Project1: LSM Tree 键值存储系统

尹楚轩 517021910929

2020 年 4 月 11 日

1 概要

该项目基于 LSM Tree 这种可以高性能执行大量读写的数据结构，设计和开发了一套简化的键值存储系统。该存储系统分为内存存储和磁盘存储两部分，其中内存部分采用跳表数据结构进行组织，磁盘部分采用分级存储的形式。该存储系统支持简单的增删查改功能，当内存部分的存储量超过阈值 (2MB) 时就会将数据写入磁盘。该项目在 windows 平台下的 visual studio 2017 上开发，但没有任何的平台依赖性。

2 项目背景

LSM 算法被设计来提供比传统的 B+ 树或者 ISAM 更好的写操作吞吐量，通过消去随机的本地更新操作来达到这个目标。

磁盘读写具有显著的随机操作慢，顺序读写快的特点，因此在设计存储结构时，应该尽量避免随机读写，最好设计成顺序读写。如果对写操作的吞吐量敏感，一个好的策略是简单地将数据添加到文件，这个策略经常被使用在日志或者堆文件，因为它们是完全顺序的，可以提供非常好的写操作性能。但它们只适用于简单的读场景，对于更复杂的读场景 (按 key 读取)，需要更复杂的算法来支持。

这就是 LSM 算法被设计的初衷，LSM 保持了日志文件的写性能，以及微小的读操作的性能损失，本质上是让所有操作顺序化，而不是随机读写。从概念上来说，LSM 将一个很大的查找结构变换为将写操作顺序的保存到一些相似的有序文件中。因为文件是有序的，所以之后的查找也会很快。文件时不可修改的，它们永远不会被更新，新的更新只会写到新的文件中，通过周期性的合并来减少文件个数。

从应用场景来考虑，随着可用内存的增加，通过操作系统提供的大文件缓存，读操作自然会被优化，写性能就变为了主要的关注点，选择一个写优化的文件存储结构变得愈发有意义。现代大型互联网企业对写性能的提高越来越敏感，工作场景从 read-heavy 转向 read-write，本项目所设计的键值存储系统也就有了一定的实际意义。

