LSM Tree 实验报告

王浩天 519021910685

6 月 3 日 2021 年

1 背景介绍

内存存储结构被称为 MemTable，其通过跳表或平衡二叉树等数据结构保存键值对。

硬盘存储采用分层存储的方式进行存储，每一层中包括多个文件，每个文件被称为 SSTable（Sorted Strings Table），用于有序地存储多个键值对（Key-ValuePairs），该项目中一个 SSTable 的大小为 2 MB。

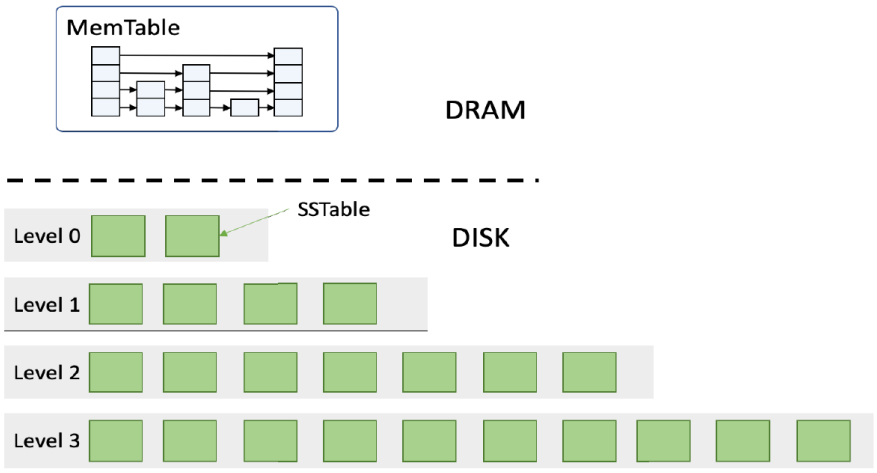

图 1: LSM Overview

每个 SSTable 文件的结构分为四个部分:

1. Header 用于存放元数据，按顺序分别为该 SSTable 的时间戳（无符号

64 位整型），SSTable 中键值对的数量（无符号 64 位整型），键最小值和最大值（无符号 64 位整型），共占用 32 B。

2. Bloom Filter 用来快速判断 SSTable 中是否存在某一个键值，本项目要求 Bloom Filter 的大小为 10 KB （10240 字节）。

3. 索引区，用来存储有序的索引数据，包含所有的键及对应的值在文件中的 offset (无符号 32 位整型)。

4. 数据区，用于存储数据 (不包含对应的 key)。

2 挑战

2.1 设计

本项目整体设计已由课程助教统一指定，但是在细化数据结构、设计类等方面都有一定难度。

具体实现时，我设计了 MemTable、BloomFilter、SSTable 三个主要的类，MemTable 通过跳表实现，BloomFilter 封装了 set、match 的接口，SSTable 通过计算 offset 实现 key-value 对应关系。

至于硬盘中的 Block 在内存中的缓存，我采用链式结构存储每层的缓存，用指针数组存储每层的索引，实现 Blocks 在 Memory 的元数据拷贝。

2.2 Bugs

1. key-value 配对错误：由于 offset 计算失败导致 key-value 配对错误。

2. 二分法查找 key 错误：函数退出的边界条件设计不合理，导致找不到 key

3. 时间戳判断问题：Compaction 计算 Range 的策略不合适，导致部分数据被裹挟产生时间戳歧义，在调整了时间戳策略、key-range 计算后修复此 bug

