LSM Tree 实验报告

康艺潇 518431910002

6 月 9 日 2021 年

1 背景介绍

1.1 LSM 树

LSM 树，即日志结构合并树 (Log-Structured Merge-Tree)。是一种可以高性能执行大量写操作的数据结构。支持增、删、读、改、顺序扫描操作。而且通过批量存储技术规避磁盘随机写入问题。LSM 树和 B+ 树相比，LSM 树牺牲了部分读性能，用来大幅提高写性能。

1.1.1 设计思想

LSM 树的设计思想非常朴素：将对数据的修改增量保持在内存中，达到指定的大小限制后将这些修改操作批量写入磁盘，不过读取的时候稍微麻烦，需要合并磁盘中历史数据和内存中最近修改操作，所以写入性能大大提升，读取时可能需要先看是否命中内存，否则需要访问较多的磁盘文件。

1.1.2 基本结构

LSM Tree 键值存储系统分为内存存储和硬盘存储两部分，采用不同的存储方式 (如图 1 所示)

内存存储 本次实验选择使用跳表，支持增删改查。值得注意的是：其中的插入操作需要覆盖原先的值。删除操作需要查看内存和当前跳表，即使存在也需要插入一个特殊字符串” DELETED ”。

硬盘存储 当内存中 MemTable 数据达到阈值 (即转换成 SSTable 后大小超过 2MB) 时，要将 MemTable 中的数据写入硬盘。

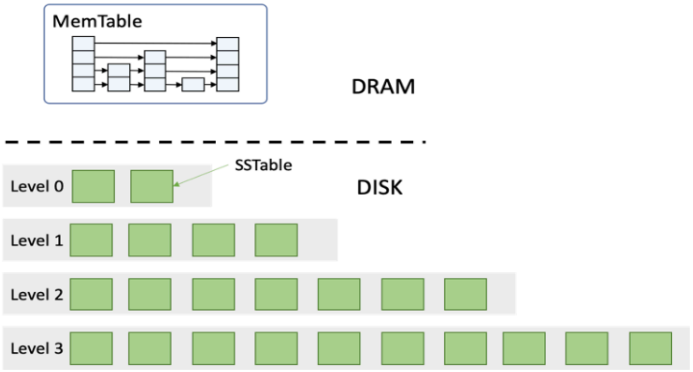

图 1: LSM-Tree 键值存储系统结构

2 挑战

2.1 困难

2.1.1 对 merge 的理解

merge 操作情况很多。对于第一层和最后一层以及中间层需要不同的处理。对其思路整理画出流程图如下：

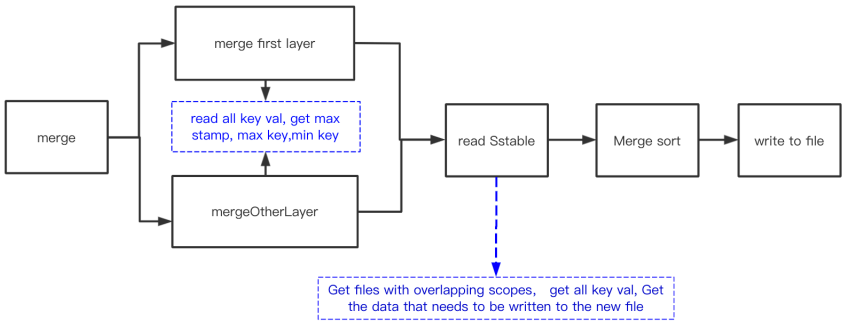

图 2: merge 相关流程

内存泄露 发现内存不断扩大，存在内存泄露。于是使用了 xcode 的内存泄露检测工具 Leak。找到相关函数，查找资料发现 vector 的 clear 不能完全清空内存，需要用 swap 函数清空。然而还是没有解决问题，最后发现在 skiplist 中使用了之前已经释放的指针。通过这个了解了管理内存的重要

