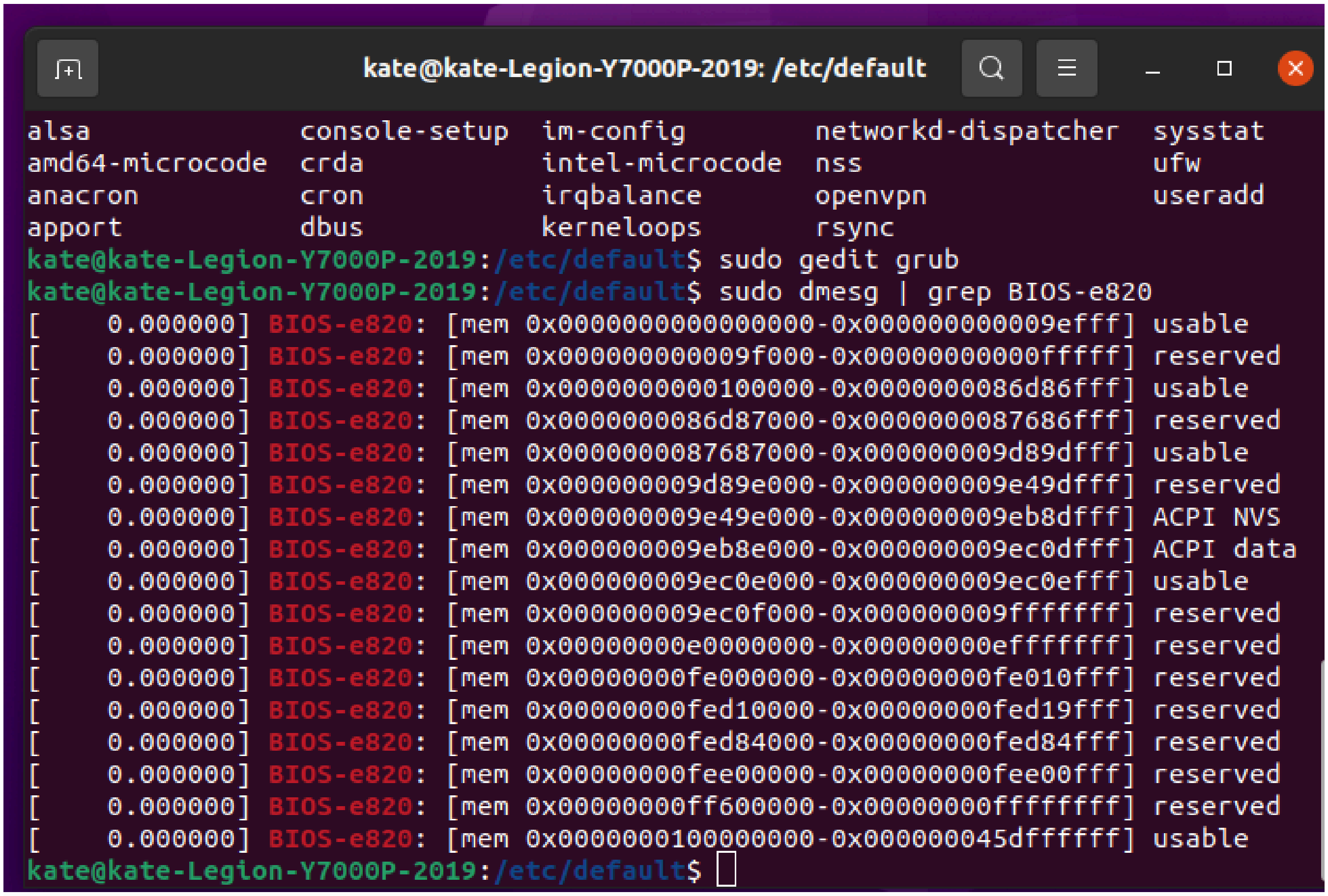
2020数据库课程设计实验报告

姓名	学号	github用户名
张尉卓	18339024	zzz4949
陈家博	18339002	JS9002
李征辉	18339011	Katelzh

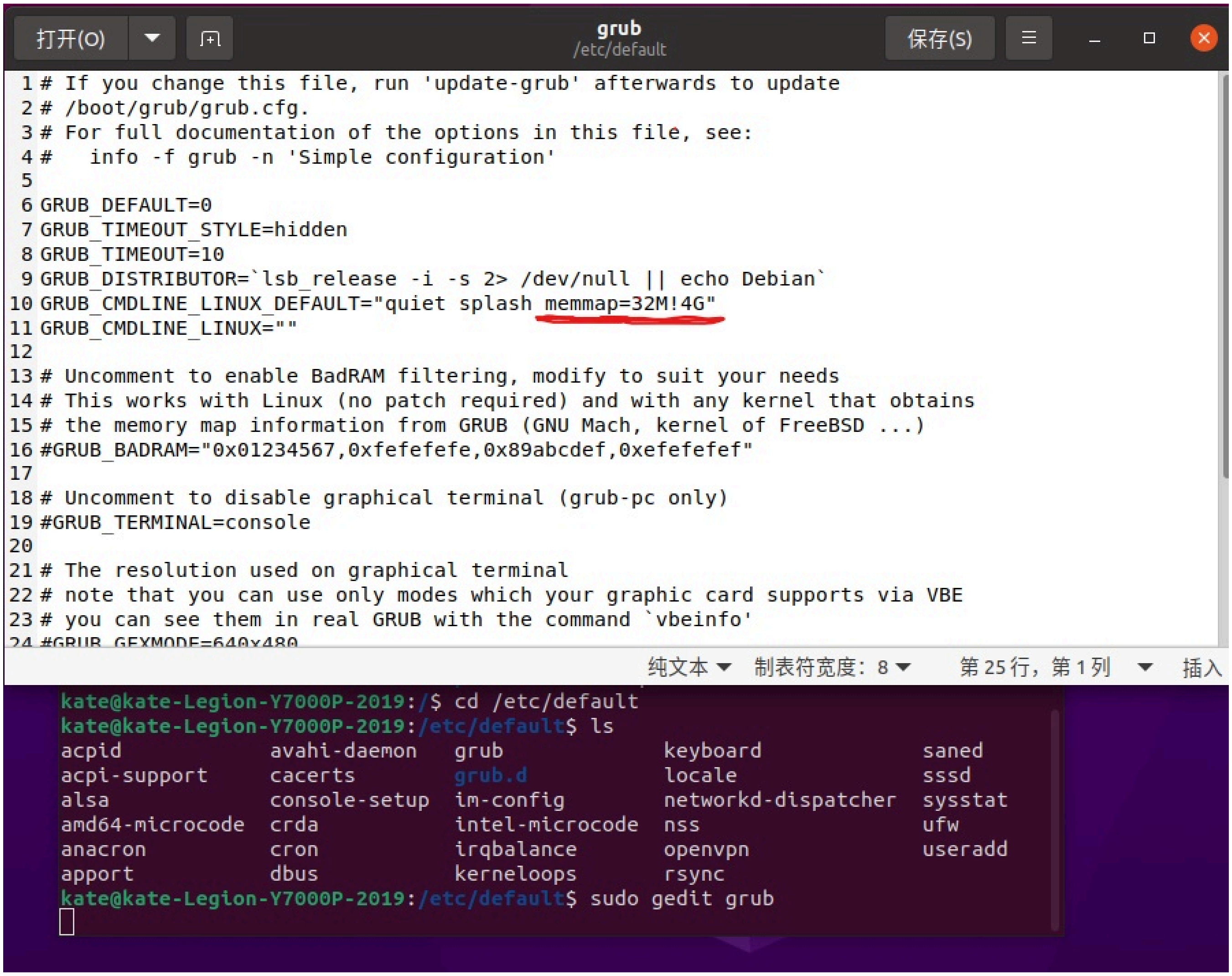
Github仓库地址： 我们的仓库
实验环境： Ubuntu 18+

一.利用普通内存模拟NVM环境

1. 根据官方教程，首先查看当前可用存储空间`sudo dmesg | grep BIOS-e820`，如图，选择在最后一行即`0x0000000100000000-0x000000045dffffff`的空间分配持久化空间；



2. 在目录 `/etc/default` 下修改 `grub` 文件，添加 `memmap=32M!4G` ，即从4G位置开始分配32MB大小的持久化空间；



3. 分配完成后在 `/dev` 目录下可以看到新增 `pmem0` 设备；

```
kate@kate-Legion-Y7000P-2019: /dev
dma_heap       i2c-23     null         tty14   tty46   ttyS19    vcsa
dri            i2c-3      nvme0        tty15   tty47   ttyS2     vcsa1
drm_dp_aux0    i2c-4      nvme0n1      tty16   tty48   ttyS20    vcsa2
drm_dp_aux1    i2c-5      nvme0n1p1    tty17   tty49   ttyS21    vcsa3
drm_dp_aux2    i2c-6      nvme0n1p2    tty18   tty5    ttyS22    vcsa4
ecryptfs       i2c-7      nvme0n1p3    tty19   tty50   ttyS23    vcsa5
fb0            i2c-8      nvme0n1p4    tty2    tty51   ttyS24    vcsa6
fd             i2c-9      nvme0n1p5    tty20   tty52   ttyS25    vcsu
full           initctl    nvme0n1p6    tty21   tty53   ttyS26    vcsu1
fuse           input      nvram        tty22   tty54   ttyS27    vcsu2
gpiochip0      kmsg       pmem0        tty23   tty55   ttyS28    vcsu3
hidraw0        kvm        port         tty24   tty56   ttyS29    vcsu4
hidraw1        lightnvm   ppp          tty25   tty57   ttyS3     vcsu5
