LSM Tree 实验报告

郭志东 519021910703

6 月 7 日 2021 年

0 术语定义

在 compaction 时，SST 数量溢出的一层称为*上层*，查找与上层 key 相交的 SST 所在的一层称为*下层*。

1 背景介绍

虽然说 LSM 树的名字中带一个“树”字，但这个数据结构并不是树，而是一种分层、层内有序存储的数据结构。这种数据结构牺牲了部分读性能（通过 Bloom Filter 可以缓解），但是大幅提高了写性能，因为在将 Mem Table 的数据写入内存、或是进行 compaction 时都是顺序读、写磁盘，而磁盘的顺序访问速度远大于随机访问速度，将键值对组织成 SST 的分块形式也可以有效降低磁盘访问次数。

LSM 树的增删改查过程在文档中已有叙述，在报告中不再重复。这里介绍一些文档中没有提到的实现细节：

1. 除了第 0 层，其他层的 SST **按照 key 的区间（不相交）升序排列**，这样在一层内查找 SST 时，也可以使用二分查找，进一步加快查找速度，但是这样 Bloom filter 的作用就被**弱化**了。

2. 在做 compaction 时，如果*上层*有多个 SST 溢出，那么对于**每个***上层*溢出的 SST，在*下层*中查找与其 key 相交的 SST，并进行合并；而非先将*上层*的**所有**溢出的 SST 区间合并，再在*下层*中查找与其相交的 SST。选择这种设计，主要是考虑到第二种设计方法可能导致过多的键值对同时存在于内存中，内存无法容纳；第一种设计虽然会加大编码难度，降低 compaction 速度，但是提高了系统的稳定性。

2 挑战

1. 作为一个存储系统，在实现时需要尽可能地提高效率。为此，在我的 LSM 树的实现中，除了顶层的接口，内部的各种参数传递都尽可能使用了引用或指针。这就导致了 SSTable 的内部存储结构类型比较复杂：

   ```
   vector<shared_ptr<vector<shared_ptr<SSTable>>>>
   ```

   即便使用了很多 typedef，在遍历时还是很容易产生类型错误。

2. 在开发过程中，耗时最多的 bug 来自于 compaction 时新的 SST 的时间戳计算方法。一开始董学长在课程群里说 SST 的时间戳不会相同，因此我就以为每个新的 SST 时间戳都是当前最大的时间戳加 1，但这种计算方法实际上是错误的。因为这会将 compaction 下层与上层 SST 相交的所有 SST 的都更新为“最新的”，这会将一些实际上不是最新的键值对错误地更新为最新的，导致查找时产生错误。正确的 SST 时间戳计算方法是取所有 SST 中时间戳最大的那个，这样也可以确保上层 SST 的时间戳一定大于等于下层，确保了从上层查找到下层的正确性。

3. 该项目的 debug 较为复杂，需要查看 SST 的内容进行 debug，每次 debug 完眼睛都非常酸。

3 测试

此部分主要是展现你实现的项目的测试，主要分为下面几个子部分。测试部分应当是文字加上测试数据的图片展示。

3.1 性能测试

3.1.1 预期结果

1. 常规实现的 Get、Put、Delete 延迟分析

   随着值的大小增大，每个 SST 可以容纳的键值对个数减少，SST 的个数增加、写磁盘的频率增加、compaction 的频率增加，所有操作的延

迟都应该增加、吞吐量降低。在三种操作中，Put 的平均延迟应该较高，因为 Put 有可能会触发 compaction，而 compaction 是非常耗时的操作。Get 和 Delete 的操作延迟应该接近，且 Delete 的延迟应该高于 Get，因为 Delete 就相当于 Get 并插入一个删除标记，插入删除标记时有可能触发 compaction。

2. 索引缓存与 Bloom Filter 的效果延迟分析

   如果内存中不缓存 SSTable 的信息，每次读取都从磁盘中读取，那么速度必然远远慢于常规实现。

   使用 Bloom Filter 不一定会提高 Get 的性能，因为第一，每一层的 SST 按照 key 升序排列，可以直接进行二分查找；第二，在使用 BloomFilter 之前可以先通过更节省时间的比较 min/max key 来确定 key 是否存在于一个 SST 中，所以真正使用到 Bloom Filter 的机会非常少。实际上，Bloom Filter 占据的额外空间和读写开销甚至可能降低 Get 的速度。