性以及学会了检测内存泄露。如图是 Leak 使用图。

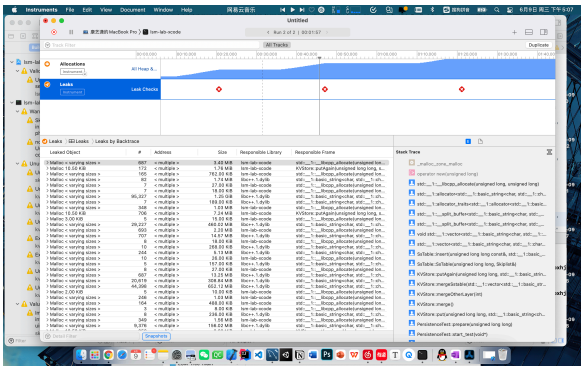

图 3: Leak 工具

可以查看内存泄露的函数

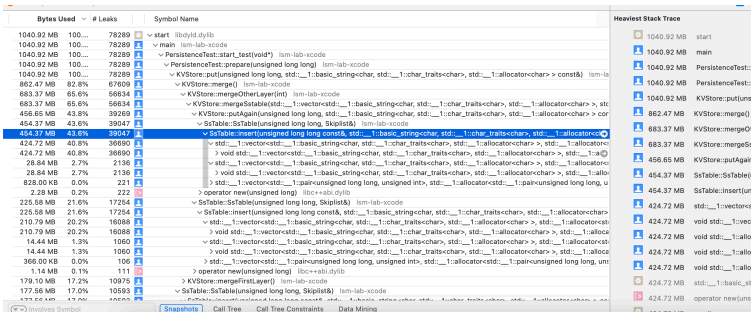

图 4: 内存泄露的函数

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. 三种操作的吞吐量和时延应为倒数关系。删除操作的流程为先查找是否存在。之后再 PUT 一个删除的标志。可能会触发合并，如果每次插

入的数据比较小，那么均摊到每次删除的操作的附加值较小，又因为 DEL 插入的字符串很小，只有 9 个字节，很少会触发合并，因此字符串很长的 PUT 操作时延会显著高于 DEL。而字符串小的时候，DEl 比 PUT 时延高。

2. 通过 BloomFliter 直接判断中是否有查询的 key，这个操作是 O(1) 的。可以避免无用的查找，因此会提高很多 GET 性能。

3. 如果缓存了索引，那每次读取数据的时候就不用先读索引，而是在内存中找。然后直接定位到相应的位置。磁盘的速度比内存慢很多个数量级，在内存中缓存 index 可以显著地加快查找的速度。

3.1.2 常规分析

数据大小	8000	12000	16000	20000	25000
GET	0.0305	0.0336	0.0356	0.0553	0.0886
PUT	0.3456	0.4543	0.4542	0.5875	0.9086
DELETE	0.0343	0.0325	0.0354	0.0453	0.0758

图 5: 各个操作的延迟，单位：ms

【参数】：每次的数据大小如表格所示，i 从 0-max, 每次插入长度为 i 的数据。
【讨论】：可以看见在数据规模较小的时候，没有进行合并操作，DEL 操作的时间比 PUT 高一些，当数据量足够大后，PUT 操作的平均时延比 DEL 低了，这符合上述的讨论。