hidraw2        log        psaux        tty26   tty58   ttyS30    vcsu6
hidraw3        loop0      ptmx         tty27   tty59   ttyS31    vfio
hidraw4        loop1      pts          tty28   tty6    ttyS4     vga_arbiter
hpet           loop10     random       tty29   tty60   ttyS5     vhci
hugepages      loop11     rfkill       tty3    tty61   ttyS6     vhost-net
hwrng          loop2      rtc          tty30   tty62   ttyS7     vhost-vsock
i2c-0          loop3      rtc0         tty31   tty63   ttyS8     video0
i2c-1          loop4      shm          tty32   tty7    ttyS9     video1
i2c-10         loop5      snapshot     tty33   tty8    udmabuf   zero
i2c-11         loop6      snd          tty34   tty9    uhid      zfs
kate@kate-Legion-Y7000P-2019:/dev$
```

二.根据PMDK的README安装教程进行库安装

1. 根据官方教程在自定义目录 `/home/zwz/DBMS/my_PMDK` 下通过命令 `git clone https://github.com/pmem/pmdk` 下载并安装PMDK；

```
zwz@LAPTOP-T800QGMC:~/DBMS/my_PMDK$ git clone https://github.com/pmem/pmdk.git
```

```
zwz@LAPTOP-T800QGMC:~/DBMS/my_PMDK/pmdk$ pwd
/home/zwz/DBMS/my_PMDK/pmdk
zwz@LAPTOP-T800QGMC:~/DBMS/my_PMDK/pmdk$
```

下载完毕；

```
zwz@LAPTOP-T800QGMC:~/DBMS/my_PMDK$ ls
pmdk
zwz@LAPTOP-T800QGMC:~/DBMS/my_PMDK$ cd pmdk
zwz@LAPTOP-T800QGMC:~/DBMS/my_PMDK/pmdk$ ls
CODING_STYLE.md  CONTRIBUTING.md  ChangeLog  LICENSE  Makefile  README.md  VERSION  appveyor.yml  doc  res  src  utils
zwz@LAPTOP-T800QGMC:~/DBMS/my_PMDK/pmdk$
```