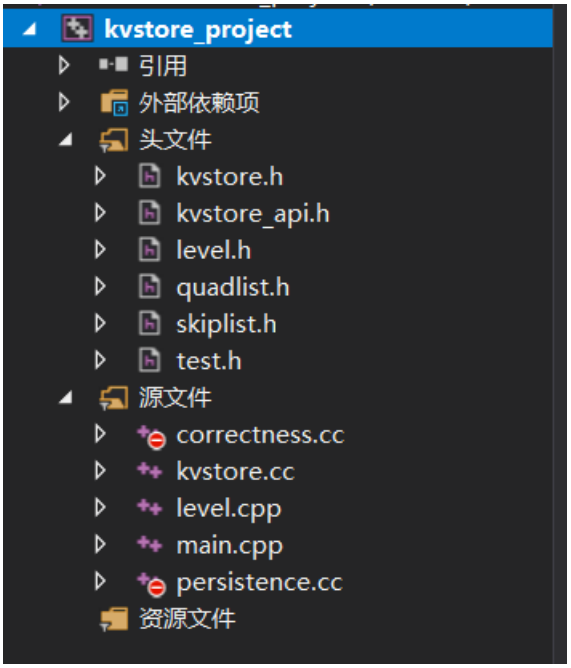
3 项目目标

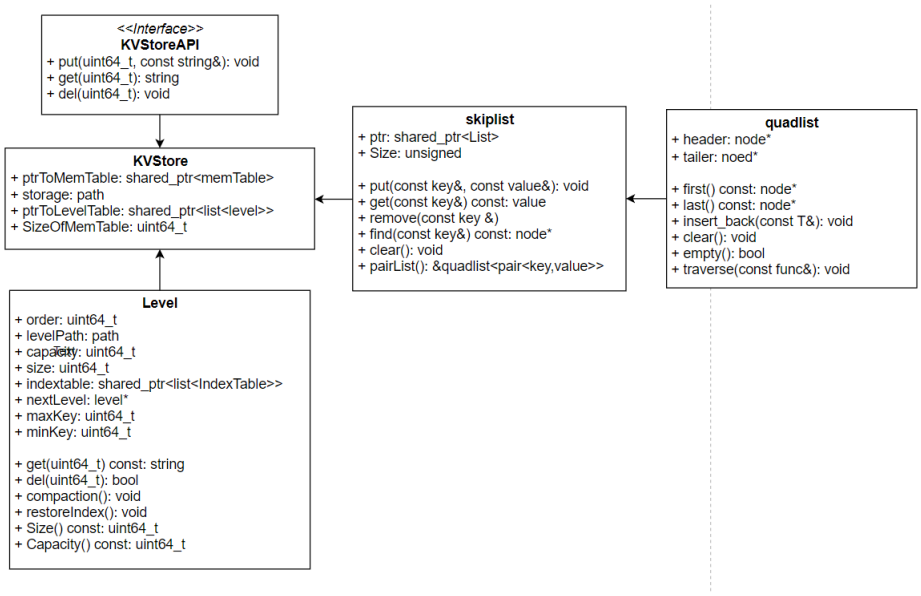
基于 LSM 算法开发一套相对完整的键值存储系统，实现 put，get，delete 功能，通过大量数据测试和可持久化测试，并在此基础上优化读写性能。

4 设计思路和解决方案

4.1 整体结构设计

整个项目由 c++ 实现，由虚基类 KVStoreAPI 规定存储系统的接口，具体实现由继承类 KVStore 完成。skiplist 模板和 quadlist 模板实现跳表数据结构，后者用来组织 KVStore 中的内存部分。level 类用于管理外存中各层的读写，合并等操作。test.cc，persistence.cc，main.cpp 用于测试。





4.2 KVStore 类的设计

由于 KVStore 内存存储和磁盘存储两部分构成，分别用 ptrToMemTable 和 ptrToLevelTable 两个私有指针指向内存存储结构和外存存储的 level 表。用私有变量 storage 记录初始化时的文件目录参数 dir，用私有变量 SizeOfMemTable 来记录当前内存存储量大小，以便于检查内存存储是否溢出。

KVStore 的公共接口主要有构造/析构函数，put()，get()，del() 函数。构造函数初始化所有私有变量，并在 dir 目录下创建 level 子目录 (level0 - level9)，每个 level 目录中包含一个专门用来存放 index 的 index 目录，并在 ptrToLevelTable 所指向的 level 表中添加关于各个 level 的信息。put()，get()，del() 实现思路相同，都是先对 memtable 进行操作，若操作失败 (找不到元素或溢出)，则转向外部存储操作。

此外,transfer 私有函数实现当 memtable 溢出时 memtable 写入 SSTable 的转换。transfer() 对 level0 调用 addSSTable()，将内存中的 memtable 清空，并将 SizeOfMemTable 赋为 0。

4.3 skiplist 模板的设计

skiplist(跳表) 是用来实现内存存储中 memtable 的数据结构，其本身又由多层四元表组成。四元表是能在四个方向上延申的链表，但在 skiplist 的使用中，往往只需要在一个方向上做遍

历，因此将 skiplist 中的四元表考虑为线性结构，在头部和尾部有哨兵标识，而每个单元多余的指针则可以让 skiplist 在每一层链表上增加一层，并非常方便地实现 skiplist 的层间跳跃。

skiplist 提供了由关键字而进行 put()，get()，find()，delete() 的公共接口。

4.4 level 类的设计

4.4.1 私有变量

level 类是实现外部存储的核心，由 ptrToLevelTable 所指向的 level 表管理着整个外部存储模块。level 类中的私有变量有: 标识 level 序号的 order，每个 level 的文件路径 levelPath，每个 level 的最大容量 capacity，level 中当前 SSTable 数量 size，指向 level 中所有 SSTable 所对应的 index 表的指针 indextable，指向做 compaction 时的下一个 level 的指针 nextLevel，level 中所有记录的 key 的最大值 maxKey 和最小值 minKey。

4.4.2 index 结构体

index 结构体定义如下:

```
struct index{
    uint64_t key, offset, size, order, level;

    clock_t timeStamp;

    bool flag;

    index(){}

    index(uint64_t k, uint64_t o, uint64_t s, uint64_t ord, uint64_t l):
    key(k),offset(o),size(s),order(ord),level(l),timeStamp(clock()),flag(false){}

    //operator< overload for sorting
    bool operator<(const index &i){
        return key < i.key;
    }
};
```

