LSM-KV 项目报告

卢骏宸 520021910544

2022 年 04 月 29 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，如今在各大公司中都得到了广泛的应用，Google 的 LevelDB 和 Facebook 的 RocksDB 都以 LSM Tree 为核心数据结构；而在本项目中，我们需要开发一个基于 LSM Tree 的简化键值存储系统。

不同于以往绝大多数的编程项目，本次 LSM-KV 的开发对于内存与磁盘的交互以及对于 I/O 操作的理解提出了更高的要求，在保证正确性的前提下，需要合理地设计磁盘的读写来提高键值存储系统的性能。LSM-KV 支持的操作有：

- 插入以及删除键值对 (K, V)
- 读取键 K 的值以及查找特定区间内所有的键值对

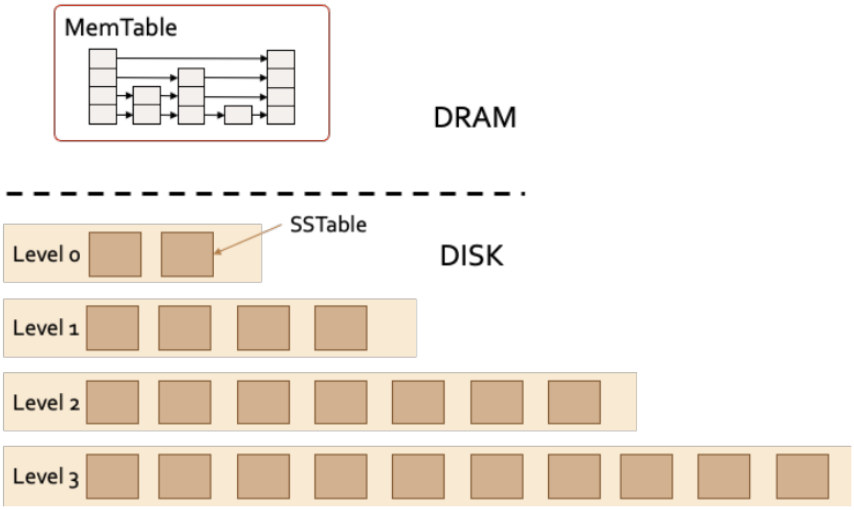

图 1: LSM-Tree 键值存储系统结构

2 数据结构和算法概括

2.1 内存数据结构：跳表

我们设计的内存核心数据结构是“跳表”，跳表具有一系列良好的性能，除了可以高效的进行增删改查等基本操作，跳表相比于哈希表等传统内存数据存储系统在范围查找上具有明显的优势，其原因在于跳表对于历史查找具有记忆功能。基于种种原因，我们选择跳表作为 LSM-KV 内存的核心数据结构。

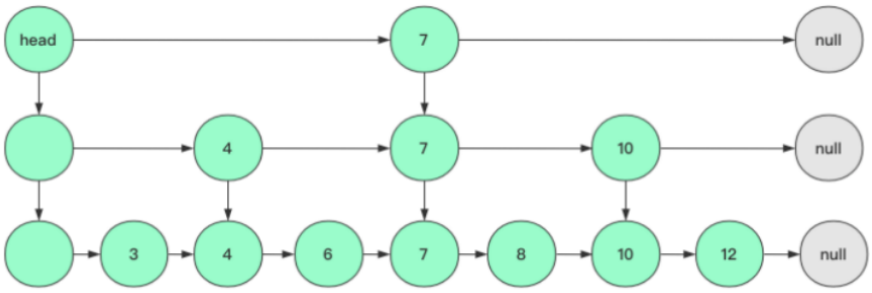

图 2: LSM-Tree 键值存储系统结构

2.2 磁盘数据结构：Bloom Filter

磁盘上每个文件大小的上限为 2MB，每个文件中记录着如下信息：文件生成的时间戳，键值对的数量，键的最小值和最大值，Bloom Filter，以及所有存储的键值对。其中键值对分为索引区和数据区存储，其中索引区记录着对应键值的偏移量。为了加速每个文件内部的查找，在文件头部会加入一个 Bloom Filter，通过一个哈希映射函数可以快速判断特定键值是否在当前文件中有对应键值对存储。