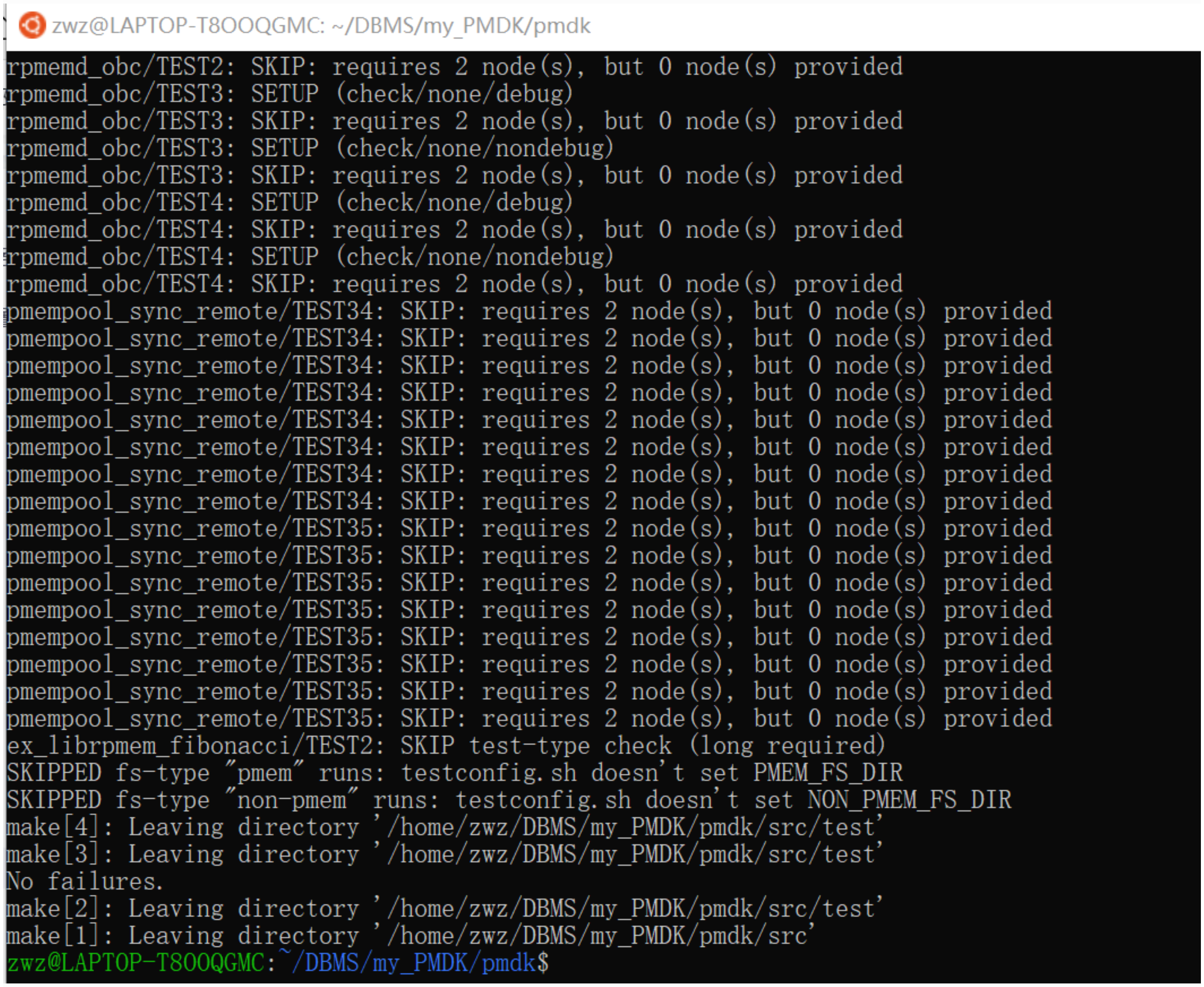
2. 在 `/home/zwz/DBMS/my_PMDK/pmdk` 目录下执行 `make` 命令失败，按照网络教程，需要安装 `make`，同时依次安装所需要的依赖包，具体安装语句如下所示：

```
sudo apt install make;
sudo apt install build-essential;
sudo apt-get install libdaxctl-dev;
sudo apt-get install libndctl-dev;
sudo apt-get install pandoc;
sudo apt-get install m4;
sudo apt-get install libfabric-dev;
```

进行pmdk测试;

```
cp src/test/testconfig.sh.example src/test/testconfig.sh;
make test;
make check;
```

如下显示测试结果无误;



```
zwz@LAPTOP-T8OOQGMC: ~/DBMS/my_PMDK/pmdk
rpmemd_obc/TEST2: SKIP: requires 2 node(s), but 0 node(s) provided
rpmemd_obc/TEST3: SETUP (check/none/debug)
rpmemd_obc/TEST3: SKIP: requires 2 node(s), but 0 node(s) provided
rpmemd_obc/TEST3: SETUP (check/none/nondebug)
rpmemd_obc/TEST3: SKIP: requires 2 node(s), but 0 node(s) provided
rpmemd_obc/TEST4: SETUP (check/none/debug)
rpmemd_obc/TEST4: SKIP: requires 2 node(s), but 0 node(s) provided
rpmemd_obc/TEST4: SETUP (check/none/nondebug)
rpmemd_obc/TEST4: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST34: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST34: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST34: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST34: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST34: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST34: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST34: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST34: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST35: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST35: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST35: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST35: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST35: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST35: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST35: SKIP: requires 2 node(s), but 0 node(s) provided
pmempool_sync_remote/TEST35: SKIP: requires 2 node(s), but 0 node(s) provided
ex_librpmem_fibonacci/TEST2: SKIP test-type check (long required)
SKIPPED fs-type "pmem" runs: testconfig.sh doesn't set PMEM_FS_DIR
SKIPPED fs-type "non-pmem" runs: testconfig.sh doesn't set NON_PMEM_FS_DIR
make[4]: Leaving directory '/home/zwz/DBMS/my_PMDK/pmdk/src/test'
make[3]: Leaving directory '/home/zwz/DBMS/my_PMDK/pmdk/src/test'
No failures.
make[2]: Leaving directory '/home/zwz/DBMS/my_PMDK/pmdk/src/test'
make[1]: Leaving directory '/home/zwz/DBMS/my_PMDK/pmdk/src'
zwz@LAPTOP-T8OOQGMC:~/DBMS/my_PMDK/pmdk$
```

