LSM Tree 实验报告

陈天予 519021910045

2021 年 5 月 30 日

1 背景介绍

LSM Tree(Log-structure Merge Tree) 数据结构，于 1996 年在 Patrick O’Neil 等人的一篇论文提出。其通过 SS-Table 的多层储存结构，利用磁盘顺序读写的高效性，实现了性能极高的写操作。LSM Tree 被广泛地在各种 NoSQL 中使用，比如 HBase，LevelDB 等。

2 挑战

1. 持久化：之前写过的程序，数据结构都在内存中，很少涉及到文件的读写。LSM Tree 通过客制化结构的 SSTable 实现持久化，需要通过二进制的方式来读写 sst 文件，因不熟悉 c++ 相关的库函数而导致的 bug 就不少。

2. sst 文件的 debug：由于 sst 文件在硬盘中，debug 的过程中无法实时看到其中的数据，造成许多麻烦和障碍。最终写了一个 peekSSTable 的小程序扫描各层 sst 文件并输出 debug 信息，解决了 debug 时的困难。

3. 一个难以复现的 bug：在调试过程中，有一次发现一个难以复现的 bug，复盘半天发现一个现象，如果两次 debug 时间间隔较长，这个 bug 就不会出现，反之就有很高机率出现。最终发现竟然是一个函数中忘记关闭文件，使得打开了过多的文件，在操作系统还未回收完这些 fd 前运行程序就有可能出现这个 bug。

4. 测试脚本和测试数据的构思：这应该是第一次接触较复杂的程序性能衡量，需要自己写测试脚本，与真实环境尽量相符；并通过不同的数据集，反应程序在不同时间，不同使用角度的性能。

3 测试

3.1 测试环境

AMD R7 3700x with SSD in Ubuntu 20.04 LTS

3.2 性能测试

3.2.1 测试方法

插入 1000 个随机大小在 1 Byte 到近 2 MB 的字符串（总数据量大约再 1G 左右），随后打乱所有 keys，执行 GET 操作，再打乱所有 keys，执行 DELETE 操作。重复上述方法连续测试三次，测试各方法的平均延时和吞吐。

3.2.2 预期结果

1. 对每次测试：测试结果应该相近。
2. 对 PUT 操作：随着插入数据量的增大，触发 compaction 的机率也越大，吞吐也越低。
3. 对 PUT 操作：随着 GET 数据量的增大，吞吐也会越低。
4. 对 DELETE 操作：由于 DELETE 操作相当于插入一个小字符串，所以对于不同大小的数据吞吐应该相同。但由于在 del 需要先判断 key 是否在数据库中，所以总体来说，其吞吐量趋势应与 PUT 类似，并稍低一些，因为其有可能触发 compaction 操作。

3.2.3 实际结果

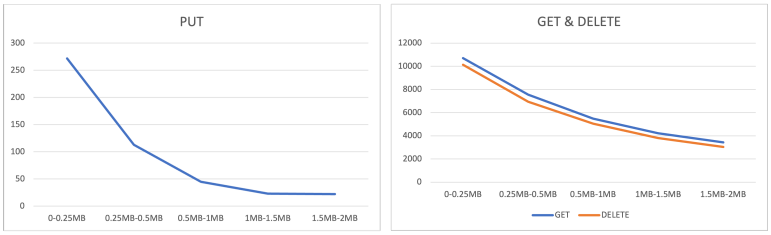
测试的结果如下：

```
>>>>> Put Test <<<<<
0...100...200...300...400...500...600...700...800...900...
```

```
Average Delay For Different Size Data:
0 ~ 0.25MB     : 3.94ms throughput: 254.00/s
0.25MB ~ 0.5MB : 8.80ms throughput: 113.58/s
0.5MB ~ 1MB    : 21.98ms throughput: 45.49/s
1MB ~ 1.5MB    : 42.43ms throughput: 23.57/s
1.5MB ~ 2MB    : 44.75ms throughput: 22.35/s

Average Delay: 28.00ms, Average Throughput: 35.72/s
Total Size Insert: 961MB

>>>>> Get Test <<<<<
Average Delay And Throughput For Different Size Data:
0 ~ 0.25MB     : 98.67µs throughput: 10134.66/s
0.25MB ~ 0.5MB : 137.13µs throughput: 7292.47/s
0.5MB ~ 1MB    : 180.61µs throughput: 5536.77/s
1MB ~ 1.5MB    : 238.53µs throughput: 4192.35/s
1.5MB ~ 2MB    : 292.84µs throughput: 3414.80/s

Average Delay: 203.35µs, Average Throughput: 4917.65/s

>>>>> Delete Test <<<<<
Average Delay And Throughput For Different Size Data:
0 ~ 0.25MB     : 100.34µs throughput: 9965.85/s
0.25MB ~ 0.5MB : 142.98µs throughput: 6993.74/s
0.5MB ~ 1MB    : 192.80µs throughput: 5186.74/s
1MB ~ 1.5MB    : 262.21µs throughput: 3813.78/s
1.5MB ~ 2MB    : 327.54µs throughput: 3053.05/s

Average Delay: 221.23µs, Average Throughput: 4520.18/s

>>>>> Summary <<<<<
It takes 28s to complete the test.
```

