LSM KV System 实验报告

李逸岩 ID:519021911103

06 月 06 日 2021 年

1 背景介绍

Log Structured Merge Trees, LSM tree, 是一种键值对存储结构，现在有很多主流的数据库引擎都是基于 LSM Tree 构建的。LSM Tree 最大的结构优势是充分采用了计算机分级储存的思想 (as figure 1)，利用了现有存储介质顺序读写速率远大于随机读写速率的特点。前者表现在数据的主要部分存储在低成本，高访问延迟，低随机访问速度的 Disk 中，需要频繁读写的数据头 (as figure 2) 则储存在内存中，保证只访问内存能够得知是否需要访问 Disk；后者表现在 Disk 中的文件是不再改变的，不需要找到 Disk 中的某个数据进行擦写，避免了随机访问；而是永远顺序读写较大量的数据，大大提高了系统的吞吐量。

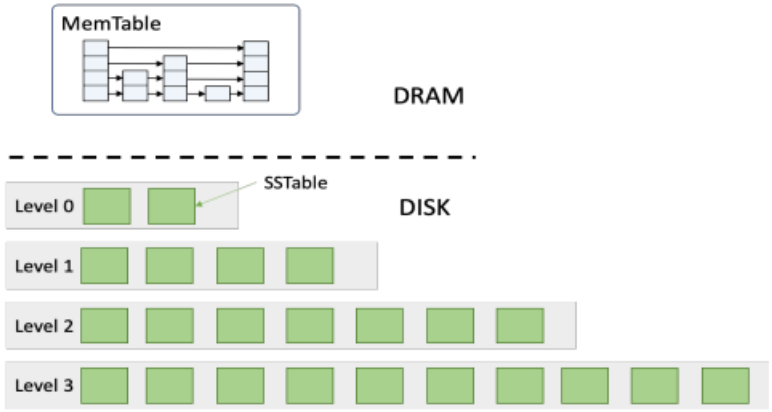

图 1: LSM KV structure: hierarchical level

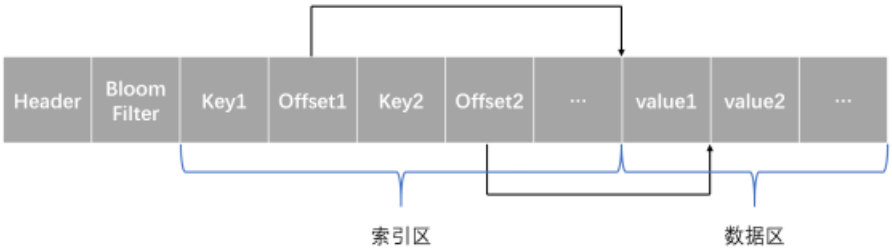

图 2: efficient header: search in memory

2 挑战

在第一阶段的最后阶段，我没能一鼓作气做完全部的工作。这导致第二阶段花了大量的时间去 Review 自己的代码，这时才感受到自己系统设计的缺陷；具体表现在代码非常难读，模块设计混乱。原因在编码前没有想清楚系统的架构，而是遇水搭桥，每次遇到问题或者觉得”能抽象复用“就编一个新的类出来。此外，这样不优雅的代码还造成了严重的内存泄漏。

在第二阶段下决心重构了自己的设计结构，创造了 sstable class，把很多有共性而且 tricky，易于出错，难以 debug 的功能在一个类中实现，将自己的 compaction 代码从原来的 750 行几乎全部删去重构到 200 行左右。此外大改了编码命名，使得类的实际作用和名字大体相符。出人意料地是，加上 debug 和调试，整个重构花费了不到一周的时间，让人有些震撼。

3 测试

3.1 测试平台

处理器 Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz
RAM 16.0 GB (15.8 GB 可用) 2400MHz
磁盘 1 KBG30ZMS512G NVMe TOSHIBA 512GB 容量:477GB
测试平台供电 (65W)，其他应用程序占用在测试时保持基本一致。

