LSM-Tree

2020 年 4 月 19 日

设计

文件结构

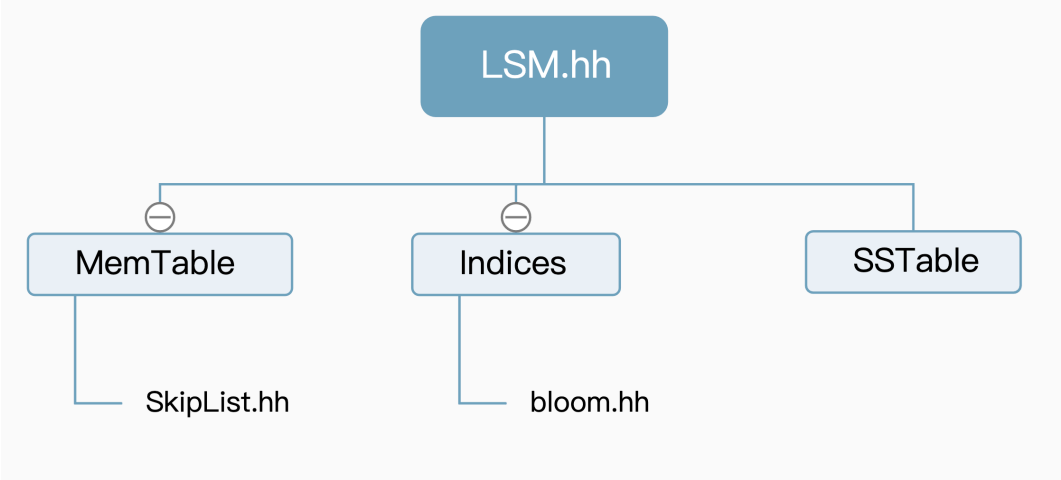

图 1: File Structure

上图为 LSM-Tree 的功能模块以及依赖的文件。LSM 为接口类，负责管理内存中的 SkipList(键值对的内存存储) 以及 Indices(SSTable 在内存中的缓存)，以及实现了对磁盘中 SStable 的操作 (如 dump,read,compact 等)。

MemTable

内存中的存储由跳表实现。我实现了字节统计和导出数据，便于 LSM 调用。除此之外与上课讲的大致相同，在此便不再赘述。

SSTable

图 2: SSTable Structure

上图为 SSTable 的结构。首先是 32 位无符号整数 size(冗余, uint32_t), 接着是存储了数据段 (dataSeg, uint32_t) 关于文件开头的字节偏移量的 dataSegBias，再接着是存储了每个数据的键值 (key, uint64_t)、关于数据段开头的偏置 (dataBias, uint32_t) 以及数据的字节长度 (dataLen, uint32_t)，最后是存储了所有值的数据段 dataSeg(sum(dataLen))。

SSTable 我将其主要功能设计为实现文件二进制数据 (Struct Bin) 与内存中键值对 (vector<Entry<uint64_t, string> >) 的转换由两个构造函数实现，另外还实现了一些接口用来方便调用。(以下代码仅为示意)

```
template<K, V>
class SST {
    explicit SST(const vector<Entry<K, V> > &_data, uint32_t _dataBytes);
    explicit SST(char *_bin)
    explicit SST(const SST<K, V> & ano);
    ~SST();
    Bin toBin();
    Bin toIndexBin();
    vector<Entry<K, V> > &vecData();
    uint32_t getSize() const;
    uint32_t getDataSegBias() const;
    K getLowBound() const;
    K getHighBound() const;
};
```

Indices

内存中的索引我设计为以查找以及根据二进制文件自动初始化为主要功能，由 Indices(单个 SSTable 的索引) 和 IndicesTab(Indices 的集合，包含在 Disk 结构中的层次信息)。

在硬盘中,我使用文件名来区分文件所处的层次结构 (levelinLevel.bin), 如 ("00.bin", "315.bin")。我设定第 0 层的 MAX_SIZE 为 4，其余层数为 $2^n \times 4$。