3.2.4 结果分析

测试结果与预期相符。不足之处在于 PUT 操作的测试颗粒度不够，对于小数据量的测试不够完全。



(a) GET (b) PUT & DELETE

图 1: 各操作对不同数据量的吞吐

3.3 索引缓存与 Bloom Filter 的效果测试

3.3.1 测试方法

测试脚本与性能测试相同，不同版本的代码存放在不同的 git 分支中，其中无缓存版本对应着 naive 分支，缓存索引信息的版本对应着 index_cached 分支，完全版本在 index_cached_with_bf 分支中。

3.3.2 预期结果

三种测试中 GET 操作的平均时延应有显著差距，但同种测试之间对不同数据大小的时延变化趋势应与性能测试中的相同。

3.3.3 实际结果

因篇幅限制，实际输出放在了 cache_test.md 中，这里只放图表。

3.3.4 结果分析

可以看到，三种不同的策略的吞吐量有显著区别。其中有无 Bloom Filter 在数据量较小时对性能影响较大，原因在于数据量较小时索引量较大，及时是二分查找也需要 log(n) 的时间，相比 Bloom Filter 的 O(1) 复杂度会逊色不少。值得一提的是，在无缓存的策略下，吞吐量反而随着数据量的大小

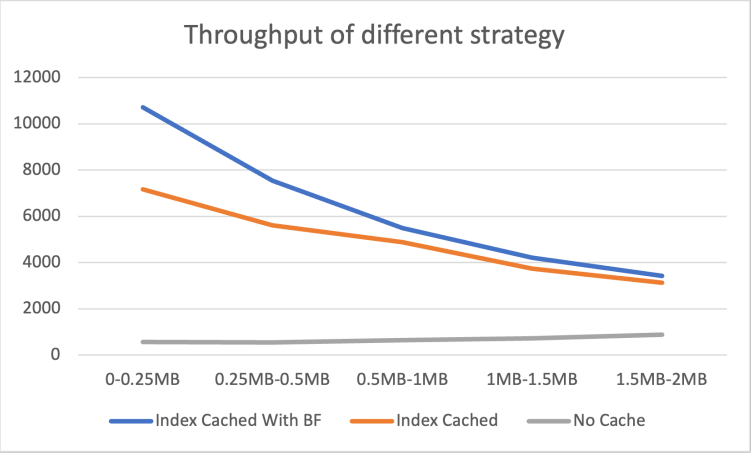


图 2: 不同策略下的吞吐量

上升而上升，猜测原因在于大数据量下一个 SSTable 中存的数据反而少，因此在索引过程所花费的时间反而较少。

3.4 Compaction 对 PUT 操作影响测试

3.4.1 测试方法

计算得出频率为每 10 次插入就需要 1 次 compaction 所需插入的数据大小，随后进行 5000 次插入（即使 compaction 大约 5000 次，数据量在 10G 左右），其中 key 的大小随机。测试脚本在 compaction_test 分支中。

3.4.2 预期结果

随时间的增长，因为层数的增加，每次 compaction 所需时间也越长。

3.4.3 测试结果

因篇幅限制，原始数据放在了 compaction-test.txt 中，以下只放了图：

3.4.4 结果分析

可以看到，吞吐量随时间的变化在局部上，呈现锯齿状，猜测每触发一次跨度较大的 compaction 就会导致该秒内吞吐量骤减；总体来说，呈阶梯

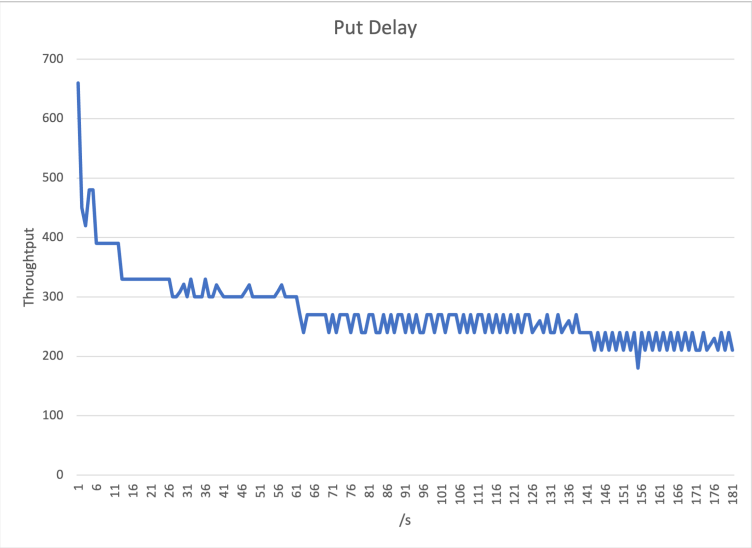


图 3: 吞吐量随时间变化

状递减趋势，猜测每个阶梯就代表了储存层数的增加。

3.5 结论

3.5.1 收获

1. 完整体验到了实现一个数据结构的流程：写代码，写调试工具，写测试工具。
2. 使用 latex 排版报告（非常整齐好看）。
3. 其余收获见挑战部分。

3.5.2 建议

1. 建议在测试脚本中增加自动清除本测试产生的数据的选项，需要手动清除很麻烦。
2. 建议提供校验和查看 sst 文件的工具，当然不提供也是对综合 debug 能力提升的一种考验。

3. 建议 SSTable 索引区的最后可增加一个空的索引，代表最后一个 value 结束的 offset，可以以很低成本，较大程度地简化读取键值对的逻辑（否则为了读取当前文件的最后一个值，还需要特殊判断是否是最后一个值，计算最后一个值的大小）