设计文档

前言

这次的 proj 是我有史以来做的最提心吊胆的一次，不仅因为刚开学时的 icslab 和 web 迭代，也因为这学期在家上课使得我有所懈怠，直到原先 ddl 4.11 前一周的周日才开始写，最终用了四天时间写完整个部分，又用了四天时间 debug（期间由于其他课程的作业耽误了两到三天，导致 proj 写与 debug 不连贯，带来不少麻烦），才在 ddl 前一天完成。

设计部分

LSM-tree 的设计思路跟文档中给出的一致，数据刚开始写入时保存在用跳表实现的 memtable 中，当 memtable 转换成 sstable 的大小达到 2M 后写入磁盘的 sstable 中。磁盘分为多层，每层能够存储的 sstable 的最大容量为等比数列递增，当某层中 sstable 超过该层最大容量时，进行合并（compaction）。LSM-tree 主要有四个功能函数：put、get、del、reset，分别是存入数据，查询数据，删除数据和重置 LSM-tree。下面重点介绍我设计的数据的结构和功能的实现。

memtable（skiplist）：声明在 SkipList.h 中的一个类，除了一般跳表中包含的成员变量之外，他还包含了 numOfNode，sizeOfKVInSs，mem_searchtable 三个变量，分别用来储存跳表中已有的数据的数量、跳表转换为 sstable 后的大小、以及将跳表中的数据写成含有 key、value、offset 的列表暂时储存，便于转化为索引区和向文件中写入

sstable：声明在 sstable.h 中的一个类，有 filename、id、searchtable 三个成员变量，分别记录 sstable 文件名、文件名的数字部分（文件以 0.txt、1.txt 等依次命名）以及文件的索引区（只含 key 和 offset）

tablenode.h：一个声明了 searchtablenode 和 mem_searchtablenode 的头文件，顾名思义，前者是 memtable 中 mem_searchtable 的节点，一个节点包含三个变量，后者是 sstable 中 searchtable 的节点，一个节点只包含两个变量（无 value）

kvstore.h：五个成员变量 dir、memtable、disk、sort_table、sort_disk。dir 记录 sstable 储存的路径名，此 porj 中为./data。memtable 为 skiplist 类的指针。disk 为 sstable 类的二维 vector，我认为这样可以很好的表明文件储存时的层次结构。sort_table 在合并时存入所有需要合并的 mem_searchtablenode，便于排序、输出到 sstable、控制 key 的单一性等。sort_disk 在合并时存入所有需要合并的 sstable 类对象，然后将其转化为一个个 mem_searchtablenode 读入 sort_table 中。

值得一提的是，我在思考后认为每个 kv-pair 的时间戳没有太大必要，甚至连每个 sstable 都不需要加上时间戳，只需保证旧的同 key kv-pair 永远不会先于新 pair 被读到即可。因此我并没有加入时间戳，具体控制数据新旧与单一性的操作在接下来会详细讲述。

put 操作：首先判断 memtable 中是否已经存在含有这个 key 的 kv-pair，若有直接删除。随后插入 kv-pair，同时判断是否达到写入 sstable 的临界。若达到临界，写入 sstable 并存入 L0 层，同时遍历 0 至 disk.size()-1 层，是否有文件数目溢出。若有，确认溢出层 i 中需要合并的文件数量，从 disk[0]开始确认文件（因为后写入的文件新，所以先合并排在前面的旧文件），根据 i 层文件中 key 范围确定 i+1 层中需要合并的文件，然后先将 i+1 层中文件 push_back 进 sort_disk，再将 i 中文件 push_back 进 sort_disk，这样确保下面操作中删除的同 key 的 kv-pair 是旧的。随后，从后往前依次读出 sort_disk 中的 sstable，读出后根据索引区 searchtable 将文件中的 kv-pair 读取出来，在将其压进 sort_table 中前，先判断 sort_table 中是否含有同 key 的 pair，若有则不压入。当 sort_disk 中所有 sstable 的

kv-pair 均已入列后，使用 list 自带的 sort 函数进行排序（我万万没想到要使用的是多个 list 的归并排序，我只用了一个 list 自身的归并），然后依次出列，每达到 2M 便生成一个新的 sstable，从 disk[i]层 push_back，调用 exporting 函数导出到 disk[i+1]层（由于我每次取文件都是使用 disk[i][0]，disk[i][1]，disk[i][disk[i].size()-1]等的方式，所以其实文件名我并不关心，但是我又需要生成不一样的文件名，因此我选择记录每个 sstable 的 id，然后用 id 命名，这样会导致运行多次后的 id 达到一个很大的数，但还好都没有达到 LONG_MAX。我认为我肯定可以找出与贪吃蛇 proj 中蛇身周期性变色类似的命名方法，可惜时间不足，以后的 proj 我一定会尽早开始……）