3.1.3 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

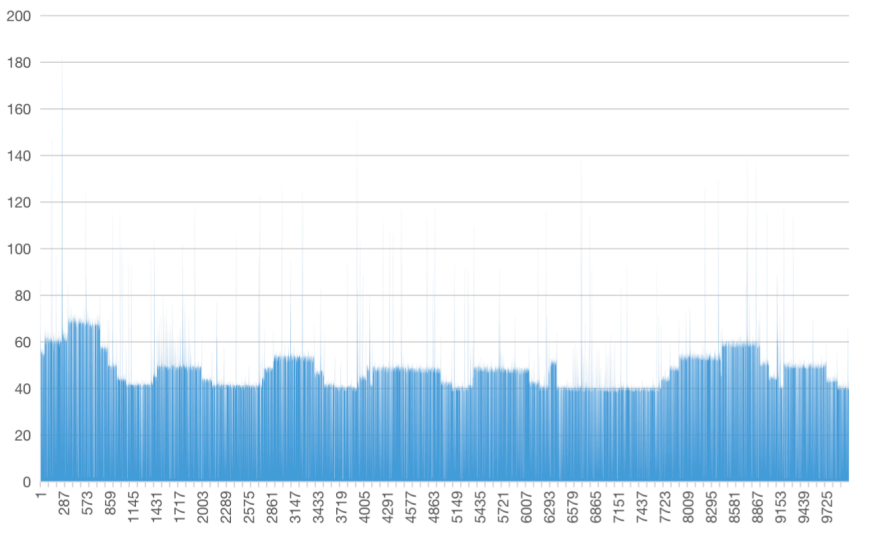

图 6: 没有缓存

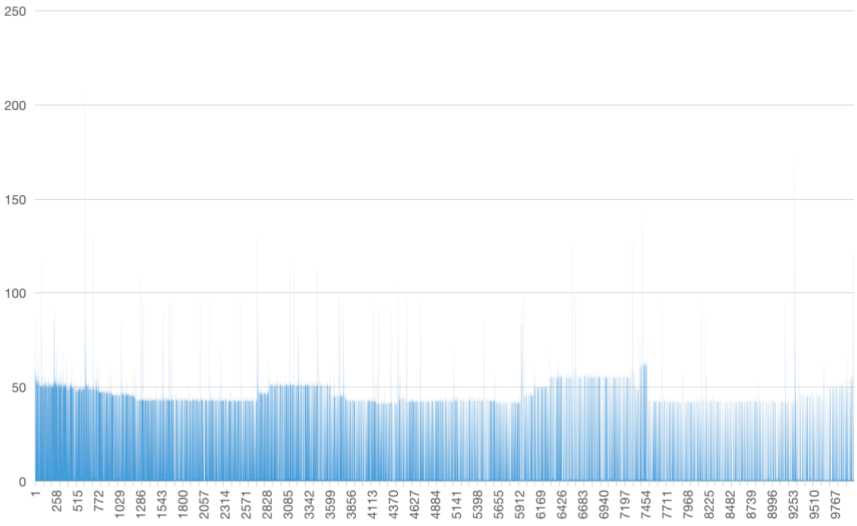

图 7: 部分缓存

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。

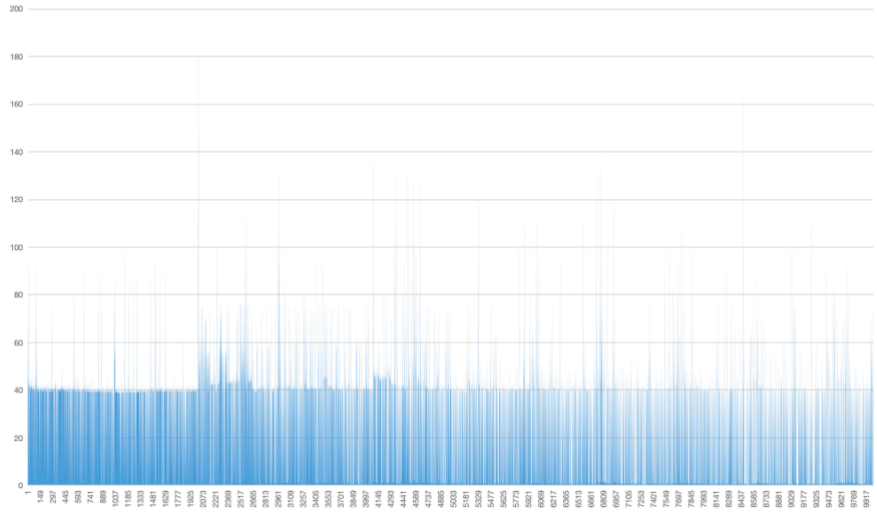

图 8: 全部缓存

从图中可以看出，全部缓存的效率 > 部分缓存 > 无缓存。

数据大小	1000	10000	20000
无缓存	0.5643ms	0.8767ms	1.4543ms
部分缓存	0.0354ms	0.0397ms	0.0964ms
有缓存	0.0235ms	0.0346ms	0.0896ms