3.2 性能测试

3.2.1 预期结果

Overview 考虑到 Disk 和 Memory 访问的速度差距非常悬殊，主要从 Disk 访问次数来判断性能好坏。此外，连续读写磁盘的同一块区域和寻道后读写磁盘不同区域也应该被区分出来。

GET 由于缓存的存在，Get 最多读取一次磁盘。而 BloomFilter 和 Memory Cache 均能显著的提高查找时间，不过考虑到 BloomFilter 主要是在内存中快速排除某些 Cache node，不会增大对磁盘的读写，而若没有 Cache 存在则需要不停的读分散的磁盘区域，所以 Cache 对性能的替身应更为显著。

同时使用 Cache 和 BloomFilter 的 LSM-KV 读写时间应该近似与 (m+n) 有关，m 是磁盘固有查找延迟，n 是所查找的字符串长度。

PUT PUT 大部分情况下在内存中进行读写，认为消耗的时间非常小。但是有时会在磁盘中进行读写，可能会造成较大的延时。最大时间应该和 Compaction 的最大层数有关。每层具体的读写大小是 k 个文件，k 在系统运行一段时间后会大于 3（和第 0 层设计有关），上限则和最新读入的 key scope 有关。

DELETE DELETE 相当于一次 GET 和一次 PUT，由于 PUT 的字符串长度很小，可以认为 PUT 不会导致 Compaction(近似认为是附近的大长度 PUT 导致的)，所以 DELETE 的时间复杂度约等于 GET。

3.2.2 常规分析

1. 测试策略

   宏定义 PTIME 为测试放大倍数，定义下面的各项范围。其中，value 长度由随机数指定，模拟真实的读写情况，限定下界为 128Bytes 以贴近实际应用。

key scope	0 - 32768 * PTIME
insert scope	0 - 32768 * PTIME / 2
value size scope	128 - 65536 Bytes
no value scope	32768 * PTIME / 2 - 32768 * PTIME
operation	16384 * PTIME

2. 测试工具

   测试工具使用 windows.h 下的 QueryPerformanceCounter 和 QueryPerformanceFrequency 来判断程序的操作时间。查文档可知误差约为 $0.5\mu s$ 级别，因此使用 μs 作为计量单位。

3. Get、Put、Delete 操作的延迟

   数据结果如下：

PTIME	Item	PUT	GET	DELETE	SIZE
1	Total	1.91E+07	1.28E+06	710674	1517MB
16384	Average	1.16E+03	7.80E+01	4.34E+01	LEVEL 8
2	Total	2.50E+07	1.61E+06	713683	1007MB
16384	Average	7.64E+02	4.90E+01	2.18E+01	LEVEL 8
3	Total	4.20E+07	1.68E+06	1.06E+06	1897MB
16384	Average	8.55E+02	3.42E+01	2.16E+01	LEVEL 8
4	Total	6.32E+07	2.27E+05	1.44E+06	2080MB
16384	Average	9.64E+02	3.46E+00	2.19E+01	LEVEL 9

   可以看出，在 $PTIME = 2 - 4$ 的情况下，Latency 趋于稳定。特别是 $PTIME = 3$ 时，与前后数据大小的 Latency 都比较接近。在硬盘存储大小约为 2GB 的情况下，compaction 层数约为 8 层，此时 PUT 操作的平均 Latency 为 $855.0\mu s$，GET 操作的平均 Latency 为 $34.2\mu s$，DELETE 操作的平均 Latency 为 $21.6\mu s$.