3.1.2 常规分析

测试方法：总共进行 5 组测试，每组的值（字符串）长度分别为 **50、500、5000、50000**。每组测试的键的构成都相同，为 **1 到 10000** 的数组随机打乱。每组测试进行 4 轮，并对统计数据取平均值。在每轮测试中，

1. 打乱键数组，并进行 10000 次**插入**，计算耗时和吞吐量
2. 打乱键数组，并进行 10000 次**查找**，计算耗时和吞吐量
3. 打乱键数组，并进行 10000 次**删除**，计算耗时和吞吐量

需要注意到的是，当值的大小为 50 时，不会发生写入磁盘，所有操作都在跳表中进行；当值的大小为 500 时，SST 会被写入第 0 层，但不会发生 compaction；其他组数据都会发生 compaction。

最后的结果如表 1所示。从表中可以看出，

	操作 / 数据大小	PUT	GET	DELETE
平均时延（秒）	50	4.08148×10^{-6}	1.2527×10^{-6}	5.62268×10^{-6}
	500	4.0366×10^{-6}	7.14342×10^{-6}	1.19999×10^{-5}
	5000	8.25781×10^{-5}	1.08449×10^{-5}	1.73969×10^{-5}
	50000	$1.80043\text{s}\times10^{-3}$	3.77659×10^{-5}	3.16451×10^{-5}
	500000	1.98109×10^{-2}	3.07034×10^{-4}	2.64466×10^{-4}
吞吐量	50	245009	798276	177851
	500	247733	139989	83334
	5000	12110	92210	57481
	50000	555	26478	31601
	500000	50	3257	3781

表 1: 各操作的平均时延与吞吐量

1. 当不发生写磁盘时，Put 和 Delete 操作速度接近，Get 操作最快。这可能是因为在跳表中，Put 和 Delete 都需要下到底层，并进行数据的删除/修改，而 Get 查找到最顶部的元素即可返回，且不需要写数据。

2. 当发生写磁盘但不 compaction 时，Put 快于 Get 和 Delete。这是因为 Put 每次都写入 MemTable 中，且总共只会写入两次磁盘，均摊下来开销微乎其微（所以吞吐量和不写磁盘接近），而 Get 和 Delete 每次都有可能访问磁盘。

3. 当发生 compaction 时，Put 操作的吞吐量就会小于 Get 和 Delete，而且数据越大，差距越大。这足以证明 compaction 确实会对性能产生较大的影响。然而，Delete 的吞吐量在值的大小较大时反而会大于 Get，其可能的原因有数据的随机性，以及在实现中，Get 相较于 Delete 多了一次按值传递的过程，value 的大小越大，按值传递的开销就越大。

3.1.3 索引缓存与 Bloom Filter 的效果测试

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

测试方法：总共进行 2 组测试，值的大小（字符串长度）分别为 **5000** 和 **50000**。每组测试的键的构成都相同，为 **1 到 10000** 的数组随机打乱。对每种实现进行 4 轮测试，在每轮测试中：

1. 打乱键数组，并进行 10000 次**插入**

2. 打乱键数组，并进行 10000 次**查找**，计算平均时延

为了确保在 SST 不缓存时每次读索引都是从磁盘访问，每次读取数据都是用全新的 fstream 对象，读完一次就 close。（因为 fstream 的 read 函数是 buffered 的）测试结果如表 2所示。从表中可以看出：

实现方法 / 数据大小	无 SST 缓存	无 Bloom Filter	常规实现
5000	1.93822×10^{-3}	1.07938×10^{-5}	1.0067×10^{-5}
50000	4.44185×10^{-3}	4.00327×10^{-5}	4.0013×10^{-5}