基于以下逻辑，该命名法还可以用来保证 lazy 方式下不使用时间戳时操作的正确性。在第 0 层中，inLevel 越大越新 ($03 > 02 > 01 > 00$)；在其他层中，因为同层键值对不会相交，故层内不必区分时间顺序，而又因为归并是向下的，上层一定新于下层。故归纳可得

$$MemTable > 03.bin > 02.bin > 01.bin > 00.bin > level1 > level2 > ...'$$

```cpp
template<K>
class IndicesTab {
    explicit IndicesTab(const string &_dir);
    bool insert(const Indices<K> &idx, string &filename);
    vector<Indices<K>> *rLevel(uint32_t levelNum);
    uint32_t getHeight() const;
    bool find(const K &key,
              string *filename = nullptr, uint32_t *dataSegBias= nullptr,
              uint32_t *bias = nullptr, uint32_t *length = nullptr);
    void addNewLevel();
    void clear();
}

template<K>
class Indices {
    explicit Indices(const Bin &bin, uint32_t _size, uint32_t _dataSegBias);
    bool find(const K &k, uint32_t *b = nullptr, uint32_t *l = nullptr);
    uint32_t getSize() const;
    uint32_t getDataSegBias() const;
    uint32_t getLowBound() const;
    uint32_t getHighBound() const;
}
```

Bonus

除了基本功能之外，我还使用了布隆过滤器用来快速判断某个 key 是否有可能在索引中，且 LSM-Tree 是以模板的方式实现的，能够存储包括 <uint64_t, string> 在内的其他键值对 (因为模板的实现与声明不能分开，故我的所有实现都在.hh 文件中)。

测试

测试环境

机器: MacBook Pro (13-inch, 2018, Four Thunderbolt 3 Ports)

系统: macOS Catalina 10.15.3

硬盘: APPLE SSD AP0512M

CPU: Intel i5-8259U

Correctness Test

对一定范围内的 key，我使用长度为 1-100 的随机字符串进行赋值，并同时随机地对 LSM 和 SkipList 进行操作，最后进行比对测试两者结果是否相同。测试结果：