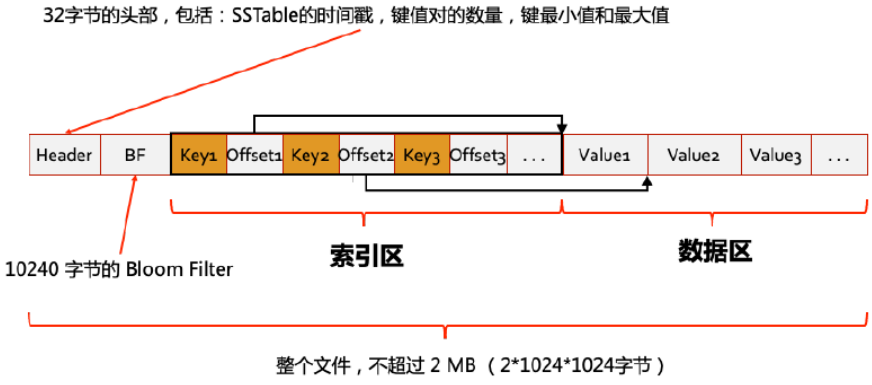

图 3: 磁盘上的文件结构以及 Bloom Filter

2.3 磁盘文件组织：递归合并

磁盘文件采取分级存储结构，每一级的文件个数以 2 的指数级增长，当某一级文件发生了溢出，系统会统一采取如下的措施来解决文件的溢出，整个处理显然是一个递归过程，在具体代码实现的时候可以采取递归算法进行函数调用：

- 在诱发文件数量溢出的文件夹中选取时间戳、键覆盖范围最小的文件进行向下合并
- 在下一层中选取与目标键覆盖范围有交集的文件进行统一合并，将排好序的键值对以 2MB 分配给新生成的文件
- 如果下一层文件数量依旧超过上限，则仿照之前过程进行循环操作，直至所有层级均达到平衡

在具体代码实现过程中，基本的增删改查操作相比起传统做法均有类似的思想，而磁盘文件的递归合并则能够很好的体现 LSM-KV 键值存储系统的特色所在。

3 LSM-KV 系统性能测试

3.1 预期结果

3.1.1 PUT(K,V):

在插入键值对的过程中，如果插入数量较少（几百个），则系统内存的跳表部分就足够来记录所有的插入，在最后关闭系统（代码上体现为类的析构函数）时将所有的键值对写入一个大小不到 2MB 的文件之中；如果插入数量较多（几万个），则在插入的过程中会有“间歇性”的延迟现象。造成这种现象的原因在于，内存操作远快于文件的读写，所以跳表的插入远快于 I/O 操作，每一次 2MB 文件的写入会造成程序发生长达数秒的“中止”。

3.1.2 DELETE(K,V):

删除过程在本系统的设置中本质上也是一个插入字符串“ DELETED ”的过程，所以在性能上理应与插入有相同的表现。但是在特定的测试集写法中，两者存在一定的区别。由于“ DELETED ”字符串相比于现实生活中很多存储的字符串更加短小，这就意味着在一个 2MB 文件中就可以记录很多的键值对删除记录。进一步地，该文件键所覆盖的范围就会很大（几万个），所以如果在大规模的删除之后再进行插入操作就会造成糟糕的时间性能。

3.1.3 GET(K)/SCAN(K1, K2):

许多数据结构以及规定都会努力使得查找具有较快的速度，例如设置 Bloom Filter 快速判断特定文件中是否存在特定键值对，各层级文件中键值对均是有序存储且各文件键覆盖范围没有交集，数量溢出时优先向下合并时间戳小的文件等等。范围查找 SCAN 与查找 GET 道理类似。

3.2 常规分析:

3.2.1 PUT(K, V):

为了测试对于不同数据大小插入过程的平均延迟以及吞吐，我们设计的测试程序如下：同样是插入 1000 个数据，而数据依次分布在 0-999，10000-10999，20000-20999，30000-30999，40000-40999 这五个范围之内；可以预期看到，如果数据取值较小，则插入可能仅仅影响内存跳表部分，插入速度较快，但是如果数据分布很大，那么对应的 I/O 操作次数就会增加，插入速度减慢。

