LSM-KV 项目报告

楼天驰 522031910290

2024年 6月 8日

1 背景介绍

LSM-KV 键值储存系统是一种以高性能执行写操作著称的数据结构，可以分为两部分——内存部分和硬盘部分，而硬盘部分又可以分为 SSTable 和 vLog 两部分，如图 1。其中内存部分是用一个跳表的结构来保存 Key-Value 数据对，SSTable（图 2左）中储存了时间戳、值范围、布隆过滤器以及对应 Key 在vLog中的储存位置，vLog（图 2右）则存储了真正的key-value对。LSK-KV 支持的操作有：PUT, GET, DEL, SCAN。

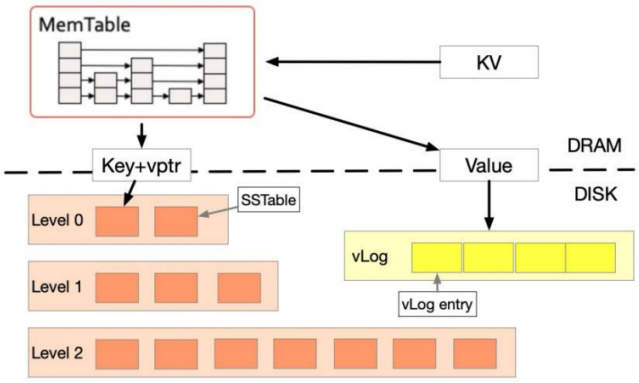
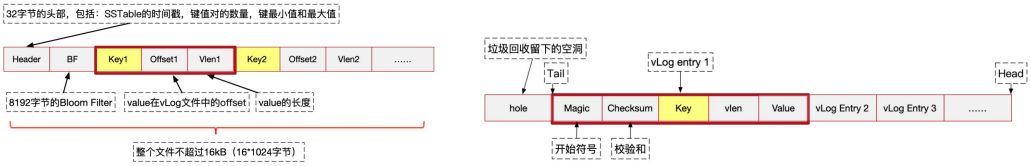


图 1: The structure of LSM-KV

在此Project的实现过程中，一是要掌握应用的各类数据结构，要写出完全正确无误的各个功能，同时要做好内存的释放，因为本次测试的数据量很大，一旦出现了内存泄漏的情况，就有可能导致出现各种奇奇怪怪的错误。

2 测试

此部分主要展现实现的项目的测试，主要分为下面几个子部分。



(a) SSTable (b) vLog

图 2: Detailed Structure

2.1 性能测试

2.1.1 预期结果

在测试之前，先进行理论分析：

1. 在仅使用内存中的跳表，即 memTable 时，所有操作都在理论上会达到最快的速度，但是这个阶段应该会非常短，基本可以忽略不计。

2. Scan操作要访问所有的SSTable，应该最慢；Get, Delete 操作的平均时延和吞吐应当相近，因为他们都有较大可能会有部分扫描SSTable的行为，但是由于Delete有可能进行写操作，会更慢；Put操作则应该更快，因为他涉及的硬盘行为更少，但是在触发Compaction时应当有一个明显的时延上升。总体平均下来，应该是 Scan >Delete >Put >Get。

3. 在数据量很大时（也是常规的情况），在未触发compaction时，时延应当为 Scan >Delete >Get >Put。

2.1.2 常规分析

表 1: 时延表现

	Put	**Get**	**Scan**	**Del**
Ave	1252883	32663	1682756022	75269
p50	1421	22027	909455835	46328
p90	14348	53073	4022008765	55684
p99	21755	110978	8377328527	180112

1. 包括Get、Put、Delete、Scan操作的延迟，测出了不同数据大小（2 000，20 000，50 000）时的操作延迟（以ns为单位），为了测试的合理性，每个数据大小测量然后计算出平均延迟以及各分位数的时延。