4. 数据写入失败：在达到阈值后写入硬盘，但是最后一个数据未成功写入导致数据丢失。

3 测试

对于 LSM Tree 存储系统，我设计并进行了一系列测试，以分析不同数据情况对系统性能的影响。

3.1 预期结果

经过理论分析，我认为在标准版本中 key 的分布以及 value 的长度对于 put latency 有较大影响，del 和 get 操作的效率基本不受影响。

而在测试索引缓存与 Bloom Filter 的版本中，get 操作的效率主要受到存储方式的影响，而且影响较大

3.2 常规分析

3.2.1 Part I

测试 Get、Put、Delete 操作的延迟，设计并进行了如下实验：

```
optimization-3 common  test:
put latency:[0.560028]  del latency:[0.063141]  get latency:[0.057648] (ms per operation)
gap test test:
put latency:[0.887634]  del latency:[0.067749]  get latency:[0.062805] (ms per operation)
random key test:
put latency:[1.407471]  del latency:[0.063446]  get latency:[0.057953] (ms per operation)
random value test:
put latency:[0.431274]  del latency:[0.063751]  get latency:[0.057037] (ms per operation)
fixed length 2048 bytes test:
put latency:[0.051817]  del latency:[0.063253]  get latency:[0.051130] (ms per operation)
fixed length 3072 bytes test:
put latency:[0.077668]  del latency:[0.057022]  get latency:[0.051014] (ms per operation)
fixed length 4096 bytes test:
put latency:[0.114849]  del latency:[0.057878]  get latency:[0.053606] (ms per operation)
fixed length 5120 bytes test:
put latency:[0.127320]  del latency:[0.057766]  get latency:[0.052330] (ms per operation)
fixed length 6144 bytes test:
put latency:[0.155222]  del latency:[0.058573]  get latency:[0.053526] (ms per operation)
fixed length 7168 bytes test:
put latency:[0.178821]  del latency:[0.059389]  get latency:[0.053381] (ms per operation)
fixed length 8192 bytes test:
put latency:[0.196432]  del latency:[0.058718]  get latency:[0.053835] (ms per operation)
fixed length 9216 bytes test:
put latency:[0.220816]  del latency:[0.097284]  get latency:[0.053869] (ms per operation)
```

图 2: Test Result

1. 常规测试，从 1-MAX 进行顺序 Get、Put、Delete，velue 长度等于当前 key 值，每个迭代的 key 增量为 1。
2. gap test，从 1-Max 进行顺序 Get、Put、Delete，value 长度等于当前 key 值的 2 倍，每个迭代的 key 增量为 2。
3. random key test，生成 key 范围为 1-MAX 的无重复序列，乱序插入，value 长度为定长，同时确保总数据量大小和前述测试相等。
4. random value test，从 1-MAX 进行顺序 Get、Put、Delete，velue 长度为随机数但 value 总大小同前述测试，每个迭代的 key 增量为 1。
5. fixed length test，每次迭代 key 增量为固定值，value 长度为预设固定值，但保证 value 总大小同前述测试

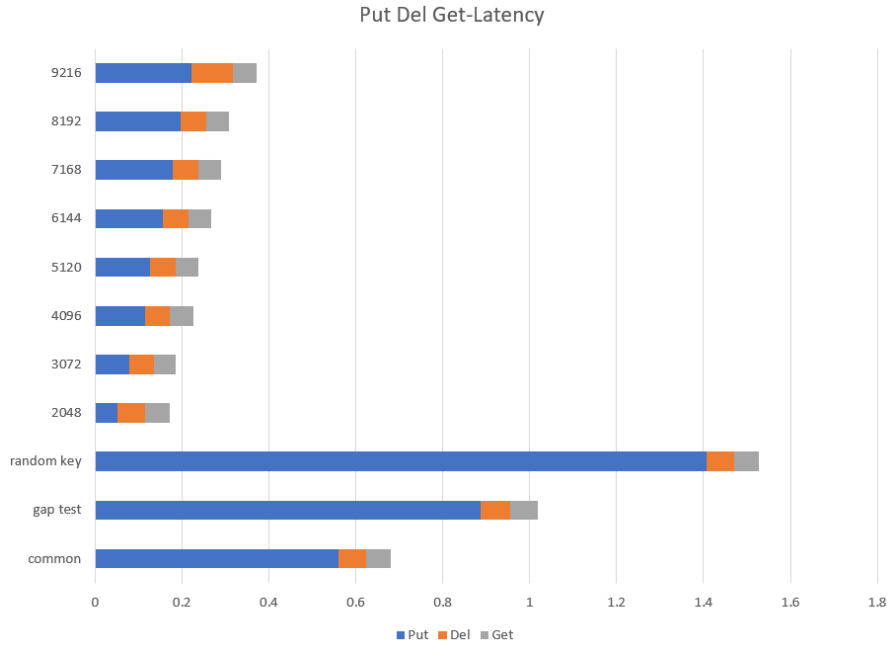

图 3: Common Test Visualization

观察数据可知，Put 操作 latency 变化范围较大，Del 和 Get 操作的 latency 变化不大。

Put 操作需要写入硬盘，时间开销较大。在数据总量均相等的情况下，compaction 次数基本相等，与 compaction 操作的开销相比，写入内存、写

入硬盘时间占比很小，因此数据长度越短、写入操作越频繁，平均时延就越短。

Del 操作的实现是向 LSM Tree 系统写入一个“DELETED”，数据长度很短，造成 compaction 次数相比 Put 操作很少，因此整体的 Del 操作 latency 很短。

Get 操作不需要写硬盘，只需要对内存中存的 SSTable 的元数据进行搜索，所以一次 Get 操作只会进行一次读操作。在搜索元数据时，采用 Bloom Filter 和二分查找的方式，时间消耗很少，因此不同测试集下 Get 操作的 latency 近似相等且耗时很短。

3.2.2 Part II

本阶段测试了不同数据总量下的平均 latency。实验设计为 key 值总量相等、每次测试中 value 长度不同、Fixed-Length 模式的顺序读写测试，输出如图 4。