表 2: GET 操作在不同实现下的平均时延（秒）

1. 直接从磁盘读数据确实会极大地降低速度

2. 使用 bloom filter 反而会使速度降低。这与之前的理论分析相符合：由于在使用 bloom filter 之前，已经进行了二分查找和检查 min/max key，使用到 bloom filter 的频率非常低。如果在上述测试中输出 bloom filter 的使用次数，可以发现在两组测试中，对于 10000 次查找，bloom filter 分别仅被使用了 9984 次和 31336 次，使用频率确实是极低的。

3.1.4 Compaction 的影响

测试方式：为了确保每次的 compaction 时间足够长，便于从图中观察，将值的大小固定为 128，持续 Put**60** 秒，key 为使用

```
std::default_random_engine r(time(nullptr))()
```

随机生成的随机数。为了排除计时的开销，在子线程中进行时间的计算和统计。测得结果如图 1所示。

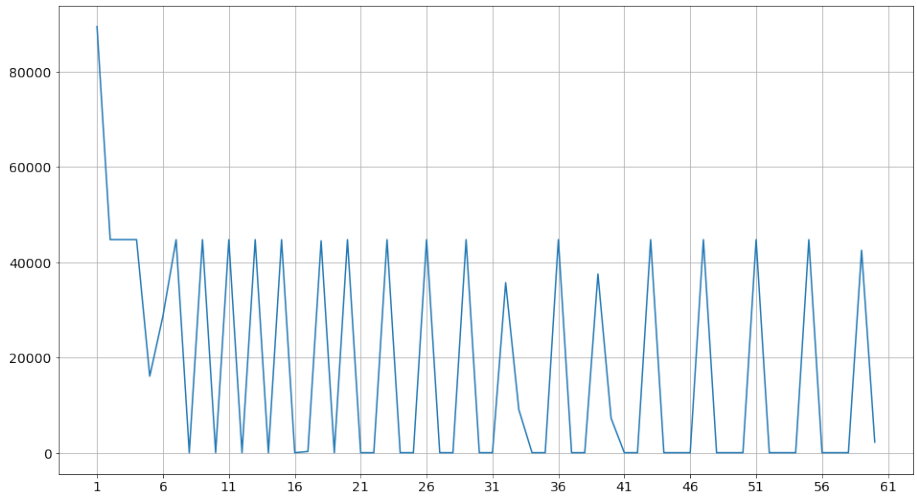

图 1: 吞吐量折线图

从图中可以看出，在开始的 5 秒内吞吐量快速下降，这是因为一开始 mem table 为空，所有写操作只需要写入内存即可，而且一开始 compact 合并次数较少。在这之后，折线图呈现出锯齿形的形状，通过在程序 compaction 时输出信息（如图 2）可以看出，即便在锯齿的顶部仍有 compaction 发生，但是每当 compaction 发生一定次数后就会有一次极为耗时的 compaction，使程序在数秒内吞吐量为 0，而且随着时间增加，compaction 的耗时越来越长，这和实际情况是符合的。

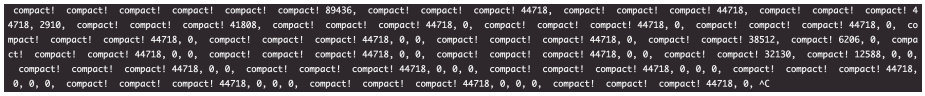

图 2: 吞吐量测试程序运行截图

4 结论

通过这次 project，我：

1. 巩固了 C++ 的各种语法，例如 lambda 表达式、运算符重载、模版、智能指针、宏
2. 对磁盘和内存的读写速度差异有了直观的体会
3. 练习了 C++ 的文件操作
4. 体验了如何通过实验测试数据结构的性能，并将结果与实现细节对应
5. 领悟到了不要别人说什么就是什么，要自己思考为什么数据结构是怎么操作的、为什么是这么操作的

建议：助教提供的测试数据可以更复杂一些，这次给的数据甚至无法触发二路归并（一个 SST 的 key 总是全部小于或大于另一个），有的同学可能会因此以为自己实现对了，但实际上并没有。