$$CorrectnessTestResult : 262144/262144 => 100\%$$

Latency Test

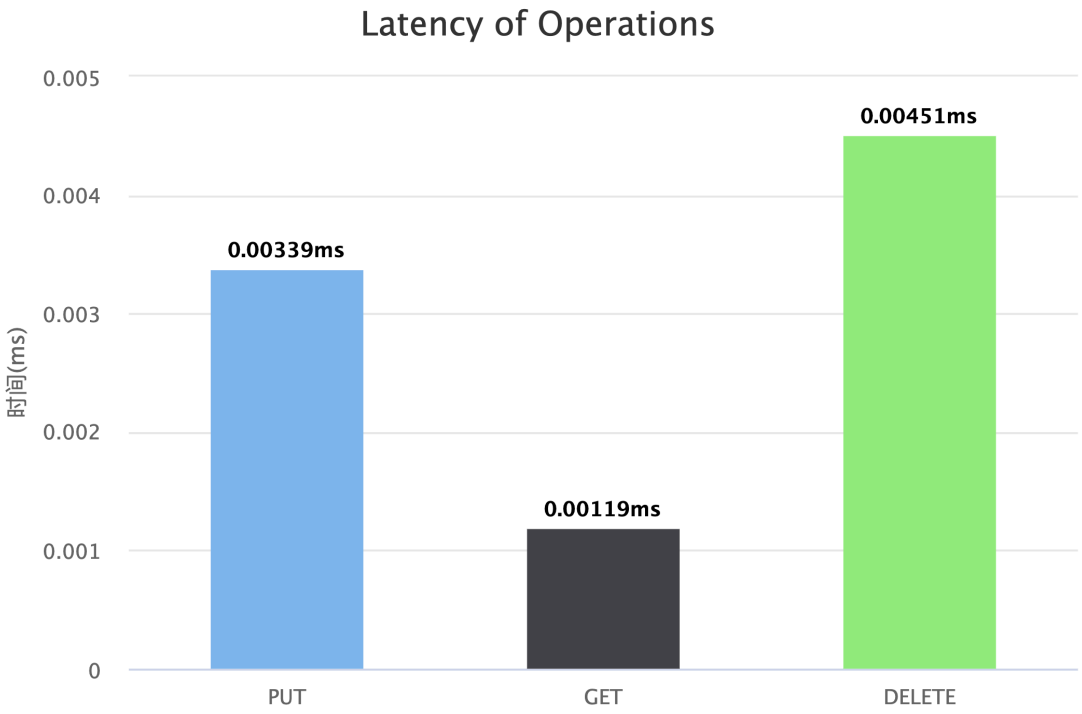

图 3: Latency Test

上图为非 Compact 情况下的 PUT, Get, Delete 延迟测试结果。可见 $DELETE > PUT > GET$ 且有 $DELETE \approx GET + PUT$。原因是在 DELETE 时先要进行与 GET 类似的查询操作，而我将 DELETE 的 lazy 操作用等价的 put(key, "") 替代了，故 DELETE 操作需经过约 PUT 和 GET 加和的时间。