3. 测试完毕后，执行`sudo make install`进行安装;

zwz@LAPTOP-T8OOQGMC: ~/DBMS/my_PMDK/pmdk/src

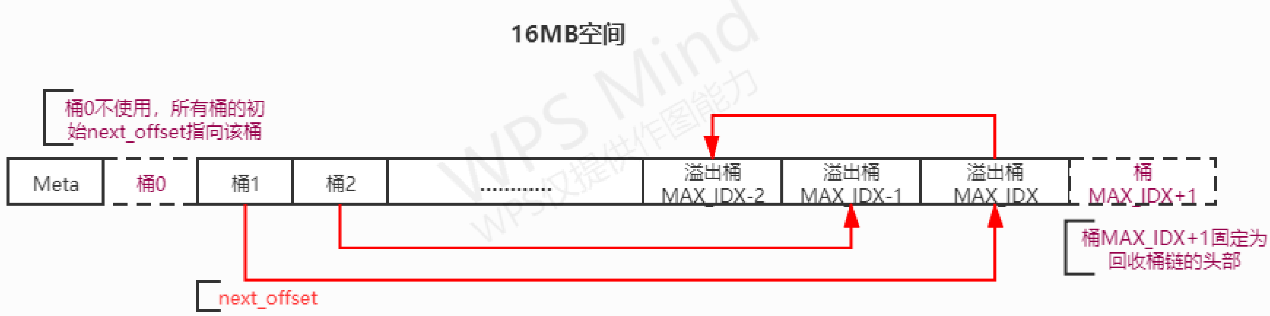
```
zwz@LAPTOP-T8OOQGMC:~/DBMS/my_PMDK/pmdk$ sudo make install
[sudo] password for zwz:
test -f .skip-doc || make -C doc all
make[1]: Entering directory '/home/zwz/DBMS/my_PMDK/pmdk/doc'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/zwz/DBMS/my_PMDK/pmdk/doc'
make -C src all
make[1]: Entering directory '/home/zwz/DBMS/my_PMDK/pmdk/src'
make -C libpmem
make[2]: Entering directory '/home/zwz/DBMS/my_PMDK/pmdk/src/libpmem'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/zwz/DBMS/my_PMDK/pmdk/src/libpmem'
make -C libpmem DEBUG=1
make[2]: Entering directory '/home/zwz/DBMS/my_PMDK/pmdk/src/libpmem'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/zwz/DBMS/my_PMDK/pmdk/src/libpmem'
make -C libpmemblk
make[2]: Entering directory '/home/zwz/DBMS/my_PMDK/pmdk/src/libpmemblk'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/zwz/DBMS/my_PMDK/pmdk/src/libpmemblk'
make -C libpmemblk DEBUG=1
make[2]: Entering directory '/home/zwz/DBMS/my_PMDK/pmdk/src/libpmemblk'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/zwz/DBMS/my_PMDK/pmdk/src/libpmemblk'
make -C libpmemlog
make[2]: Entering directory '/home/zwz/DBMS/my_PMDK/pmdk/src/libpmemlog'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/zwz/DBMS/my_PMDK/pmdk/src/libpmemlog'
make -C libpmemlog DEBUG=1
make[2]: Entering directory '/home/zwz/DBMS/my_PMDK/pmdk/src/libpmemlog'
```

安装成功；

```
zwz@LAPTOP-T8OOQGMC:~/DBMS/my_PMDK/pmdk$ cd src
zwz@LAPTOP-T8OOQGMC:~/DBMS/my_PMDK/pmdk/src$ ls
LongPath.manifest       PMDK.sln     common.inc  examples  libpmem2     libpmempool  nondebug      windows
LongPathSupport.props   README       common.pc   freebsd   libpmemblk   librpmem     rpmem_common
Makefile                benchmarks   core        include   libpmemlog   libvmem      test
Makefile.inc            common       debug       libpmem   libpmemobj   libvmmalloc  tools
zwz@LAPTOP-T8OOQGMC:~/DBMS/my_PMDK/pmdk/src$ cd libpmem
zwz@LAPTOP-T8OOQGMC:~/DBMS/my_PMDK/pmdk/src/libpmem$ ls
Makefile   libpmem.def   libpmem.link.in  libpmem.vcxproj          libpmem_main.c  pmem.h        pmem_windows.c
libpmem.c  libpmem.link  libpmem.rc       libpmem.vcxproj.filters  pmem.c          pmem_posix.c
zwz@LAPTOP-T8OOQGMC:~/DBMS/my_PMDK/pmdk/src/libpmem$
```

三.PML-Hash代码实现

实现思路



"pml_hash.h"

1.全局变量

`TABLE_SIZE` : **16**，即桶（包括原桶和溢出桶）的大小；

`HASH_SIZE` : **1**，初始的原桶的个数；

`FILE_SIZE` : **1024 * 1024 * 16**，即操作的文件大小为16MB；

`MAX_IDX` : **60000**，在有限空间内桶（包括原桶和溢出桶）的实际可达个数；

2.数据结构

```
typedef struct metadata {
    size_t size;            //原桶的个数
    size_t level;           //全局深度
    uint64_t next;          //指示分裂的next指针
    uint64_t overflow_num;  //溢出桶的个数
} metadata;

typedef struct entry {
    uint64_t key;           //桶中元素的键
    uint64_t value;         //桶中元素的值
} entry;

typedef struct pm_table {
    entry kv_arr[TABLE_SIZE];   //桶中元素数组
    uint64_t fill_num;          //桶中已有元素个数
    uint64_t next_offset;       //指针，指向下一个溢出桶
} pm_table;

//持久性线性哈希类
class PMLHash {
private:
    void* start_addr;      //文件起始地址
    metadata* meta;        //指针，指向元数据
    pm_table* table;       //指针，指向桶的起始地址（包括原桶和溢出桶）

    void split();
    uint64_t hashFunc(const uint64_t &key);
    uint64_t newOverflowIndex(); //新增溢出桶并返回其下标
    void recycle(uint64_t index); //回收空的溢出桶

public:
    PMLHash() = delete;
    PMLHash(const char* file_path);
    ~PMLHash();

    int insert(const uint64_t &key, const uint64_t &value);
    int search(const uint64_t &key, uint64_t &value);
    int remove(const uint64_t &key);
    int update(const uint64_t &key, const uint64_t &value);
    void print(); //打印现有的所有桶及其溢出桶
    void destroy(); //清空哈希表
};
```