每一个 index 用来记录 SSTable 中一个条目的信息。index 中除了有记录 key，offset 和 timeStamp 之外，还有一些其它的信息，如其所处的 level 编号和 SSTable 编号，这主要是为了方便定位该条目所处的 SSTable，flag 用于标记该条目是否被删除。operator< 运算符重载用于 index 表排序。一个 SSTable 所有条目的 index 信息由一个 vector 组织，这个 vector 和 SSTable 路径组成的 pair 被定义为 IndexTable 类型。

```
typedef std::pair<std::vector<index>, fs::path> IndexTable;
```

一个 level 中的所有 IndexTable 统一由一个指向链表的指针 indextable 管理。

4.4.3 接口设计

level 类同样提供了 put() 和 del() 的公共接口，但是将 addSSTable() 设计成了友元函数。公共接口中，compaction() 触发当前的 level 与后一个 level 的合并操作；restoreIndex 负责在系统重启时将备份在磁盘中的 index 表载入内存中，Size() 和 Capacity() 返回当前 level 的大小和容量。此外在 level 类内部还有一些私有成员函数:binarySearch() 用于查询 index 表时进行二分查找；ReadFromSSTable() 用于从 SSTable 中读取特定的键值；merge() 用于在 compaction 时融合两层的 SSTable；renaming() 用于在 compaction 之后重新对 SSTable 进行命名和记录。下面具体介绍几个接口的设计。

4.4.4 addSSTable 接口设计

addSSTable() 函数原型如下:

```
void addSSTable(const quadlist<std::pair<uint64_t, std::string>> &l, level *le);
```

addSSTable() 被设计为一个 non-member function，主要是因为在一次 compaction 中可能会有多个 level 需要执行 addSSTable，因此需要实现 level 和 addSSTable 的解耦。addSSTable() 根据当前 level 中的 size 给新增添的 SSTable 标号为 size + 1，并将其以编号命名。然后打开新建的 SSTable 文件循环写入 quadlist 中的内容，并将其 key 和 offset 等信息记录到一个临时创建的 IndexTable 中。将 vector<index> 排好序后加入到 indextable 所指向的链表中，并检查是否更新 minKey 和 maxKey。

4.4.5 compaction 接口设计

compaction 启动当前层和下一层的合并，如果当前层已经是最后一层，下一层不存在 (level9)，则抛出异常。否则通过对下一层 indextable 的遍历，与当前层 minKey 和 maxKey 比较寻找

key 被覆盖的所有 Indextable。将所选出的 IndexTable 与当前层的所有 IndexTable 合并，重新排序，并执行 merge 将所有 key 重复的条目和已删除的条目更新。随后再遍历新得到得 indextable，从中读出硬盘内容重新写入新的 SSTable。最后再将之前选中的所有 IndexTable 所对应的 SSTable 全部删除，对下一层的所有 SSTable 进行重命名，检查其 size 是否超过 capacity，如果是，则继续对下一层执行 compaction。