Throughput Test

可能是因为固态硬盘随机读取性能较好的缘故，1s 内难以体现出 Compaction 对吞吐的影响，于是我把时间设定为 0.2ms 进行测试，结果如下。

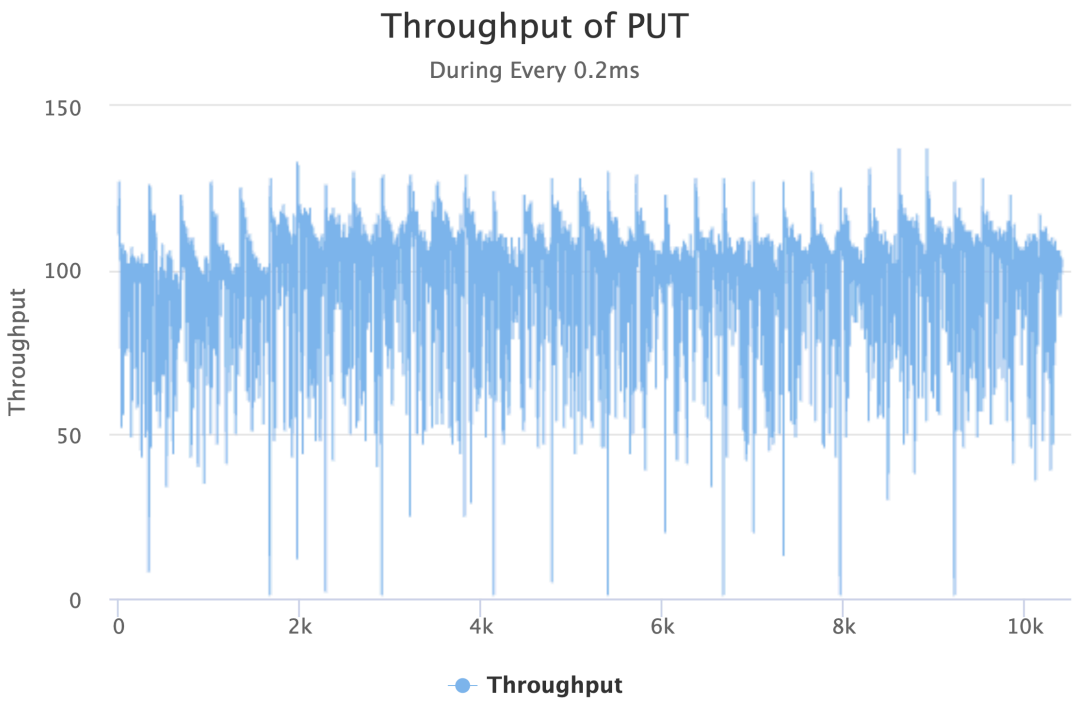

图 4: Throughput Test

上图可见随着 put 的元素越来越多，LSM 的吞吐量呈现周期性的波动，但总体稳定在 $100/0.2ms$ 左右

Code

```
#define TEST_SIZE (1 << 20)

char *randstr(char *str, const int len) {
    int i;
    for (i = 0; i < len; ++i) {
        switch ((rand() % 3)) {
        case 1:
            str[i] = 'A' + rand() % 26;
            break;
        case 2:
            str[i] = 'a' + rand() % 26;
            break;
        default:
```

```
                str[i] = '0' + rand() % 10;
                break;
            }
        }
        return str;
    }

    void doSomething(LSM<uint64_t, string> &lsm, uint64_t key, SkipList<uint64_t, string> *memTab = nullptr) {
        switch(rand() % 2) {
            case 0: {
                char ranStr[100]; int len = rand() % 99;
                randstr(ranStr, len);
                lsm.put(key, string(ranStr, len));
                if (memTab) { memTab->put(key, string(ranStr, len)); }
                break;
            }
            case 1: {
                lsm.remove(key);
                if (memTab) { memTab->remove(key); }
                break;
            }
        }
    }

    void correctnessTest(LSM<uint64_t, string> &lsm, uint64_t size) {
        lsm.reset();
        SkipList<uint64_t, string> memTab;
        for (uint64_t i = 0; i < size; ++i) {
            doSomething(lsm, i, &memTab);
        }
        for (uint64_t i = 0; i < size; ++i) {
            doSomething(lsm, i, &memTab);
        }

        uint64_t cnt = 0;
        auto data = memTab.data();
        for (uint64_t i = 0; i < size; ++i) {
            string lsmGet = lsm.get(i), *memGet = memTab.get(i);
            if (memGet) {
                if (lsmGet != *memGet) {
                    cout << "LSM get: "<< lsmGet << endl << "MenTable get: " << *memGet << endl;
                }
                else { ++cnt;}
            }
            else {
                if (!lsmGet.empty()) { cout << "LSM get: "<< lsmGet << endl << "MenTable get: " << endl; }
                else { ++cnt; }
            }
        }
        cout << "Correctness Test Result: " << cnt << '/' << size << " => " << double(cnt) / size * 100 << '%' << endl;
    }

    struct Lat {
        double putLat;
        double getLat;
        double delLat;
        uint64_t size;
    };


    void latencyTest(LSM<uint64_t, string> &lsm, uint64_t size) {
        lsm.reset();
        vector<Lat> latRecd;
        clock_t start;
        for (uint64_t i = 0; i < size; ++i) {
            Lat lat;
            char ranStr[100]; int len = rand() % 100;
            randstr(ranStr, len);
            string input = string(ranStr, len);
            start = clock(); bool flag = lsm.put(i, input); lat.putLat = double(clock() - start) / CLOCKS_PER_SEC * 1000;
            if (flag) {
                start = clock(); lsm.get(i); lat.getLat = double(clock() - start) / CLOCKS_PER_SEC
```

```
    * 1000;
                start = clock(); lsm.remove(i); lat.delLat = double(clock() - start) /
CLOCKS_PER_SEC * 1000;
                latRecd.push_back(lat);
            }
        }

        double avePut = 0, aveGet = 0, aveDel = 0;
        for (auto i = latRecd.begin(); i != latRecd.end(); ++i) {
            avePut += i->putLat;
            aveGet += i->getLat;
            aveDel += i->delLat;
        }

        avePut /= latRecd.size(); aveGet /= latRecd.size(); aveDel /= latRecd.size();

        cout << "Latency Test Result: " << "Put: " << avePut << "ms " << "Get: " << aveGet << "ms
" << "Delete: " << aveDel << "ms" << endl;
    }

    void throughputTest(LSM<uint64_t, string> &lsm, uint64_t size) {
        lsm.reset();
        uint64_t i = 0;
        vector<int> t;
        char data[100][100];
        for (int i = 0; i < 100; ++i) { randstr(data[i], 100); }

        clock_t start = clock(); uint64_t cnt = 0;
        while (i < size) {
            if (clock() - start > 200) {
                t.push_back(cnt);
                cnt = 0;
                start = clock();
            }
            lsm.put(i, string(data[i % 100], rand() % 100)); ++cnt; ++i;
        }
        cout << "Throughput Test Result:" << endl;
        cout << '[';
        uint64_t c = 0;
        for (auto &i:t) {
            cout << '[' << c++ << ',' << i << ']' << ',';
        }
        cout << ']';
    }

    int main() {
        srand(time(0));
        LSM<uint64_t, string> lsm("./data");
        correctnessTest(lsm, TEST_SIZE);
        latencyTest(lsm, TEST_SIZE);
        throughputTest(lsm, TEST_SIZE);
        return 0;
    }
```