"pml_hash.cpp"

1.初始化哈希表

1. 初始化，调用`pmem_map_file()`函数返回所取16MB大小空间的起始地址`start_addr`，并通过该地址得到`meta`和`table`的起始地址；
2. 若读取文件后`meta->size = 0`，即新创建文件，此时需初始化：原桶的个数初始为**1**（1号桶）；全局深度为**0**；溢出桶个数为**0**；指示分裂的`next = 1`（指向1号桶）；
3. 将可能用到的每一个桶初始化：桶中元素个数为0；指示下一个溢出桶的`next_offset = 0`（指向不使用的0号桶）；

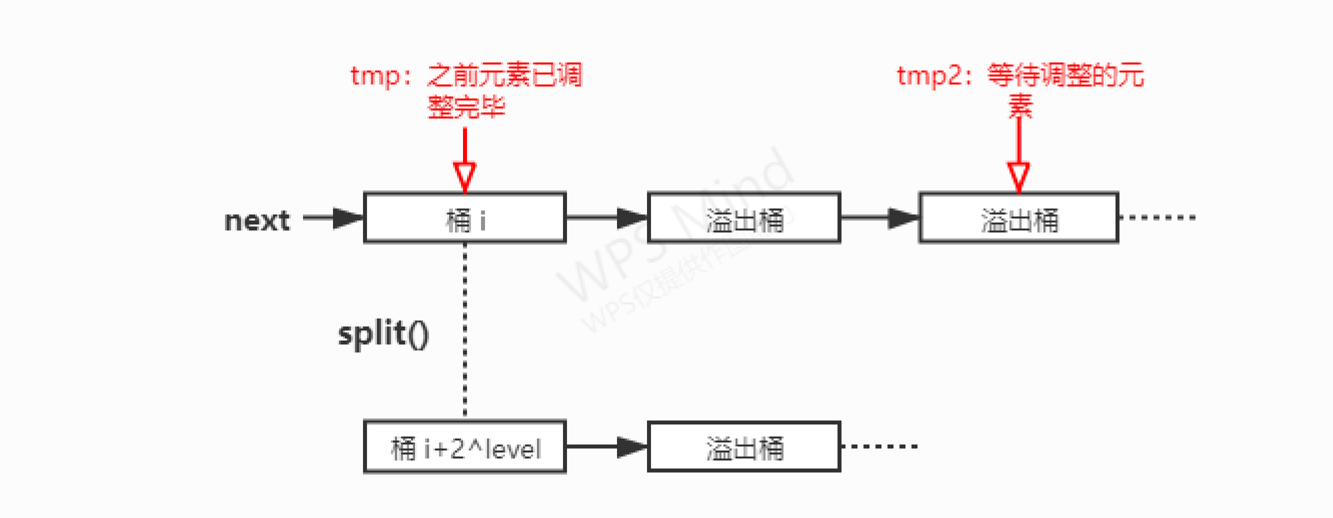
```
PMLHash::PMLHash(const char* file_path) {
        size_t mapped_len;
        int is_pmem;
        if((start_addr = pmem_map_file(file_path, FILE_SIZE, PMEM_FILE_CREATE,
                0666, &mapped_len, &is_pmem)) == NULL){
                exit(0);
        }
        meta = (metadata*)start_addr;
        table = (pm_table*)((char*)start_addr + sizeof(meta));

    //初始化数据结构
        if(meta->size == 0){
                meta->size = 1;
                meta->level = 0;
                meta->overflow_num = 0;
                meta->next = 1;

                for(int i = 0 ; i < MAX_IDX + 2 ; i++ ){
                        table[i].fill_num = 0;
                        table[i].next_offset = 0;
                }
        }
}
```

2.桶的分裂

1. 通过`next`得到待分裂的桶序号，同时根据全局深度由 $\mathbf{next+2^{level}}$ 得到分裂产生的新桶的序号；
2. 遍历待分裂的桶及其溢出桶，对每一对键值对重新分配其所在的桶位置；
3. 如图，两个指针在原桶及其溢出桶上移动，`tmp`指示的桶位置对应该位置之前的所有元素已确定位置；`tmp2`指示待重新分配元素所在的桶位置；
4. 通过 $\mathbf{key\ \%\ 2^{level+1}+1}$ 确定待分裂桶中元素的新位置（原桶/新桶），每重新填满一个原桶（或其溢出桶），通过`tmp = table[tmp].next_offset`向后移动；若新桶填满，则调用`newOverflowIndex()`新分配其溢出桶得到其序号继续上述移动；
5. 通过`tmp`调用`recycle()`对空闲溢出桶进行回收，修改对应桶的`fill_num`，原桶的个数加1；
6. 对`next`判断，若到达当前全局深度的最后一个原桶，则重置`next = 1`，全局深度加1；否则`next++`；


