**Project1:LSM Trees 键值存储系统**

**——项目设计文档**

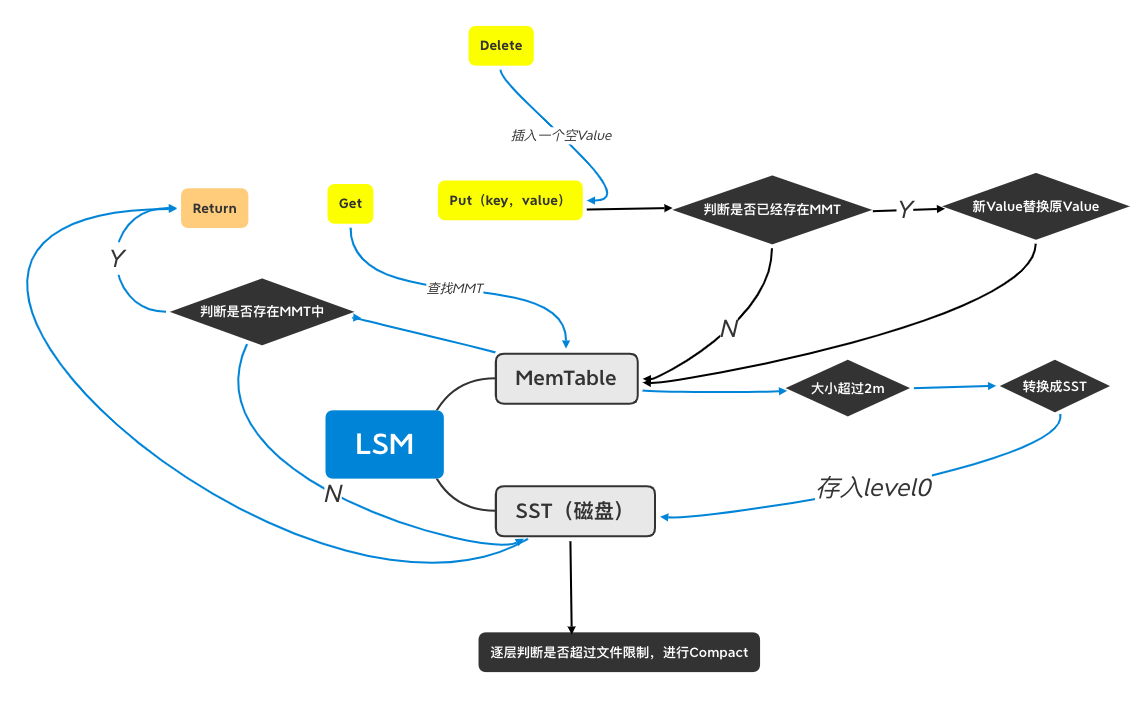
姓名：啊巴啊巴

学号：51802191\*\*\*\*

1. **简介**

基于 Log-Structured Merge Tree键值对存储系统，能够实现🧍‍大量键值对存储与修改（put）、查找（get）与删除（delete）操作，提供比传统的 B+ 树更好的写性能。LSM 通过将磁盘的随机写转化为顺序写来提高写性能，并通过cache等优化措施是的读性能不至于过低。

算法流程图如下：



1. **代码实现**
2. Node类

存储键值对节点，存储于mmt中，在sst读取时也会用到此类。

代码如下：

#pragma once

#include <cstdlib>

#include <string>

using namespace std;

struct node

{

uint64\_t key;

string value;

node \*up;

node \*down;

node \*succ;

node \*pred;

node();

node(uint64\_t key,const string &s);

void Delete(node \*t);

};

实现：

#include "node.h"

node::node(uint64\_t key , const string &s){

this->key = key ;

this->value = s ;

this->up = nullptr;

this->down = nullptr;

this->succ = nullptr;

this->pred = NULL;

}

node::node(){

this->up = nullptr;

this->down = nullptr;

this->succ = nullptr;

this->pred = NULL;