表 2: 吞吐量表现

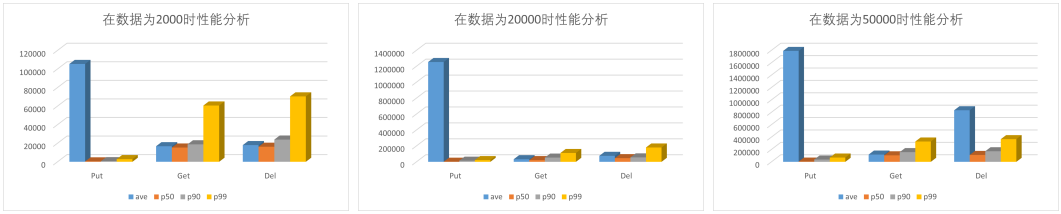
	Put	**Get**	**Scan**	**Del**
2000	9392.633	59156.57	28.067	52311.63
20000	786.95	17791.12	0.383	12603.82
50000	202.517	518044	0.033	353.1

2. 包括Get、Put、Delete、Scan操作的吞吐（指系统单位时间（按1s计）内能够响应的请求的次数），同样在以上不同数据大小下做了测试。

在初始数据量为20 000时，实验结果如表 1所示。在三种数据规模下的吞吐量如表 2。

可以发现，在这种情况下，scan的时延远大于其他操作，为了图片的直观性，在下面制作统计图时就不包括scan的信息了。

对比在三种数据量的情况下的数据统计图 3



(a) 2000 (b) 20000 (c) 50000

图 3: Normal Test Result

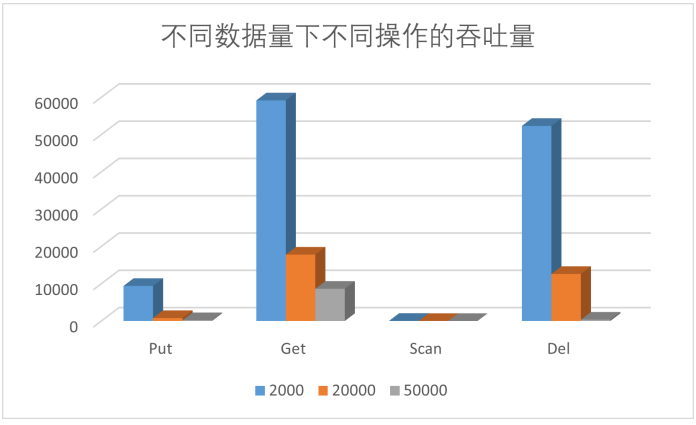


图 4: The throughput of different data size

从各个百分位数的时延来看，是符合我的预期，也就是 Scan >Delete >Get >Put。这是

因为对单一的一次常用的操作来说，性能排布确实应该是这样。

但是从平均时间来看，反而是Put操作的时延比Get和Del操作更高，并且50百分位数、90百分位数、99百分位数的时延都远低于平均的时延，我们从中就已经可以窥见Compaction操作所消耗的时间是非常大的，而且触发的次数是占比很小的。

而从吞吐量的角度来看（如图 4），也是基本一致的，且随着数据量的增加，每种操作的吞吐量都在下降，这是因为他们硬盘访问的数量都在上升，吞吐量也就相应的下降了。

整体而言，除了Compaction带来的影响，实验结果还是符合我的预期的。

2.1.3 索引缓存与Bloom Filter的效果测试

在取数据量为20000时进行测试，对比下面三种情况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
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