tmp：之前元素已调整完毕
tmp2：等待调整的元素
next
桶 i
溢出桶
溢出桶
split()
桶 i+2^level
溢出桶

```
void PMLHash::split() {
        uint64_t index = meta->next;
        uint64_t newt = meta->next + (1<<(meta->level)) ;

        uint64_t i , j , k , tmp , tmp2;
        tmp = index; //指针，指向待分裂且已调整好元素位置的桶位置
        tmp2 = index; //指针，指向待分裂的等待调整的元素的桶位置
        j = 0;
        k = 0;
        while(tmp2){        //由tmp2和i遍历要分裂的桶
                for(i = 0 ; i < table[tmp2].fill_num ; i++ ){
                        //由tmp和j确定留在旧桶的数据的位置
                        if(table[tmp2].kv_arr[i].key % (1<<(meta->level+1)) + 1 == index){
                                table[tmp].kv_arr[j].key = table[tmp2].kv_arr[i].key;
                                table[tmp].kv_arr[j].value = table[tmp2].kv_arr[i].value;
                                if(++j == TABLE_SIZE){
                                        table[tmp].fill_num = TABLE_SIZE;
                                        tmp = table[tmp].next_offset;
                                        j = 0;
                                }
                        }
                        else{   //由newt和k确定留在新桶的数据的位置
                                if(k == TABLE_SIZE){
                                        table[newt].fill_num = TABLE_SIZE;
                                        newt = newOverflowIndex();
                                        k = 0;
                                }
                                table[newt].kv_arr[k].key = table[tmp2].kv_arr[i].key;
                                table[newt].kv_arr[k].value = table[tmp2].kv_arr[i].value;
                                k++;
                        }
                }
                tmp2 = table[tmp2].next_offset;
        }

        recycle(table[tmp].next_offset); //回收溢出桶
        table[tmp].fill_num = j;
        table[tmp].next_offset = 0;
        table[newt].fill_num = k;   //修改存有数据的桶的信息
        meta->size++;
        if(meta->next == (1<<(meta->level))){
                meta->next = 1;
                meta->level++;   //原桶全部分裂，修改深度和next
        }
        else meta->next++;
}
```

3.哈希函数返回对应原桶的序号

1. 根据 **index=key % 2^{level}+1** 找到原桶的序号；
2. 判断该序号是否在 `next` 的前面，若 `index < next`，则需计算 **index=key % $2^{level+1}$+1**；

```
uint64_t PMLHash::hashFunc(const uint64_t &key ) {
        uint64_t index = key % (1<< meta->level) + 1;
        if(index < meta->next)
                index =  key % (1<< (meta->level+1)) + 1;
        return index;
}
```

4.新分配一个溢出桶并返回其序号

1. `recycle_head` 为已回收桶链的头部，若回收桶链中有已回收的桶可用，即 `table[recycle_head].next_offset != 0`，则使用该桶并将回收桶链头部指针后移；
2. 若无可用的已回收桶，则从16MB固定空间中尾部开始找到一个空闲桶（序号为 `MAX_IDX - overflow_num` 作为新溢出桶；同时如果与原桶序号冲突即已将16MB空间使用完毕，则令新溢出桶序号为0，后续操作将失败；
3. 返回新分配的溢出桶的序号；

```
uint64_t PMLHash::newOverflowIndex(){
        uint64_t recycle_head = MAX_IDX + 1;    //回收桶的头部
        uint64_t rh = recycle_head;
        uint64_t index;
        if(table[rh].next_offset){          //有可用的回收桶
                index = table[rh].next_offset;
                table[rh].next_offset = table[index].next_offset;
        }
        else{
                index = MAX_IDX - (meta->overflow_num); //从右往左找一个空闲桶作溢出桶
                if(index <= (meta->size)) index = 0;  //没有可用的溢出桶
                if(index) meta->overflow_num++;
        }
        table[index].next_offset = 0;
        table[index].fill_num = 0;
        return index;
}
```

5.回收溢出桶

1. 固定回收桶链的头部为序号 `recycle_head = MAX_IDX + 1`，空闲桶都将插入该链；
2. 对于即将回收的桶（输入为该桶序号），其后续的溢出桶一定也为空桶需一同回收，即存在**待回收的溢出桶链**，使用 `while` 循环找到该链的尾部位置，将该链整体插入**回收桶链**的头部（`MAX_IDX + 1` 号桶）之后，完成回收；

```
void PMLHash::recycle(uint64_t index){
        uint64_t tmp = table[MAX_IDX + 1].next_offset;
        uint64_t tmp2 = index;
        table[index].fill_num = 0;
        while(table[tmp2].next_offset){   //回收一连串的溢出桶
                tmp2 = table[tmp2].next_offset;
                table[tmp2].fill_num = 0;  //将回收桶的数据设为无效状态
        }
        table[tmp2].next_offset = table[tmp].next_offset;
        table[MAX_IDX + 1].next_offset = index;
}
```

6.插入键值对

1. 通过 `key` 调用 `hashFunc()` 得到原桶序号;
2. `while` 循环遍历该原桶及其溢出桶，直到找到未满的桶（即 `table.fill_num != TABLE_SIZE`）时，结束循环；特殊情况为该桶及其溢出桶都为**恰好填满**，此时需要新分配溢出桶，调用 `newOverflowIndex()` 并最终得到该桶的序号;
3. 根据得到的桶的序号插入键值对;
4. 若此序号不为原桶序号，即有溢出桶存在，则调用 `split()` 将 `next` 指向的桶分裂;
5. 插入成功则返回0，返回前调用 `pmem_persist()`；

```
int PMLHash::insert(const uint64_t &key, const uint64_t &value) {
        uint64_t index = hashFunc(key);
        uint64_t tmp = index;

        while(table[tmp].fill_num == TABLE_SIZE){ //找到空的桶
                if(table[tmp].next_offset == 0)
                        table[tmp].next_offset = newOverflowIndex();
                tmp= table[tmp].next_offset;
                if(tmp == 0)      //没有可用溢出空间，插入失败
                        return -1;
        }

        table[tmp].kv_arr[table[tmp].fill_num].key = key;
        table[tmp].kv_arr[table[tmp].fill_num].value = value;
        table[tmp].fill_num++;
        if(tmp != index)
                split();
        pmem_persist(start_addr, FILE_SIZE);
        return 0;
}
```

7.根据键查找相应的值

通过 `key` 调用 `hashFunc()` 得到原桶序号后，执行遍历查找（包括通过 `next_offset` 遍历查找溢出桶），若找到则记录其对应值，返回0，遍历完未找到则返回-1;

```
int PMLHash::search(const uint64_t &key, uint64_t &value) {
        uint64_t tmp = hashFunc(key);
        while(tmp){  //遍历原桶及其所有溢出桶
                for(int i = 0 ; i < table[tmp].fill_num ; i++ ){
                        if(table[tmp].kv_arr[i].key == key){
                                value = table[tmp].kv_arr[i].value;
                                return 0;
                        }
                }
                tmp = table[tmp].next_offset;
        }
        return -1;
}
```

