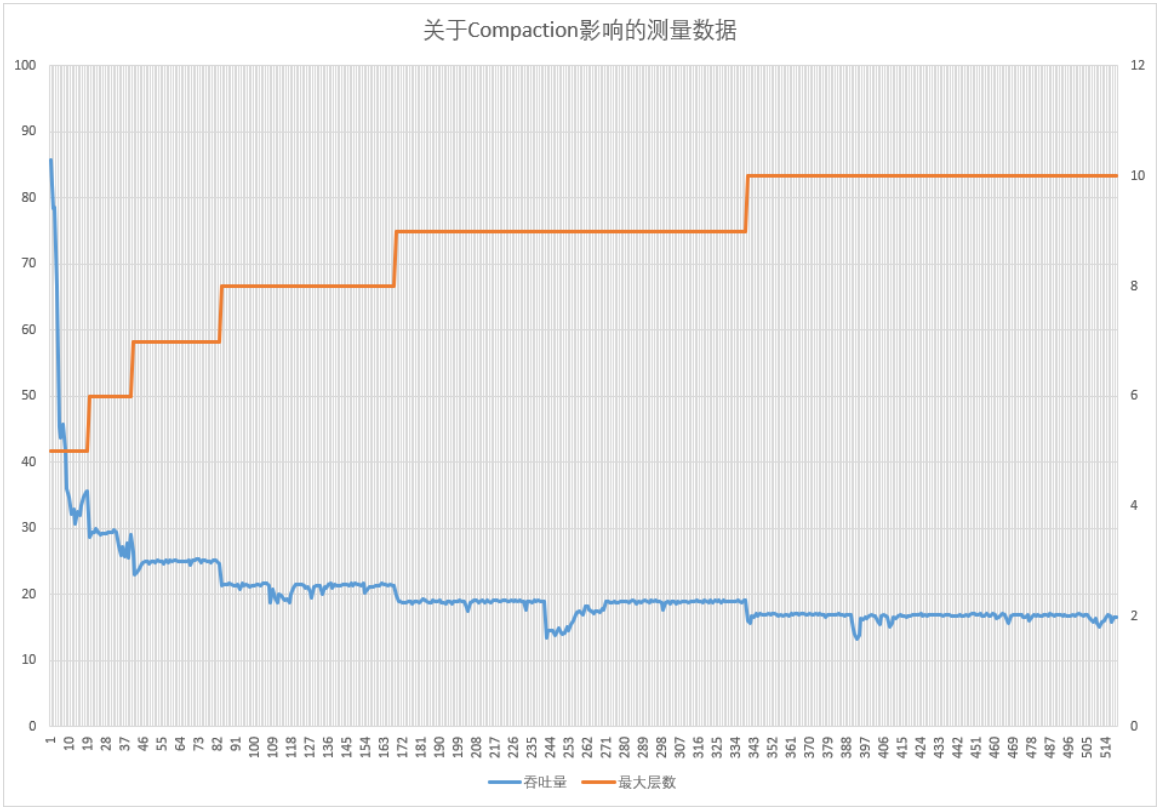

图 1: 2.1.4 Compaction 影响的测试

2.1.5 Level 配置的影响

测试方法为，通过改动配置文件后，清空过的 lsm 中执行数次数据范围为 0 1024*16 的键值对的插入、查找和删除，并且测量统计每次所有操作的耗时。另外，我的 lsm 新增层数的策略为：层数最大容量为上一层的两倍，该层的模式同上一层一样。下为测试结果：

Level 配置:

0 2 Tiering

1 4 Leveling

2 8 Leveling

3 16 Leveling

4 32 Leveling

运行时间:

time=57.756996s

Level 配置

0 2 Tiering

1 4 Tiering

2 8 Leveling

3 16 Leveling

4 32 Leveling

运行时间

time=57.306689s

Level 配置

0 2 Tiering

1 4 Tiering

2 8 Tiering

3 16 Tiering

4 32 Leveling

运行时间

time=61.122488s

Level 配置

0 2 Tiering

1 4 Tiering

2 8 Tiering

3 16 Tiering

4 32 Tiering

运行时间

time=73.462177

从中可以看出，当模式为 Tiering 的层数比较多时，操作的时间会比较慢，相对而言吞吐量就比较低。而当 Tiering 层数为两层时，相对一层 Tiering 的 LSM 的操作时间反而有一点点下降。原理上而言，应该一部分是因为 Tiering 的层在合并的时候需要选取该层中所有的文件，从而导致合并的开销增加不少。但当层数只有两层时，此处的开销增加不太明显。而且又由于 Level 0 和 Level 1 的合并时最频繁的合并，而当 Level 1 的模式也为 Tiering 时合并过程就不用选取 Level 1 的文件，从而导致这两层在合并时的开销略有缩小，总体上对性能造成的增益大于 Level 1 向下层合并时造成的减益，使结果更好。因此，对于模式而言，Level 配置时不应令太多的层为 Tiering 模式，让 2 层左右的 Level 为 Tiering 模式更优。

3 结论

通过这次的 Project，我获得了了一些对大规模项目的编程经验，包括将一个较大的项目拆分为不同部分时的管理，编程和调试。同时，对这个数据结构和文件操作、多线程测试吞吐量等也有了全新的知识。这个项目的制作周期很长，但最后的结果还算令人满意。首先就是在解决各种各样奇形怪状规模各异的 bug 后，最终这个 project 能成功通过各个规模的正确性和持续性测试。但令人遗憾的一点是，它跑最大规模的正确性测试耗时比我预想中要长。我认为最大的时间开销应该就在我的 compaction 操作上，但我仍然对它的优化问题无从下手，所以就没有管他。除此之外，这个 project 的其他部分也算是圆满完成了。我也实现了 scan 的操作，但第二个额外项目因为我看不懂所以就没有实现。

4 致谢

感谢我的离散数学和工科创老师，允许我在他们的课上敲这个东西，也不来刻意点我名。

感谢互联网上众多分享自己问题的博主，我通过查阅无数资料并合并它们了解到了 C++ 中 FILE 的 open 模式后面还要加一个 b 来标识二进制，否则会出现奇奇怪怪的 bug。

感谢 chatgpt，解决了我网上查不到的问题，让我明白原来测吞吐量要用多线程来测而不是时延的倒数。

5 其他和建议

感谢我的暗影精灵，它十分钟自动熄屏安抚了不少我运行中的测试程序。

感谢我的 c++ 的文件读写经验的贫瘠。读写模式没加 b 的 bug 过于抽象，使我很难理解这是一个 bug 而不是灵异现象。

感谢运行时栈空间这个问题，我又花了数天才调试出来。我理解四位数深度的深搜会爆栈，但我怎么也理解不了我个位数的合并操作也会爆栈。

感谢 sort 函数，我花了数天到最后都没搞懂为什么 sort(**.begin(),**.end()) 会越界。