get 操作：先检查 memtable 中是否含有该 key，若有则直接取出，否则从 L0 层开始，从后(disk[i][disk[i].size()-1])往前(disk[i][0])遍历 sstable 的索引区，找到相同 key 后不再继续遍历(后面即使有也是旧的)

del 操作：先检查 memtable 中是否含有该 key，若有先取出 value——value 为“”则代表已被删除，直接返回 false，否则删除 kv-pair 并返回 true。若 memtable 中不存在，则与 get 操作类似遍历 sstable，若无此 key 或 value 为“”则代表已删除，返回 false，若 value 不为空则 put(key,“”)。

reset 操作：调用每次写入 sstable 后都会调用的 memtable->reset()重置跳表与 memtable 自带的参数，同时清空 sort_table、sort_disk，并 remove_all(dir)以及 create(dir/L0)

KVStore 构造函数：先检查 dir/L0 是否存在，若存在则代表磁盘中已有文件，于是从 L0 开始依次遍历每层下的所有文件（使用 struct _finddata_t）并依靠 filename 构造每个 sstable 对象。若 dir/L0 不存在，则新建 dir/L0 目录并结束构造

测试部分

开发环境：

Microsoft VS Code 1.44.2.0

测试环境：

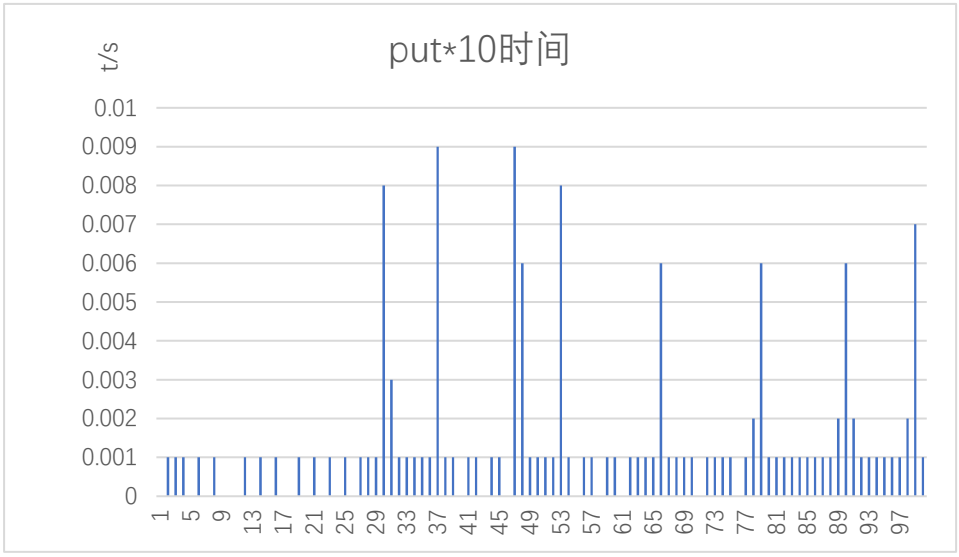
机型：Surface Pro 5
处理器：Intel Core i7-7660U
内存：16.0 GB
硬盘：512GB SSD
系统：Windows 10

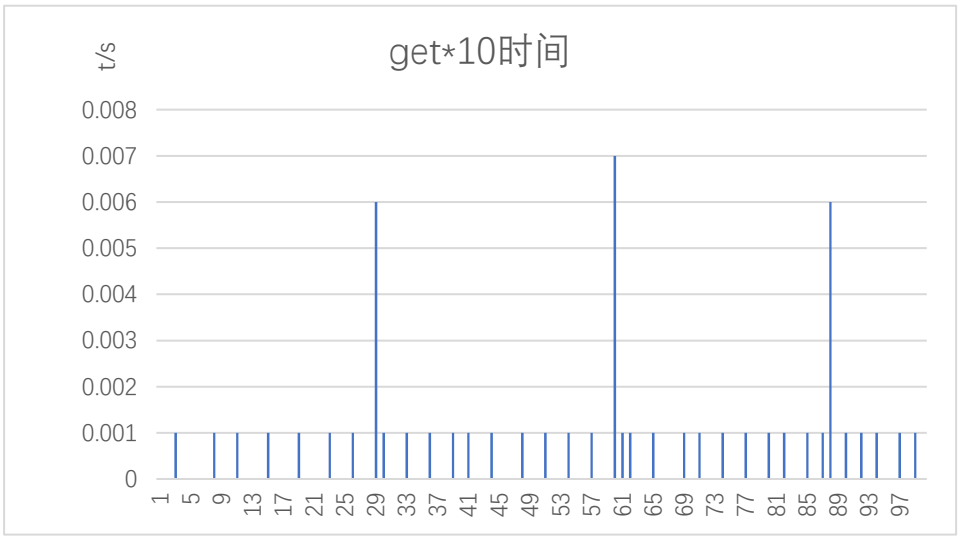
时延：在正常情况下（无 compaction 操作）PUT、GET、DELETE 操作所需要的平均时间的柱状图；