}

void node::Delete(node \*t){

node \*del=t;

while(t!=nullptr){

t=t->succ;

delete del;

del=t->succ;

}

}

1. Memtable部分

由skiplist组成，用于快速存储kv对，大小为2m，超过2m将转化为sst存入磁盘并清空mmt。

函数接口：

~KVStore();

void put(uint64\_t key, const std::string &s) override;//插入键值对

node\* Put(uint64\_t key, const std::string &s, node \*h,node \*t);//重载put，在h-t层插入键值对

void display();

void clear();//清空mmt

void grow(node \*t);//层数生长

std::string get(uint64\_t key) override;

bool del(uint64\_t key) override;//删除键值对

void reset() override;//清空mmt和所有sst

重要函数实现：

node\* KVStore::Put(uint64\_t key, const std::string &s, node \*h, node \*t){

node \*tmp = h;

node \*ta = t;

while(tmp->succ->key < key) tmp = tmp->succ;

if(tmp->succ->key == key&&tmp->succ != t) {//已经出现，换值

node \*ex = tmp->succ;

size+=(sizeof(char)\*s.size()-sizeof(char)\*ex->value.size());

while(ex){

ex->value = s;

ex=ex->up;

}

if(size>(1<<21)-128) {

sst->putFile(head,tail);//转化成sst

clear();

}

return nullptr;

}//换值

node \*flg = tmp->succ;

tmp->succ = new node(key,s);

flg->pred = tmp->succ;

tmp->succ->succ = flg;

tmp->succ->pred = tmp;//插入该层

node \*ne = tmp->succ;

if(h->up == nullptr){

h->up = new node(0,"");

t->up = new node(UINT64\_MAX,"");

h->up->down = h;

t->up->down = t;

t->up->pred = h->up;

h->up->succ = t->up;

}

if(rand()%2==0){

//如果可以生长

tmp=h->up;

ta =t->up;

node \*u = Put(key,s,tmp,ta);

ne->up = u;

u->down= ne;

}

if(h==head){

size+=(sizeof(uint64\_t)\*2+sizeof(char)\*s.size())+sizeof(int);

if(size>(1<<21)-128) {

sst->putFile(head,tail);//超过2m，转化成sst插入磁盘level0层

clear();//清空mmt

return nullptr;

}

}

return ne;

}

std::string KVStore::get(uint64\_t key)

{

node \*del=head->succ;

while(del!=nullptr){

if(del->key==key&&del!=tail) {

return del->value;

}

if(del->key>key) break;

del=del->succ;

}//查找mmt

return sst->get(key);//查找sst

}

1. SST部分

由SStable类实现，相关函数接口如下：

class SStable

{

private:

int level;

int \*num;

//int num0;

struct cell

{

// 存储文件单元，每个文件对应一个cell并存在缓存中;

string name;

int socindex;

string path;

uint64\_t minkey;

uint64\_t maxkey;

cell \*next;

cell \*down;

cell(){next=nullptr;down=nullptr;}

};

cell \*head;

public:

SStable(/\* args \*/);

void compact(int levelone,int leveltwo);

void putFileToLevel(node\* li,int le);

void deleteFilefromlevel(cell \*f,int le);

void Remove();

void putFile(node \*h,node \*t,int le=0);

void Delete(node \*t);

void dis(int le);

void showlist();

void updatecache();

void readcache();

string get(uint64\_t k);

~SStable();

};

其中，putfile函数将从mmt传递来的head和tail指针之间的kv节点写入磁盘level0，结束后检查level0磁盘文件数量，若超过2，则对0，1两层进行compact。

compact函数将对levelone和leveltwo两层进行合并。首先判断levelone是否为0，若为0，则取出0层所有文件（3个），并对他们进行归并排序存入newh1中。若levelone不为0，则将该层超出文件一次读入newh1中（一定是有序的）。此后将leveltwo中与newh1有交集的文件一次读入newh2中，并删除这些文件。此后归并newh1和newh2并插入leveltwo中对应位置。