8.根据键删除桶中的键值对

1. 通过`key`调用`hashFunc()`得到原桶序号；
2. 两个指针，`tmp`指向该序号对应原桶的最后一个桶（原桶/溢出桶），`tmp2`指向`tmp`的前一个桶（即倒数第二个桶）；
3. 遍历对应序号的原桶及其溢出桶，若找到对应`key`值，用`tmp`指向的桶（即最后一个桶）中的最后一个元素与其替换，并`table[tmp].fill_num--`；
4. 若恰好使得最后一个溢出桶为空，则调用`recycle()`将其回收并修改倒数第二个桶指针，即`table[tmp2].next_offset = 0`；
5. 删除成功返回0，未找到该`key`值返回-1，返回前调用`pmem_persist()`；

```
int PMLHash::remove(const uint64_t &key) {
        uint64_t index = hashFunc(key);
        uint64_t tmp = index;
        uint64_t tmp2 = index;

        while(table[tmp].next_offset){
                tmp2 = tmp;         //tmp2为index对应的“倒数第二”个桶（可能只有一个桶）
                tmp = table[tmp].next_offset;   //tmp为最后一个桶
        }

        while(index){
                for(int i = 0 ; i < table[index].fill_num ; i++ ){
                        if(table[index].kv_arr[i].key == key){
                                table[tmp].fill_num--;   //删除的位置需要由“最后一个溢出桶”的最后一个元素来填充
                                table[index].kv_arr[i].value = table[tmp].kv_arr[table[tmp].fill_num].value;
                                table[index].kv_arr[i].key = table[tmp].kv_arr[table[tmp].fill_num].key;

                                if((table[tmp].fill_num == 0) && (tmp != hashFunc(key))){
                                        table[tmp2].next_offset = 0;
                                        recycle(tmp);    //回收空的溢出桶
                                }
                                pmem_persist(start_addr, FILE_SIZE);
                                return 0;
                        }
                }
                index = table[index].next_offset;
        }
        pmem_persist(start_addr, FILE_SIZE);
        return -1;
}
```

9.更新键对应的值

1. 通过 `key` 调用 `hashFunc()` 得到原桶序号后，执行遍历查找（包括通过 `next_offset` 遍历查找溢出桶），若找到则修改其对应值，返回0，遍历完未找到则返回-1；
2. 返回前调用 `pmem_persist()`；

```
int PMLHash::update(const uint64_t &key, const uint64_t &value) {
        uint64_t tmp = hashFunc(key);
        while(tmp){   //遍历查找
                for(int i = 0 ; i < table[tmp].fill_num ; i++){
                        if(table[tmp].kv_arr[i].key == key){
                                table[tmp].kv_arr[i].value = value;
                                pmem_persist(start_addr, FILE_SIZE);
                                return 0;
                        }
                }
                tmp = table[tmp].next_offset;
        }
        pmem_persist(start_addr, FILE_SIZE);
        return -1;
}
```

10.按次序打印所有桶（包括其溢出桶）

遍历现有的原桶，以键值对形式打印其所有元素，若有溢出桶则依次打印其溢出桶元素；

```
void PMLHash::print(){ //打印结果验证
        uint64_t tmp;
        int i , j;
        for(i = 1 ; i <= meta->size ; i++){
                tmp = i;
                j = 0;
                while(tmp){
                        if(j == 0) cout << " table " << i << ": " ;
                                else cout << " -- overflow: " ;
                        for(int k = 0 ; k < table[tmp].fill_num ; k++){
                                cout << "(" << table[tmp].kv_arr[k].key << ","
                                     << table[tmp].kv_arr[k].value << ") ";
                        }
                        cout << endl;
                        tmp = table[tmp].next_offset;
                        j++;
                }
                cout << endl << endl;
        }
}
```

11.清空哈希表

清空已创建的哈希表，即恢复其初始状态，类似类的初始化；（用于YCSB测试使用）

```
void PMLHash::destroy(){
    meta->size = 1;
    meta->level = 0;
    meta->overflow_num = 0;
    meta->next = 1;

    for(int i = 0 ; i < MAX_IDX + 2 ; i++ ){
        table[i].fill_num = 0;
        table[i].next_offset = 0;
    }
    pmem_persist(start_addr, FILE_SIZE);
}
```

"main.cpp"

```
#include "pml_hash.h"
int main() {
        PMLHash hash("/home/zwz/test/newfile");
        for (uint64_t i = 1; i <= 17; i++) {
            hash.insert(i, i);
        }
        cout << "Insert(1 ~ 17) OK!" << endl;
        hash.print();

        for (uint64_t i = 18; i <= 33; i++) {
            hash.insert(i, i);
        }
        cout << "Insert(18 ~ 33) OK!" << endl;
        hash.print();

        for (uint64_t i = 34; i <= 35; i++) {
            hash.insert(i, i);
        }
        cout << "Insert(34 ~ 35) OK!" << endl;
        hash.print();

        for (uint64_t i = 15; i <= 20; i++) {
            uint64_t val;
            hash.search(i, val);
            cout << "Search(key: " << i << ")--> (value: " << val << ")" << endl;
        }

        for (uint64_t i = 15; i <= 20; i++) {
            hash.remove(i);
        }
        cout << "Remove(15 ~ 20) OK!" << endl;
        hash.print();

        for (uint64_t i = 25; i <= 30; i++) {
            hash.update(i, i+1);
            cout << "Update(" << i << ", " << i <<") -->"
                << "(" << i << ", " << i+1 << ") OK!" << endl;
        }
        hash.print();
        return 0;
}
```