【注】所有的性能测试程序都是基于 correctness.cc 和 persistence.cc 进行修改。

PUT(K, V)	平均延迟(毫秒)	平均吞吐(次/秒)
0~999	0.003	333333
10000~10999	0.075	13333.3
20000~20999	0.169	5917.16
30000~30999	0.222	4504.5
40000~40999	0.345	2898.55

图 4: 插入操作平均延迟与吞吐量的实验数据

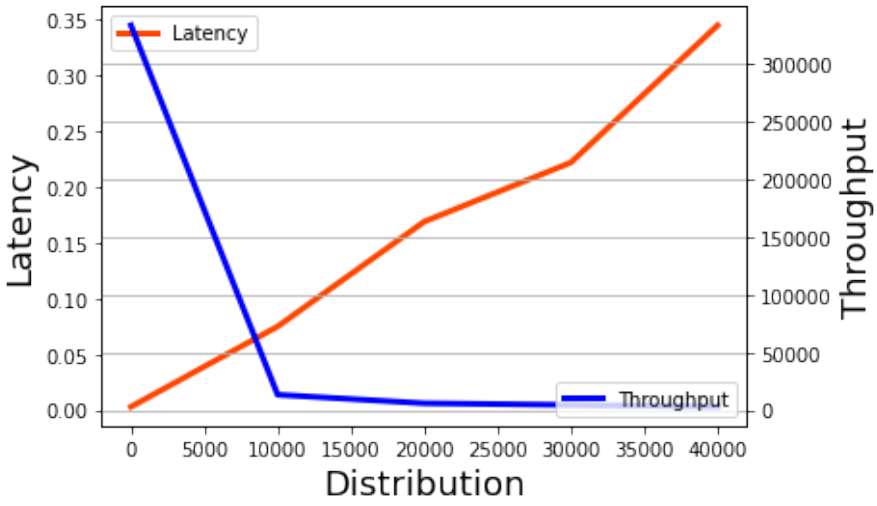

图 5: 随着数据取值增加插入操作的性能变化

以上的两幅图表可以印证我们的结论，当插入数据大小到达数千时，数据存储开始从内存向磁盘过渡，这样就会导致平均插入延迟明显增加以及平均吞吐量明显降低。

3.2.2 GET(K)/DELETE(K):

此处将查询以及删除合并在一节中论述是主要基于以下考虑，在性能实验过程中发现，这两类操作在时间性能上随时间并没有显著的变化。虽然从理论上来看，删除操作本质上应该与插入操作相类似，但是稍加分析后我们会发现：连续的删除操作并不会使数据区的字符串呈现爆炸式的增长，所以删除并不会造成过多的向下溢出或者是合并操作，这就导致了数据的大小对于删除的性能并不会造成太大的影响。

GET(K)	平均延迟(毫秒)	平均吞吐(次/秒)
0~999	0.213	4694.84
10000~10999	0.224	4464.29
20000~20999	0.224	4464.29
30000~30999	0.228	4385.96
40000~40999	0.225	4444.44

图 6: 查询操作平均延迟与吞吐量的实验数据

DELETE(K)	平均延迟(毫秒)	平均吞吐(次/秒)
0~999	0.219	4566.21
10000~10999	0.219	4566.21
20000~20999	0.227	4405.29
30000~30999	0.238	4201.68
40000~40999	0.236	4237.29