在所有归并排序包括写入读取时都要注意动态释放内存避免内存泄漏。

putfiletolevel和deletefilefromlevel两个函数用于在插入删除文件后修改文件对应的cell并执行相应的文件更名操作。

updatecache在每一次putfile之后执行，用来将所有cell写在一个单独文件夹cache中，这样，每次程序被打断都可以重新读取cache来将cell通过readcache读入缓存。

get函数从sst中寻找键值对，由于level0是乱序，且后面的文件比前面的文件新，因此找到key之后继续查找level0，将最后一个key对应的value取出。当level0中不存在对应的key，则向下查找，level0之外的层一定没有重复key，因此找到一个key之后就不必再向下查找，将较上层的key对应的value取出。

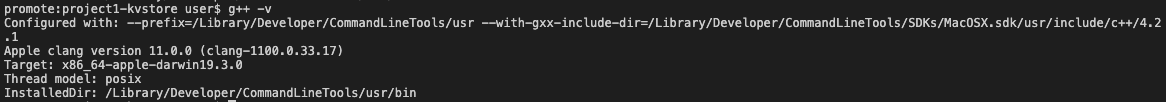
1. **运行效果**

程序可以正确通过correctness和persistence的所有phase。

测试环境为MacOS10.15，

IDE：VScode

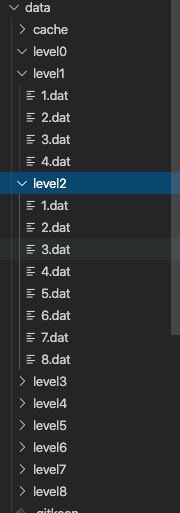
编译版本：



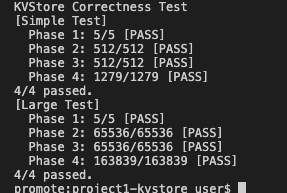
效果截图如下。

Correctness测试：

sst磁盘：文件夹由程序动态生成，每层文件最多为n^(n+1)个。



测试结果如下：



compact性能如下

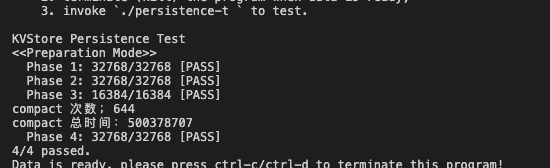
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试数据量 | 1024\*8 | 1024\*16 | 1024\*32 | 1024\*64 |
| Compact次数 | 10 | 71 | 436 | 2376 |
| Compact总时间 | 6610916ticks | 47929962ticks | 290306455ticks | 1608247284ticks |

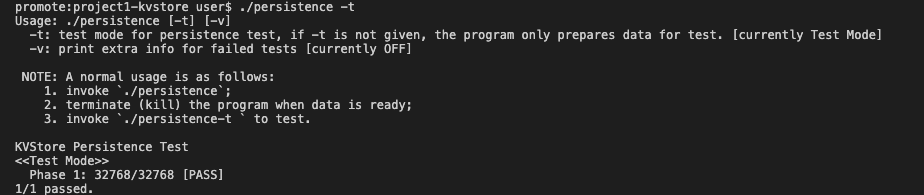
correctness测试中，键值对均以顺序插入，每次插入数据量约为32m，128m，512m，2048m，吞吐量提升四倍后，compact所需时间约增加6倍左右。compact占程序运行大多数时间。

**persistence测试**：

Disk部分同correctness测试。在persistence测试中，增加了乱序插入和键值对的修改，以及退出程序重新启动时的磁盘访问。

运行效果截图如下：





compact效果如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试数据量 | 1024\*8 | 1024\*16 | 1024\*32 |  |
| Compact次数 | 16 | 108 | 644 |  |
| Compact总时间 | 12505396ticks | 84100439ticks | 500378707ticks |  |

对比correctness测试，persistence测试中，相同数据量触发compact次数更多，此外，对于Compact所花费的平均时间更长。这可能与persistence中数据的插入机制有关。相比正确性测试，持久性测试中多出了更多的删除和修改操作，这使得compact过程多出了更多的比较择优操作。乱序插入也导致了在compact过程中，leveltwo总会选出一些文件与levelone文件有交集（在正确性测试中，这种情况是不会发生的），一致一次compact所需时间更长。