4. Get、Put、Delete 操作的吞吐

   在单线程的情况下，吞吐量是所求 Latency 的倒数。根据公式：

$$Throughput = \frac{1}{Latency} \tag{1}$$

   可以计算出三个操作的吞吐量为：

Item	PUT	GET	DELETE	SIZE
Total	4.20E+07	1.68E+06	1.06E+06	1897MB
Average	8.55E+02	3.42E+01	2.16E+01	LEVEL 8
Throughput	1.17E+03	2.93E+04	4.64E+04	

即 PUT 操作吞吐量为每秒 1170 次，GET 操作吞吐量为每秒 29300 次，DELETE 操作吞吐量约为每秒 46400 次。

3.2.3 索引缓存与 Bloom Filter 的效果测试

需要对比下面三种情况 GET 操作的平均时延

1. 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。这里不使用 Bloom filter，只是在查找的时候略去了查找 bloom filter 这一步。为了便于测试，选取 PTIME=2, 3。数据如下：

PTIME	Item	GET
2	TOTAL	1.01E+06
	Average	3.09E+01
3	TOTAL	1.73E+06
	Average	3.52E+01

Latency 大约在平均 33μs. 可以看出，较 Release Version 差距不大 (反而小，这会在结论中说明)。说明 Bloom Filter 不是降低延迟的根本原因，这与我们的预期是相符的。

2. 内存中不缓存 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。这里稍微做了一些优化，首先是对有序的层做了二分查找，减少硬盘的读取次数，其次是如果 key 不在本 sst 文件的范围内就只读 32bytes 的 header 后终止。为了便于测试，选取 PTIME=3，数据如下：

PTIME	Item	GET
2	TOTAL	4.12E+08
	Average	1.26E+04
3	TOTAL	9.04E+08
	Average	1.84E+04

Latency 大约在平均 15000μs. 可以看出，较 Release Version 差了约 2-3 个数量级，较 With Bloom Filter 版本差了约相同的 2-3 个数量级。可以看出，sstable 头部缓存在内存中能造成极大的性能提升，这与我们的预期一致。

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。这就是最后上交的版本，测试数据在 3.2.2中展示。

3.2.4 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，测试需要表现出 compaction 对吞吐量的影响，故稍微调整了测试策略。这次操作不模拟实际情况，只为了查看 Compaction 影响，因此固定大小为 262144 Bytes，这样可以保证约 8 次 PUT 可以触发一次 Compaction，使得效果明显。

得到结果如下。Figure 3是我根据 Latency 和出现频率绘制的。纵轴是出现频率，横轴是时间范围，应当注意在中间偏下有一条很矮的数据条。显然更高的数据条是不含 compaction 的 PUT 操作的 Latency，他们占了绝大部分。而远远更低的数据条是含有 compaction 操作的 PUT 的 Latency，他们只是占了一小部分，随着 compaction level 的逐渐增大，Latency 也逐渐增大。为了看得更清楚，我绘制了靠底部的放大版 Figure 4. 可以看出绝大部分的 PUT 操作 Latency 都在 $10^3 - 10^4 \mu s$ 量级。(读者会意识到放大后靠左的柱子有多高。) 而含有 compaction 的 PUT 操作 Latency 在 $1 * 10^5 - 2.5 * 10^5$ 之间，要比正常情况下大 1-2 个数量级。

4 结论

本次 Project 完成了一个相对高效，代码风格有进步的 LSM-KV Store System，并且做了性能测试，大部分的测试结果符合预期。

对结果主要说明两点。其一，DELETE 的性能优于 GET，但是 DELETE 的操作却是 GET 的超集。我认为，在测试脚本中连续 DELETE 能显著降低 Compaction 的次数 (DELETE PUT 插入的 value 长度很小)，从而提高 PUT 的效率；而 GET 的效率也要更高。所以有这样的结果。其二是消去