图 7: 删除操作平均延迟与吞吐量的实验数据

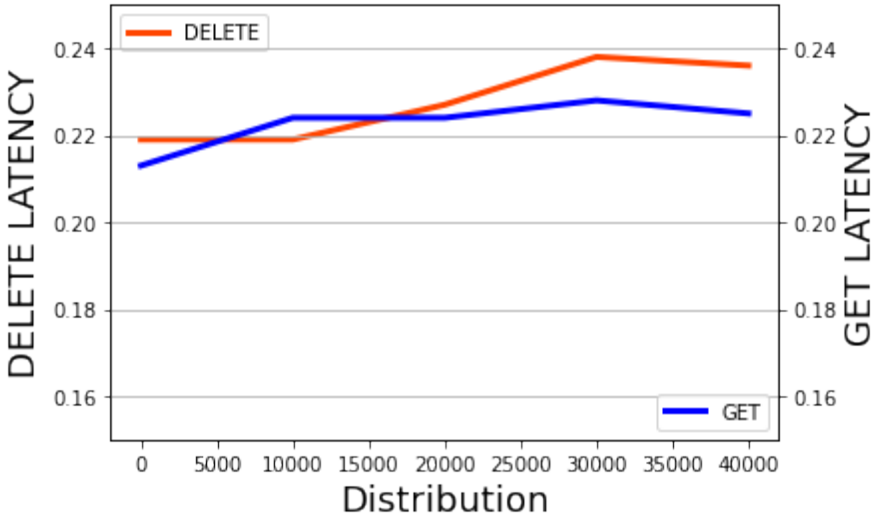

图 8: 查询与删除的性能比较以及数据大小造成的影响

3.3 索引缓存与 Bloom Filter 的效果测试

3.3.1 测试结构以及预期结果

首先明确我们的比较目标是 GET 操作的平均时延，对于我们实现的 LSM-KV 系统，内存中有每个磁盘文件的索引缓存结构体，而每个结构体中记有除了数据区以外的全部信息；这种存储设计使得该系统在不访问磁盘的前提下就可以极大的获得全部有用信息。但是，如果内存中的索引缓存并不存在，理论上查找的性能就会受到严重的影响，那么我们就通过以下 3 种情形来改写我们的 LSM-KV 键值对存储系统来量化这种影响的大小：

- 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
- 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
- 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

这三种设计方案从上到下在内存种保存的信息是由少变多的，理论上查找操作的平均时延是由长变短的。第三种情形已经非常类似于我们现在的设计，但是细微的差别在于文件头部的时间戳，键值对的数量以及键所覆盖的范围。这些信息对于查询以及向下合并文件等过程均会产生一定的益处，若遗漏该部分信息预计也会有一定的影响。

3.3.2 实验数据记录以及预期结果验证

通过修改程序部分的接口以及实现方式，我们发现结果的变化趋势较为符合我们之前的预期，能够较好的反映我们的理论分析：

GET(K) 平均时延	0~999	10000~10999	20000~20999	30000~30999	40000~40999
修改1	1.572	1.693	1.648	1.825	1.803
修改2	0.633	0.704	0.728	0.697	0.805
修改3	0.382	0.388	0.394	0.402	0.398
最佳版本	0.213	0.224	0.224	0.228	0.225

图 9: 索引缓存与 Bloom Filter 对查找性能的影响测试

3.4 Compaction 合并操作的影响

本节的实验目的是，在不断插入数据的情况下，统计每秒钟处理的 PUT 请求个数，并绘制其随时间变化的柱状图。可以让键值对中 value 占用的空间大一些，从而提高 compaction 的频率。

```
void regular_test()
{
    clock_t begin_time, end_clock;
    begin_time = clock();
    double time;
    int second = 0;
    uint64_t prev=0, curr=0;

    for (uint64_t i=0; i<1024*64*32; ++i) {
        store.put( key: i, s: std::string( Count: 1000, Ch: 's'));
        end_clock = clock();
        time = (static_cast<double>(end_clock - begin_time) / CLOCKS_PER_SEC);
        if (time >= 1) {
            second++;
            curr = i+1;
            std::cout << "Total time is: " << second << " s" << std::endl;
            std::cout << "Number of PUT instructions: " << i+1 << std::endl;
            std::cout << "Number of PUT instructions in the last second: " << curr-prev << std::endl;
            begin_time = clock();
            prev = curr;
        }
    }
}
```