图 9: 缓存效果测试

3.1.4 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，测试需要表现出 compaction 对吞吐量的影响。插入 10240 个 key=1,value 等于 std::string(1024,'s') 的数据，测试的如图。从图中可以看出，合并的次数越多，层数约大，最终的时延迟约多。出现了两个峰，应该是在 put 过程中生成了 sstable，合并过程中没有 put 操作。

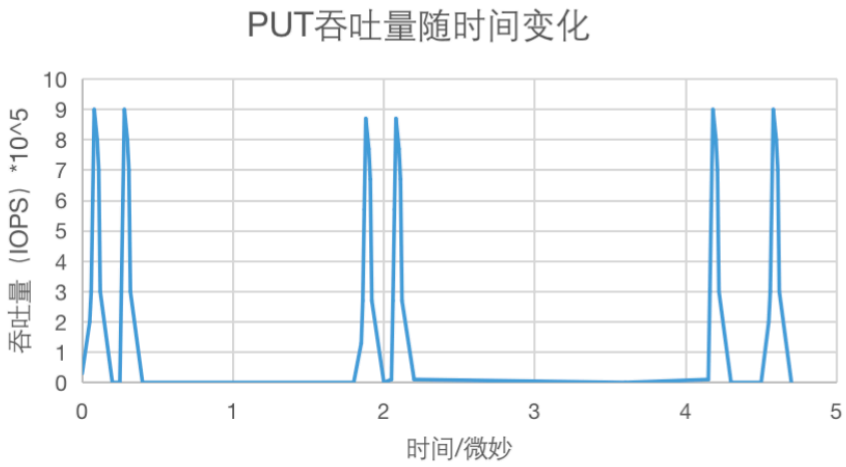

图 10: GET 操作吞吐量随时间变化

4 结论

通过对比分析，可以看到缓存索引，bloom 过滤器对于速度提升有重要作用。了解了 LSM 树这种可以高性能执行大量写操作的数据结构，同时对于局部性有了一定的认识。将对数据的修改增量保持在内存中，达到指定的大小限制后将这些修改操作批量写入磁盘。这种 cache 的思想也值得借鉴。之后在代码时，需要注意如何提升性能。

4.0.1 两个测试的结果

```
8
9   class CorrectnessTest : public Test {
10  private:
11      const uint64_t SIMPLE_TEST_MAX = 512;
12      const uint64_t LARGE_TEST_MAX = 1024 * 64;
13
14      void regular_test(uint64_t max)
```

```
Usage: /Users/kangyixiao/Library/Developer/Xcode/DerivedDa
  -v: print extra info for failed tests [currently OFF]

KVStore Correctness Test
[Simple Test]
  Phase 1: 5/5 [PASS]
  Phase 2: 512/512 [PASS]
  Phase 3: 512/512 [PASS]
  Phase 4: 1279/1279 [PASS]
4/4 passed.
[Large Test]
  Phase 1: 5/5 [PASS]
  Phase 2: 65536/65536 [PASS]
  Phase 3: 65536/65536 [PASS]
  Phase 4: 163839/163839 [PASS]
4/4 passed.
Program ended with exit code: 0
```

图 11: CorrectnessTest

```
7
8   class PersistenceTest : public Test {
9   private:
10      const uint64_t TEST_MAX = 1024 * 32;
11      //const uint64_t TEST_MAX = 1024 * 32; 原始
```

```
Usage: /Users/kangyixiao/Library/Developer/Xcode/DerivedData/LSMLAB-d
  -t: test mode for persistence test, if -t is not given, the program
  -v: print extra info for failed tests [currently OFF]

 NOTE: A normal usage is as follows:
    1. invoke `/Users/kangyixiao/Library/Developer/Xcode/DerivedData/
    2. terminate (kill) the program when data is ready;
    3. invoke `/Users/kangyixiao/Library/Developer/Xcode/DerivedData/
       test.

KVStore Persistence Test
<<Preparation Mode>>
  Phase 1: 32768/32768 [PASS]
  Phase 2: 32768/32768 [PASS]
  Phase 3: 49152/49152 [PASS]
3/3 passed.
Data is ready, please press ctrl-c/ctrl-d to terminate this program!
```

图 12: PersistenceTest