对以上三种不同的情况，绘制了平均时延的对比图（图 5）

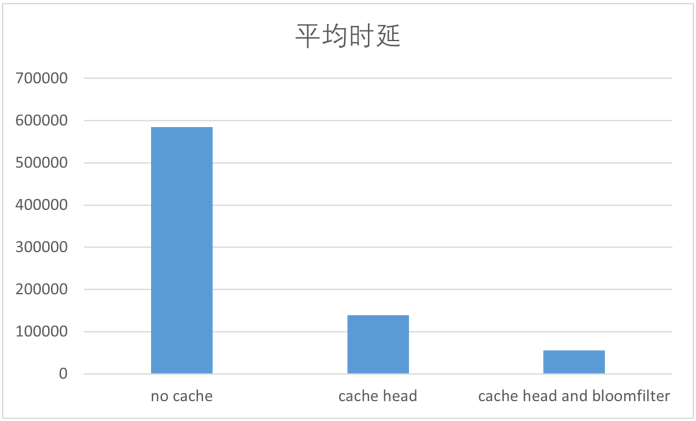
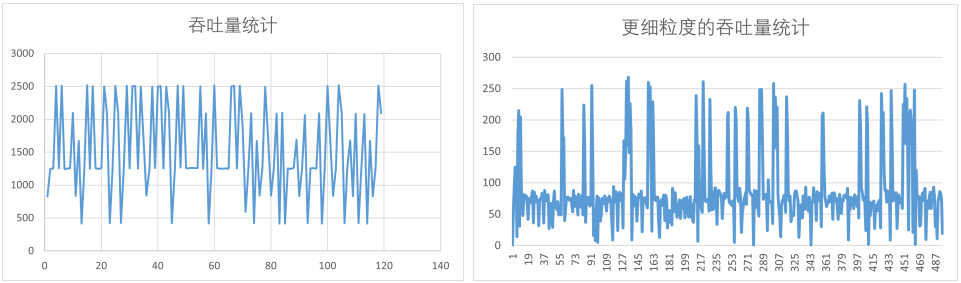


图 5: The average latency of different policies

从图中可以很明显地看出平均时延随着缓存的数据的增加显著地降低，缓存一个索引的提升更加明显，可能是因为每个SSTable的索引都会非常频繁地被访问到，因此成为了关键路径，而优化这一点就有非常大的性能提升。但是由于内存是有限的，我们需要在时间上的性能和空间上的性能之间达到一个平衡，不可能将所有数据都保存到内存中，这也不利于持久化。

2.1.4 Compaction的影响

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），并绘制其随时间变化的折线图如图 6：从折线图中我们可以发现，每隔一端时间，单位时间的吞吐量就会达到一个低谷，而这就意味着在这个时间范围内可能出现了多层的Compaction操作。此外由于我的计时器是会等待Compaction结束的，因此在实际情况下这段时间的吞吐量会更低（约为原来的1/3）。在更细粒度的测试中可以发现在一段时间中吞吐量接近于0，这应该就是在进行Compaction操作。而图中出现的高峰应该是level 0、level 1 中的SSTable都不满，因此可以较快地写入。



(a) 按 1s 统计　　(b) 按 0.5s 统计

图 6: Throughput Flow

2.1.5 Bloom Filter 大小配置的影响

Bloom Filter 大小的影响：Bloom Filter 过大会使得一个 SSTable 中索引数据较少，进而导致 SSTable 合并操作频繁；Bloom Filter 过小又会导致其 false positive 的几率过大，辅助查找的效果不好。尝试了7种 Bloom Filter 并保持 SSTable 大小不变，比较系统Get、Put操作性能的差异结果如图 7。

我本以为随着布隆过滤器的减小，平均的Put操作的时延不断减小，相应的吞吐量也会增大，而Get的平均时延则不断上升，相应的吞吐量也在减小，这种情况也是非常符合直觉的，因为布隆过滤器减小意味着每个SSTable可以存储的数据量、memTable的数据量都在上升，因而Put操作就会更少地触发硬盘操作，因此Put更快；同时，Bloom Filter出现误判的次数增加，访问SSTable的次数就会显著上升。