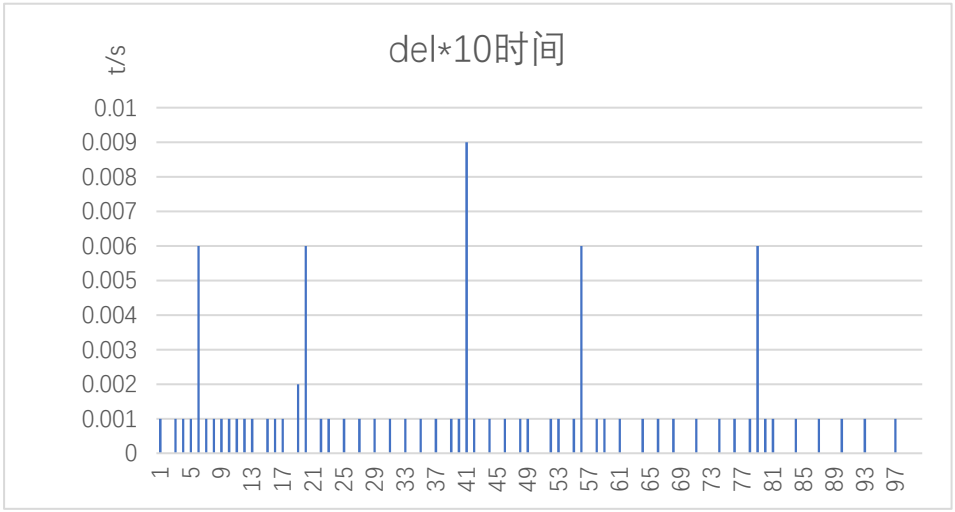
测试程序：

```
const uint64_t NORMAL_TEST_MAX = 1000;

void regular_test(uint64_t max)
{
    store.reset();

    // Test multiple key-value pairs
    clock_t start1=clock();
    for (uint64_t i = 0; i < max; ++i) {
        store.put(i, std::string(i+1, 's'));
        if(i%10==9)
            cout << (double)(clock() - start1) / CLOCKS_PER_SEC << endl ;
    }
    cout<<endl;
    phase();

    // Test after all insertions
    clock_t start2=clock();
    for (uint64_t i = 0; i < max; ++i)
    {
        EXPECT(std::string(i+1, 's'), store.get(i));
        if(i%10==9)
            cout << (double)(clock() - start2) / CLOCKS_PER_SEC << endl ;
    }
    cout<<endl;
    phase();

    // Test deletions
    clock_t start3=clock();
    for (uint64_t i = 0; i < max; ++i)
    {
        EXPECT(true, store.del(i));
        if(i%10==9)
            cout << (double)(clock() - start3) / CLOCKS_PER_SEC << endl ;
    }
    cout<<endl;
    phase();
    report();
}
```

条形图：







由于每次操作时间过短，因此我选择 10 次操作计算一次时间，可以看出三种操作的平均时间都在 0.001 秒左右。put 时时间过长可能是因为某些塔的层数较高，而 get 时间过长则可能是某些塔时间过低。可以看到这两者的较长时间不重合。由于 del 时是遍历整个