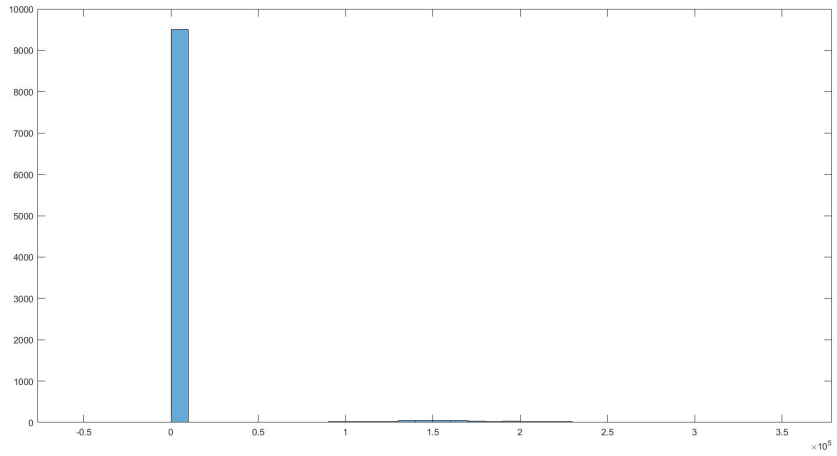

图 3: frequency vs. Latency

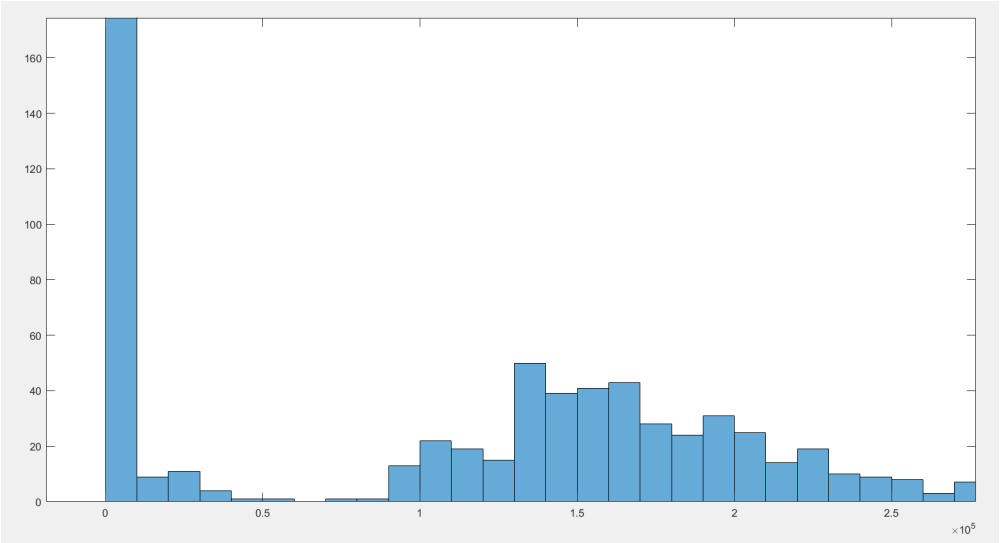

图 4: Enlarge the lower left corner

了 Bloom Filter 之后部分样例下 Latency 居然变小了。原因并不明确，我认为有可能是样例设计使得命中率比实际工业界生产时更大导致，使得使用 Bloom Filter 反而 Latency 增大，得到的收益反而不及因为验证 bloom filter 产生的支出；或者是因为二者测试平台并未保持一致：在 Without Bloom Filter 测试中，未开启”Run in Terminal”，限于篇幅就不复现测试环境了，

但这也有可能是原因之一。

这次的 Lab 算是本学期挑战收获最多的 lab 之一。主要面对的挑战就是系统的设计。这教会我很多。一位著名 OI 选手说过'Think twice, Code once.'Coder 可能要在很多次重构后才能明白这句话的意义。除此之外我学会了文件位级的读写操作，Debug 的技巧，性能测试的方法等，这些也算是收获。但最大的收获还在于明白了写出高质量代码的关键就在把绝大多数时间花在构思设计而不是调试打补丁上；只希望下次有这样的 project 我能一次写出优美，漂亮，高质量的代码。