对结果进行可视化，从图中我们可以发现，在不同数据规模的情况下，Put 的平均 latency 与数据量有较大关系，基本呈线性相关。而 Del、Get 与数据量基本没有关系。

```
amount test :16384 bytes per value test:
put latency:[0.334045]  del latency:[0.060181]  get latency:[0.055359] (ms per operation)
amount test :18432 bytes per value test:
put latency:[0.382629]  del latency:[0.061401]  get latency:[0.060852] (ms per operation)
amount test :20480 bytes per value test:
put latency:[0.455688]  del latency:[0.061401]  get latency:[0.057739] (ms per operation)
amount test :22528 bytes per value test:
put latency:[0.505066]  del latency:[0.063232]  get latency:[0.057739] (ms per operation)
amount test :24576 bytes per value test:
put latency:[0.600586]  del latency:[0.063293]  get latency:[0.058960] (ms per operation)
amount test :26624 bytes per value test:
put latency:[0.623535]  del latency:[0.065063]  get latency:[0.059570] (ms per operation)
amount test :28672 bytes per value test:
put latency:[0.701965]  del latency:[0.065674]  get latency:[0.060791] (ms per operation)
amount test :30720 bytes per value test:
put latency:[0.748047]  del latency:[0.066284]  get latency:[0.061462] (ms per operation)
```

图 4: Amount Test Result

3.3 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

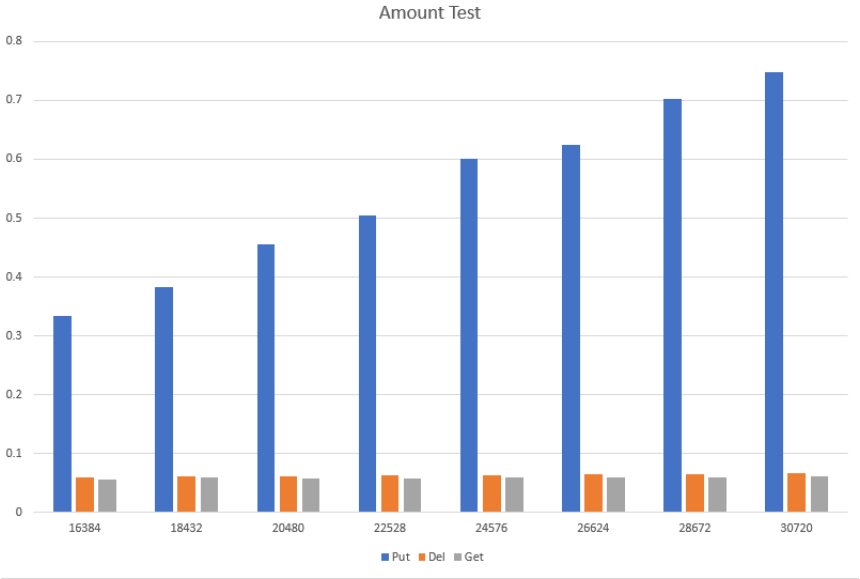

图 5: Amount Test Visualization

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

本部分实验设计采取 Random-Key、Fixed-Length 的方案，乱序插入 Key 后进行顺序 Get，排除顺序插入对 Bloom Filter 的影响。测试结果如图 6，可视化效果如图 7。

后两种情况下，Get 的访问速度并无明显差异，而第一种明显要慢很多，与后两者相比不在一个数量级。

第一种情况对应 Opt-1。由于内存中并无 SSTable 的索引区信息，也无 Bloom Filter，每次访问文件都需要进行硬盘读操作，在文件总量为 n 的情况下需要读取 $O(n^2)$ 个文件，故非常耗时。

第二种情况对应 Opt-2。内存中只有 SSTable 的索引，没有 Bloom Filter，

```
optimization-3 common  test:
put latency:[1.023346]  del latency:[0.056610]  get latency:[0.051697] (ms per operation)
optimization-2 common  test:
put latency:[0.955048]  del latency:[0.056915]  get latency:[0.054138] (ms per operation)
optimization-1 common  test:
put latency:[0.959595]  del latency:[0.103607]  get latency:[6.179352] (ms per operation)
```