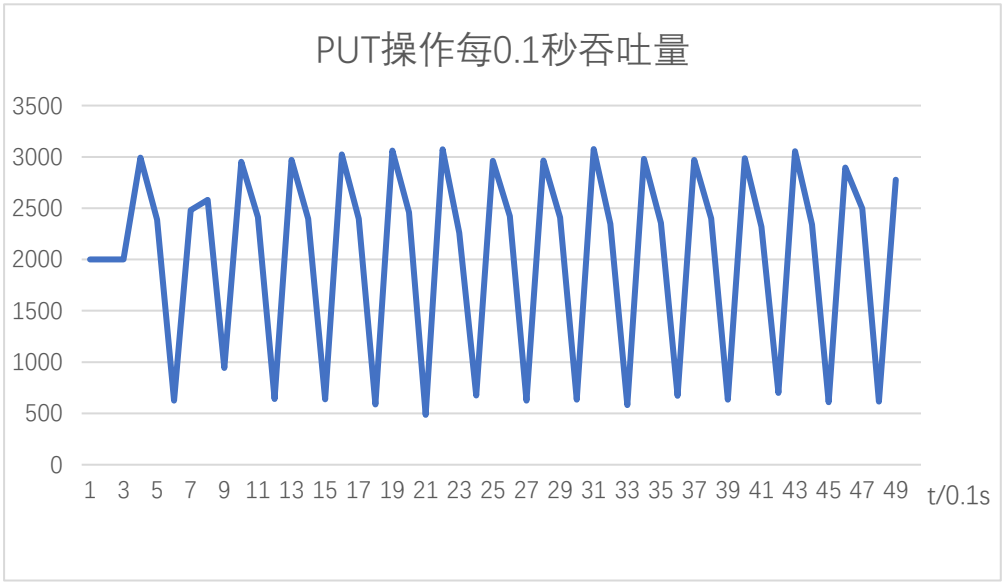
跳表，所以 del 出现过长的时间我也有点不能理解，可能是不同时间 cpu 状态不同吧。但越到后面时间越总体下降是符合规律的，因为节点数变少了。

2）吞吐量：不断插入数据的情况下，每秒钟处理的 PUT 请求个数（即吞吐量）随时间变化的折线图。

测试程序:

```
const uint64_t LARGE_TEST_MAX = 1000*100;
void large_test(uint64_t max)
{
    store.reset();

    // Test multiple key-value pairs
    clock_t start1=clock();
    int num1=0;
    for (uint64_t i = 0; i < max; ++i) {
        ++num1;
        store.put(i, std::string(1024, 's'));
        if((double)(clock() - start1) / CLOCKS_PER_SEC >= 0.1)
        {
            cout << num1 << " " ;
            num1=0;
            start1=clock();
        }
    }
    cout<<endl;
    phase();
    report();
}
```

折线图:



由于我刻意安排每次插入的 value 的长度都是 1024，这就导致发生 compaction 是有一定周期性的，可以看出图像较符合此规律。当吞吐量最大（3000 左右）时是不发生写入的时候，当吞吐量较大（2500 左右）时是发生写入 sstable 但不发生 compaction 时，当吞吐量最小（低于 1000）时是发生 compaction 时。

总结

总结几个浪费了不少时间的 bug：

① 合并时检查下层 sstable 中 key 的范围与上层是否有交集，忽略了上层整个的范围是下层文件范围的真子集的情况

② string 写入文件、从文件读出到 string 和 uint_64t 的方法有误，多亏了谭世炜同学的讲解和帮助，我才能正确读入和写入文件

③ del 操作的时候先检查 memtable 中是否有 key，检查过之后我不论 value 是否为空都直接删去并返回 true 了，实际上若 value 为空则不应删掉并返回 false

④ 合并时将 kv-pair 写入 sort_table 时要先判断 sort_table 中是否含有该 key，应该先判断 if(contain(it->key))再让迭代器 it++，我不知道怎么的先让 it++再判断，由于我 debug 的时候与写的时候相隔了近一个星期，导致我一个一个翻 sstable 文件发现一些 key 压根没有，才得以找出这个 bug...

⑤ sstable 类我手贱加了个析构函数，会 remove 掉相应文件，导致我写完一个文件就被删，写完一个就被删，我一度以为闹鬼了，后来单步调试时才意识到是析构函数（这又是因为我 debug 的时候早就忘了写的时候心里是怎么想的）

⑥ sort_disk 与 sort_table 俩名字太像，导致我合并时稍微不注意 clear 错了，我一度以为 list 容器限定大小，溢出后会全部清除，搞得我怀疑人生...

⑦ 其实本来这些都是小 bug，应该很容易看出，但是由于刚开始我 debug 时打断点无效（程序忽略断点继续向下执行），所以只能采取原始的 cout 法以及很麻烦的 gdb，心态崩了.....后来不知道什么原因又可以打断点了（好像是因为我点了一下启用所有断点？），于是我才可以欢乐地 debug

⑧ correctness 通过之后我自认为 KVStore 的构造函数写的天衣无缝，persistence 肯定一遍过，结果由于①和④，prepare 时就 fail 了，我一度逃避艰难的人生，开始追番.....后来因为 ddl 的催赶，我不得不一个一个翻 sstable 文件并且重新理逻辑，才终于找出 bug

特别鸣谢：斯金泽。斯金泽同学在比我早做完两到三天的情况下向我施以援手，我得以和他深入地讨论具体结构与细节，这对于我对整个项目的理解的加深有很大帮助。另外他还指正过许多我忽略的 bug：比如上述的①和③。可以说如果没有他的帮助，我可能每天要晚睡 3 个小时。