但是在实际测试后发现，在布隆过滤器的大小不断增大时，两个操作的平均时延都在上升，吞吐量都在减小。我认为可能的原因有以下几个：

1. SSTable的数量较少对时延和吞吐量的影响相对更大。

2. 由于测试时数据量并不是非常大，布隆过滤器的快速筛选的优势没有体现出来。

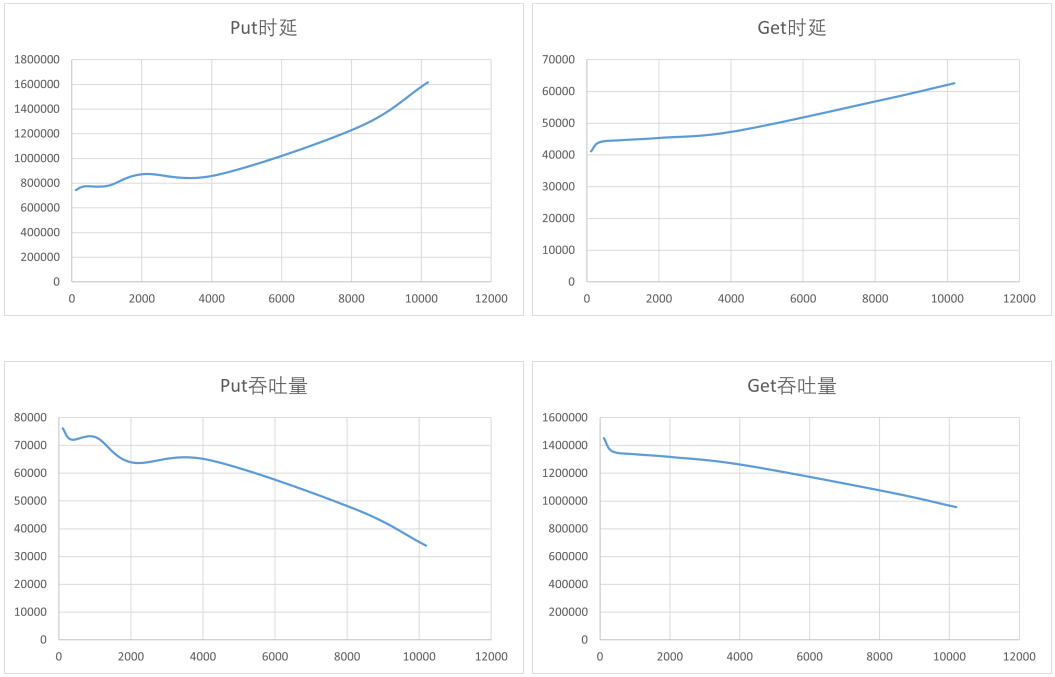


图 7: 测试结果

3. 由于我在访问SSTable获取offset时，使用了mmap的映射函数，使二分查找即使是随机访问但是其速度仍然比较快，即使没有布隆过滤器或是布隆过滤器误判，也可以较快地得出结果。

3 结论

到这里本次 Project 就算完成了，算是得到了一个比较好的结果。代码的编写、测试程序的编写、数据的获得与分析以及实验报告都已经完成。实验的结果也基本符合我的预期，有一些我考虑不周的地方、与预料的结果有出入的地方也有了一些合理的解释，同时加深了我对 LSM-KV 这个存储结构的理解，是非常有意义的。

本次 Project 是对我码力、Debug能力的一次检验，也是一次获得提升的机会，从中也对用到的跳表、布隆过滤器等数据结构有了更加深入的理解。很遗憾没有时间进行并行的实现，从本学期的学习中已经深深地感受到并行编程的高深与高效了。

4 致谢

感谢舒洪宇同学与我在各种操作上的思路碰撞，让我得以较快地完成此项目。感谢ChatGPT的帮助，让我在Debug无望时给我一些启发。

5 其他和建议

说实话这次 LSM Project 的编码并不算困难，在这学期过于摆烂的情况下仍然可以在一个周末左右的时间内完成全部的编码和Debug，但是报告和另一个ddl冲突了交晚了对不起 orz. 主要还是出现问题时的 Debug 异常复杂，因为在各种地方都有可能有细节没有注意到引发各种奇奇怪怪的错误。比如我除了常规的segfault还遇到了因为爆内存而直接导致进程被kill的问题，后来检查发现应该是指针释放的问题（经典）。但是一般来说，遇到问题时并不需要考虑的过多，一般都是在新增代码的地方出了问题，没有那么多是由于和原先代码组合出现的新的bug。当然，在帮舒洪宇同学debug的时候确实遇到了一个组合的bug，防不胜防啊！

不过这次的测试感觉还是有点复杂（不过这学期的lab基本上都有，笑），比如缓存部分的测试感觉挺复杂的，因此用了一些比较tricky的方法（实际上在内存上缓存了但是通过再次访问文件来模拟再次读取的操作），希望不会对测试结果有影响吧。