图 6: Indexes and Bloom Filter Caching Test

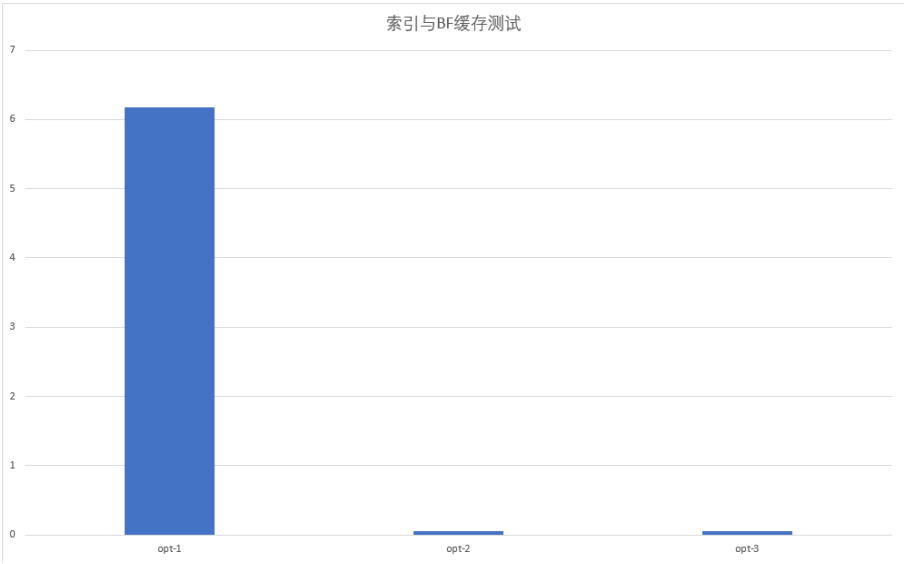

图 7: Caching Test Visualization

在这种情况下直接通过二分查找寻找 offset，Get 时延略高于第一种。但是由于 SSTable 索引直接存放在内存中，访问速度较快，故并无太大的时延增长。

第三种情况对应 Opt-3。内存有 Bloom Filter 和 SSTable 索引，通过两次优化加快了查找 key 的速度，时延最短。

3.4 Compaction 的影响

使用 Random-Key、Fixed-Length 模式，消除每次 Compaction 区间的局部性以及 Compaction 频率不同造成的影响。不断插入数据的情况下，测定单次操作的 latency，绘制出 latency 随插入操作次数变化的柱状图 8。

从柱状图可以看到，latency 的具有一定周期性，会周期性地出现峰值。

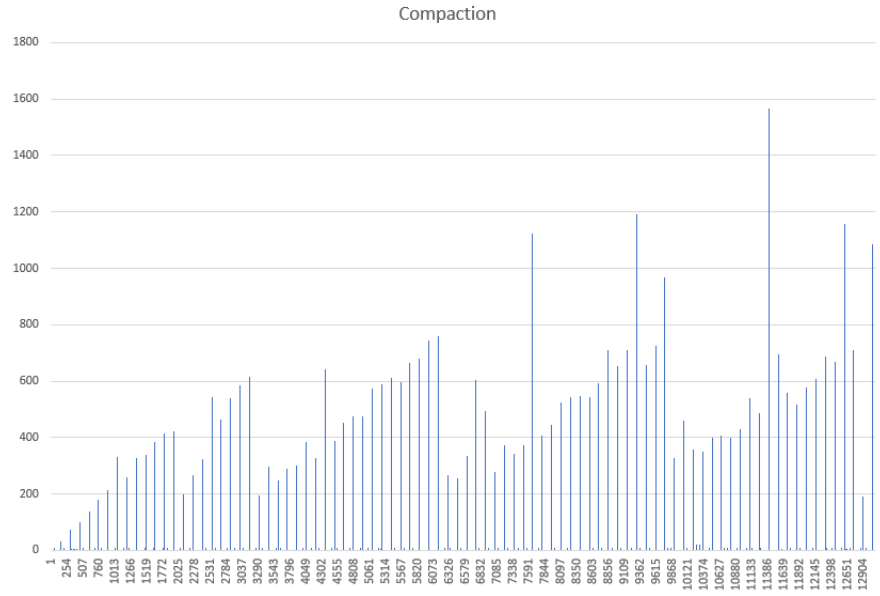

图 8: Compaction Test Visualization

分为小峰和大峰：小峰为写入硬盘的 latency，大峰为发生 compaction 的 latency，一般的插入 memTable 的 latency 基本可以忽略不计。

Compaction 呈现逐步增长的周期性变化。周期性源于固定的插入次数造成 Compaction 的时延，而逐步增长则是随着一个周期内插入数据逐步变多，造成周期内 Compaction 涉及的文件集变大，因此 latency 会逐步增长。

4 结论

在这次实验中，我深入理解了 LSM-Tree 的原理和存储方式，对以这种方式存储的数据库结构有了更深层的认识。

初步探索 LSM Tree 存储结构后，我认为 LSM Tree 的 Put latency 主要与数据量有较大关系，这与系统的层级结构与 Compaction 有关。Get 操作主要与存储方式有关，在内存中存储 SSTable 的索引可以极大优化 Get latency。Del 操作的 latency 的基本为常数，这是因为 Del 操作可以通过向 memTable 插入“DELETED”标记解决，十分简洁。

同时在本次实验中我也遇到了一些挫折和挑战，克服这些困难使我的能力大大提升，这次经历是人生中一笔不可多得的财富。