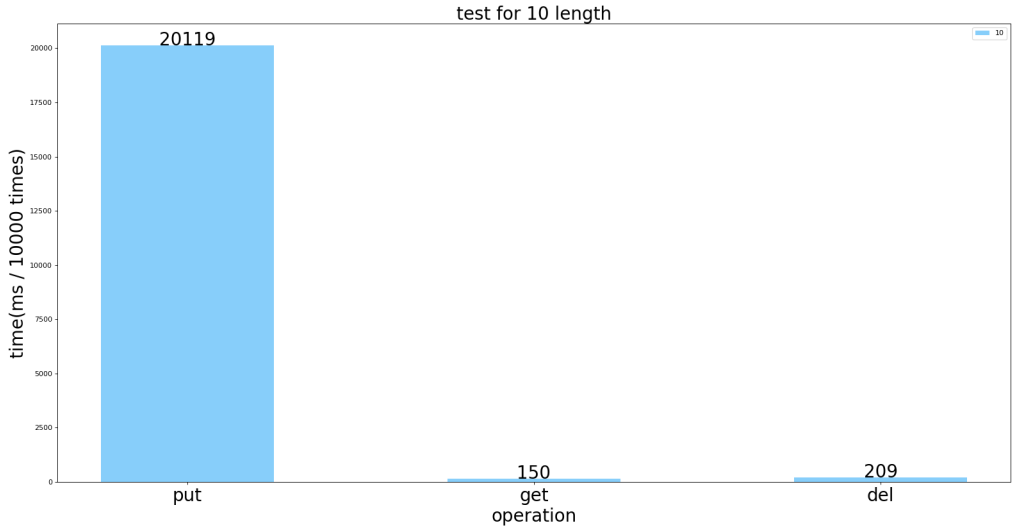
5 测试方案及结果

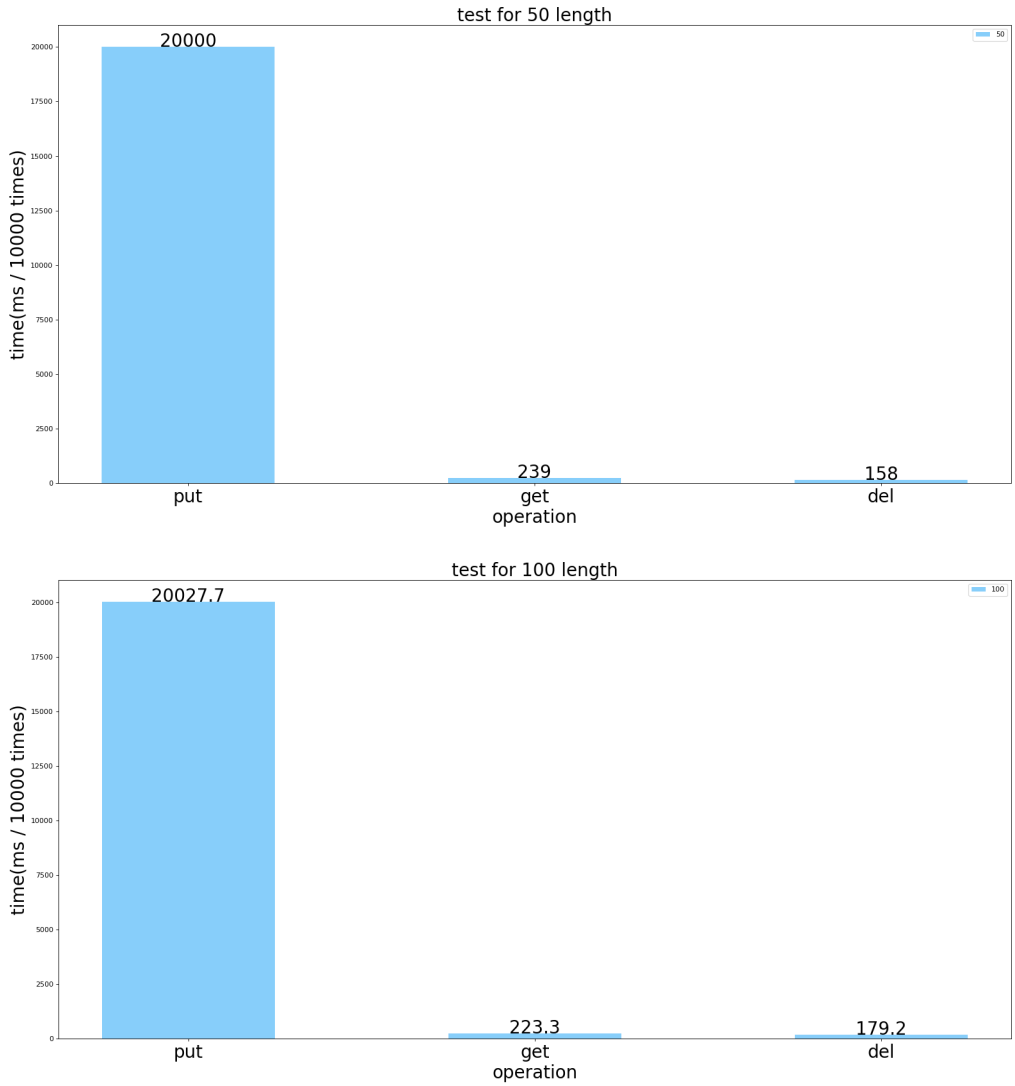
5.1 正确性测试

该项目已通过所有简单测试，复杂测试和可持久化测试。

5.2 性能测试

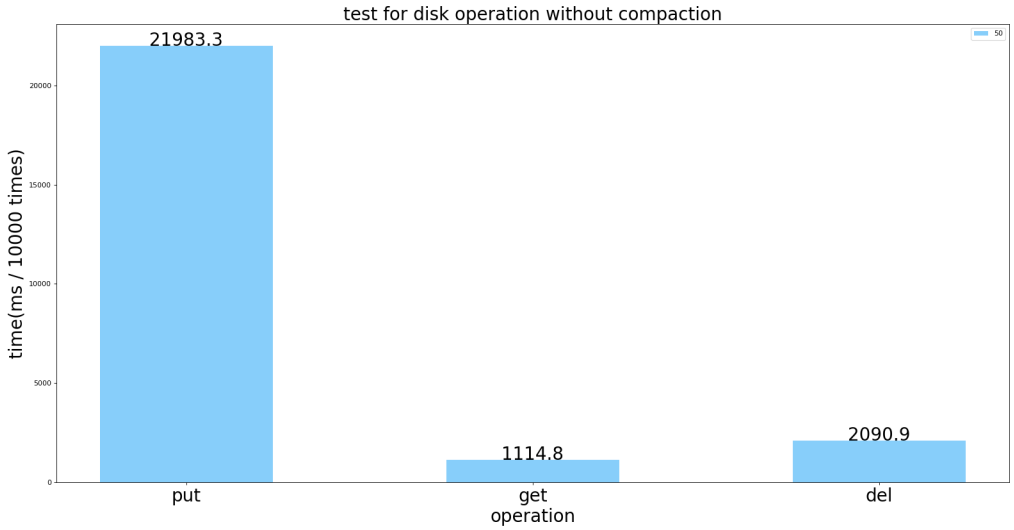
性能测试部分编写于 main.cpp 中，主要的测试策略是重复地连续插入相同的字符串，连续读取和连续删除，记录下所用时间并取平均值。
首先考察内存部分 memtable 的读写性能，以下分别是连续插入/读取/删除 10000 次长度为 10，长度为 50，长度为 100 的字符串时所需要消耗的时间:



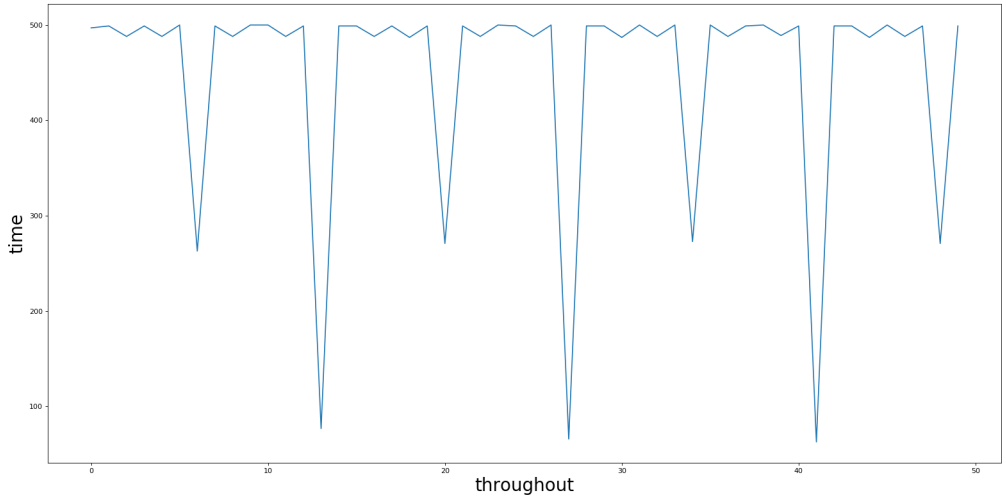


可以看出执行 put 的时间要远远长于 get 和 del，且写入字符串的长度对操作时间几乎没有影响。

接下来考察将数据从内存写入 SSTable 的时间: 将完成 transfer 前后的时间记录下来，相减得到将 memtable 写入 SSTable 所需要的时间，可得写入一次的时间大约为 650ms。以下是外存操作 (不包含 compaction) 所消耗的时间:



可以看到 put 操作所需的时间增长不明显，但 get 和 del 所需时间都明显增加。接着考察 put 指令的吞吐量: 连续执行 put 命令 20 秒，统计执行 put 命令的数量，计算平均每秒的 put 命令吞吐量。循环执行 50 次，得到结果如下：



从图上可以看到 transfer 和 compaction 对 put 吞吐量非常明显的影响。