图 10: 探究 Compaction 影响的测试程序

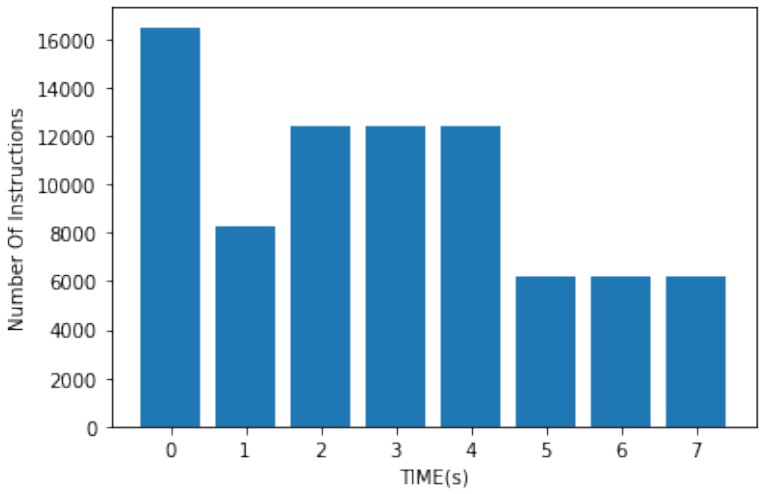

图 11: 每秒钟运行的指令数量

对上述结果进行简单的分析可以看到，每秒钟执行的 PUT 指令数量随着程序的执行有明显的下降。除了在一开始存在一定的波动的现象，柱状图整体呈现下降的趋势。但是这种现象并不能完全的否定 LSM-KV 系统的时间性能，因为随着程序的继续运行（例如 1 分钟以后），每秒钟运行的指令数量仍然能够很好的保持在 6000 左右的水平。从这个简单的实验中，LSM-KV 的时间持久性得到一定的体现。

3.5 针对内存数据结构的对比试验

本 LSM-KV 键值存储系统中的内存数据结构采用的是跳表，而另外一种方法是使用 std::map 实现 MemTable。在本节中，我们拟通过实验数据，比较这两种实现对键值存储系统 PUT/GET/DELETE 这三种操作的性能影响，并得出相关结论，探索每一种实现方式的适用情景。（这里选取平均时延作比较）

	平均时延	跳表	std::map
PUT	0~999	0.003	0.005
	10000~10999	0.075	0.226
	20000~20999	0.169	0.496
	30000~30999	0.222	0.528
	40000~40999	0.345	0.664
GET	0~999	0.213	0.325
	10000~10999	0.224	0.312
	20000~20999	0.224	0.334
	30000~30999	0.226	0.332
	40000~40999	0.225	0.341
DELETE	0~999	0.219	0.282
	10000~10999	0.219	0.274
	20000~20999	0.227	0.279
	30000~30999	0.238	0.284
	40000~40999	0.236	0.297

图 12: 跳表与 std::map 的性能对比

通过以上的数据对比，我们可以粗略的得到以下一系列结论：LSM-KV 系统的一大优势就是具有较好的写性能，这种数据结构能够以高性能执行大量写操作，而这一点在本项目中也得到了比较好的反映，如果搭配理想的内存数据结构（如跳表优于 std::map），就能够有效地降低 PUT 操作的平均时延；在 GET/DELETE 操作上，虽然跳表同样具有优势，但是在时间上的差距并不是很明显，所以如果一个系统需要执行大量的查询或者删除操作，也可以基于以上实验数据合理地选择内存数据结构。

4 结论与总结

在本文最后的总结部分，我主要会列举出本项目在实现过程中，我犯下的一些主要错误：

- 本实验需要设定大量的指针，并且需要在内存堆上通过动态分配获得大量的空间（new），如果在编程时在使用完指针之后忘记回收堆上的空间（delete），就会造成严重的内存泄露问题。以前在一些小型的项目中，如果忘记回收并不会造成严重的问题，因为不大的问题规模并不会利用内存的所有资源，每次运行结束之后系统会帮助我们自动回收空间；但是一旦问题规模扩大，内存泄漏的问题就会被无限的扩大，导致 bad alloc 之类的报错。
- 本项目的另外一个特点就是大量的磁盘访问 I/O 操作，我们在设计代码的过程中应该时刻考虑如何尽可能的去减少磁盘的访问次数。例如项目文件中要求建立索引缓存就是处理这个问题的一个有效的方法；在我们自己书写的过程中，如果同样不注意 I/O 速度的问题，就会导致极差的时间性能（从原本 2 3 分钟到超过 1 小时）。如何能够减少磁盘访问也是一个值得思考的问题。
- 大项目的 debug 过程也是非常的麻烦与枯燥的，但是在看似枯燥的过程中，我们应该主动的去思考在哪些地方去设置断点，去猜测程序可能存在的问题并且想办法去验证自己的猜想。

5 致谢

本次 LSM-KV 项目已经初步完成，感谢在整个项目的完成过程中给我提供帮助以及灵感的老师、助教、同学等等，如果需要进一步去研究这个键值对存储系统也欢迎大家一起讨论！