1. **性能测试**
2. **时延：在不进行compact的情况下获得每次put、delete、get操作所需要的时间。**

测试代码如下

store=new KVStore("./data");

for(int m=0;m<100;++m){

start=clock();

for(int i=1024\*m\*8;i<1024\*(m\*8+8);++i){

store->put(i,string(i+1,'x'));

}

end=clock();

time+=float(end-start)/CLOCKS\_PER\_SEC;

cout<<"put次数:"<<1024\*8\*(m+1)<<"\t时间:"<<time<<endl;

cout<<"平均每秒put:"<<1024\*8\*(m+1)/time<<"次"<<endl;

}

time=0;

for(int m=0;m<10;++m){

start=clock();

for(int i=1024\*m\*8;i<1024\*(m\*8+8);++i){

store->get(i);

}

end=clock();

time+=float(end-start)/CLOCKS\_PER\_SEC;

cout<<"get次数:"<<1024\*8\*(m+1)<<"\t时间:"<<time<<endl;

}

time =0;

cout<<"平均每次get:"<<time\*1000/(1024\*8\*10)<<"ms"<<endl;

for(int m=0;m<10;++m){

start=clock();

for(int i=1024\*m\*8;i<1024\*(m\*8+8);++i){

store->del(i);

}

end=clock();

time+=float(end-start)/CLOCKS\_PER\_SEC;

cout<<"delete次数:"<<1024\*8\*(m+1)<<"\t时间:"<<time<<endl;

}

cout<<"平均每次delete:"<<time\*1000/(1024\*8\*10)<<"ms"<<endl;

获得平均花费时间柱状图如下：

分析：由于没有compact操作，所有文件均存储在level0层，put和get读写花费时间相当，在delete操作中，每次delete都有判断是否在mmt中，随后判断是否在sst中，一次费时较长。

吞吐量测试：

测试在有compact和无compact情况下每秒put次数

测试代码如下

store=new KVStore("./data");

for(int m=0;m<100;++m){

start=clock();

for(int i=1024\*m\*8;i<1024\*(m\*8+8);++i){

store->put(i,string(i+1,'x'));

}

end=clock();

time+=float(end-start)/CLOCKS\_PER\_SEC;

cout<<"put次数:"<<1024\*8\*(m+1)<<"\t时间:"<<time<<endl;

cout<<"平均每秒put:"<<1024\*8\*(m+1)/time<<"次"<<endl;

}

无compact情况吞吐量如下：

有compact操作时，吞吐如下：

分析：

随着put操作次数增加，吞吐量将降低，这不难理解，由于插入的字符串越来越长，因此，生成sst并写入disk这一环节将越来越频繁。写磁盘对性能的影响是巨大的。

当有compact操作时，吞吐量相比无compact操作显著降低，这是由于compact操作需要更多的读文件写文件操作，拖慢了程序运行。

1. **感悟心得**

在这次project中，令我印象最为时刻的还是二进制文件读写问题，在这方面我遇到了不少问题。

比如，二进制文件读写的binary打开方式，缺少这一参数，二进制文件就不能正确读写。在读文件是，采用char [ ]来缓存数据要注意将最后一位制‘\0’，否则将会发生不可知的符号。另外，目录的读取，创建，删除，更名都需要更加深入的研究。

在设计方面，我采用了cell的方式来缓存文件，每一个cell包括了该文件的地址，索引位置，最小key最大key等关键属性，极大的改善了文件的存储问题。在disk的文件中，我将前一部分存储key-value，后一部分存储key-索引，并在索引末尾加上了一个0-索引起始位置以标注文件结尾而不产生其他影响。

最后，这次pro也让我熟悉了char\*，char[ ],string，const str等各种类型之间的联系、区别和转换，在这里感谢所有解答我疑惑的老师、助教和同学。