代码测试

将 `pml_hash.cpp` 和 `main.cpp` 分别编译为 `pml_hash.o` 和 `main.o` 文件，并将其编译得到 `pml.out` 的可执行文件，`./pml` 执行程序；

```
zwz@LAPTOP-T800QGMC:~/test$ ls
ex  ex.cpp  main.cpp  pml_hash.cpp  pml_hash.h
zwz@LAPTOP-T800QGMC:~/test$ g++ -c pml_hash.cpp -lpmem -o pml_hash -I /home/zwz/test
zwz@LAPTOP-T800QGMC:~/test$ g++ -c main.cpp -lpmem -o main -I /home/zwz/test
zwz@LAPTOP-T800QGMC:~/test$ ls
ex  ex.cpp  main  main.cpp  pml_hash  pml_hash.cpp  pml_hash.h
zwz@LAPTOP-T800QGMC:~/test$ g++ main pml_hash -lpmem -o pml
zwz@LAPTOP-T800QGMC:~/test$ ls
ex  ex.cpp  main  main.cpp  pml  pml_hash  pml_hash.cpp  pml_hash.h
zwz@LAPTOP-T800QGMC:~/test$
```

1. 如结果所示，由初始的**桶1**（`level = 0`），插入1~16恰好填满，当插入17时触发分裂得到**桶1、2**并重新分配元素，`level = 1`，`next = 1`；
2. 插入18~33，当插入32后**桶1、2**恰好填满，接着插入33至**桶2**，有溢出，触发`next`指向的**桶1**分裂，`next = 2`；
3. 插入34~35，当插入35至**桶2**后，触发**桶2**分裂，`level = 2`，`next = 3`；

```
zwz@LAPTOP-T800QGMC:~/test$ ./pml
Insert(1 ~ 17) OK!
 table 1: (2,2) (4,4) (6,6) (8,8) (10,10) (12,12) (14,14) (16,16)


 table 2: (1,1) (3,3) (5,5) (7,7) (9,9) (11,11) (13,13) (15,15) (17,17)


Insert(18 ~ 33) OK!
 table 1: (4,4) (8,8) (12,12) (16,16) (20,20) (24,24) (28,28) (32,32)


 table 2: (1,1) (3,3) (5,5) (7,7) (9,9) (11,11) (13,13) (15,15) (17,17) (19,19) (21,21) (23,23) (25,25) (27,27) (29,29)
(31,31)
 -- overflow : (33,33)


 table 3: (2,2) (6,6) (10,10) (14,14) (18,18) (22,22) (26,26) (30,30)


Insert(34 ~ 35) OK!
 table 1: (4,4) (8,8) (12,12) (16,16) (20,20) (24,24) (28,28) (32,32)


 table 2: (1,1) (5,5) (9,9) (13,13) (17,17) (21,21) (25,25) (29,29) (33,33)


 table 3: (2,2) (6,6) (10,10) (14,14) (18,18) (22,22) (26,26) (30,30) (34,34)


 table 4: (3,3) (7,7) (11,11) (15,15) (19,19) (23,23) (27,27) (31,31) (35,35)
```

查找`key`为15~20并得到其对应`value`；

删除`key`为15~20的元素；

```
Search(key: 15)--> (value: 15)
Search(key: 16)--> (value: 16)
Search(key: 17)--> (value: 17)
Search(key: 18)--> (value: 18)
Search(key: 19)--> (value: 19)
Search(key: 20)--> (value: 20)
Remove(15 ~ 20) OK!
 table 1: (4,4) (8,8) (12,12) (32,32) (28,28) (24,24)


 table 2: (1,1) (5,5) (9,9) (13,13) (33,33) (21,21) (25,25) (29,29)


 table 3: (2,2) (6,6) (10,10) (14,14) (34,34) (22,22) (26,26) (30,30)


 table 4: (3,3) (7,7) (11,11) (35,35) (31,31) (23,23) (27,27)
```

将25~30的`key`对应的`value`加1；

```
Update(25, 25) -->(25, 26) OK!
Update(26, 26) -->(26, 27) OK!
Update(27, 27) -->(27, 28) OK!
Update(28, 28) -->(28, 29) OK!
Update(29, 29) -->(29, 30) OK!
Update(30, 30) -->(30, 31) OK!
 table 1: (4,4) (8,8) (12,12) (32,32) (28,29) (24,24)


 table 2: (1,1) (5,5) (9,9) (13,13) (33,33) (21,21) (25,26) (29,30)


 table 3: (2,2) (6,6) (10,10) (14,14) (34,34) (22,22) (26,27) (30,31)


 table 4: (3,3) (7,7) (11,11) (35,35) (31,31) (23,23) (27,28)
```

四.YCSB测试

编写YCSB测试

1. `convert()`函数用于截取测试集中数据的前8字节；
2. 进行YCSB测试，5次循环中都先读取`10w-rw-0-100-load.txt`初始化数据库，接着读取不同“读写比”的`run.txt`，统计每一个运行阶段的总耗时、OPS、延时的结果并输出；
3. 每一次运行`run.txt`测试完后调用`hash.destroy()`清空数据库并在下一次运行前重新初始化；

```
uint64_t convert(string str){
        uint64_t s = str[0] - '0';
        for(int i = 1 ;i< 8 ; i++){
                s *= 10;         //读取前八字节
                s += str[i] - '0';
        }
        return s;
}

int main(){
        PMLHash hash("/home/zwz/test/pml_hash");
        uint64_t key  , value;
        string op , str;
        char file[10][50] ={"/home/zwz/test/data/10w-rw-0-100-load.txt",
                            "/home/zwz/test/data/10w-rw-0-100-run.txt" ,
                            "/home/zwz/test/data/10w-rw-25-75-run.txt" ,
                            "/home/zwz/test/data/10w-rw-50-50-run.txt" ,
                            "/home/zwz/test/data/10w-rw-75-25-run.txt" ,
                            "/home/zwz/test/data/10w-rw-100-0-run.txt" };
                                              //benchmark数据集绝对路径
        for(int i = 1 ; i < 6 ; i++){

                ifstream read(file[0]);
                if(!read.is_open()) {
                        cout << "load failed " << endl;
                        cin.get();
                        return 0;
                }
                cout<<"Start loading("<<i<<")..."<<endl;
                while(!read.eof()){ //读取load文件
                        read >> op >> str;
                        key = convert(str);
                        hash.insert(key , key);//key和value默认相等
                }

                cout<<"Load("<<i<<") successfully!"<<endl;
                read.close();

                ifstream read2(file[i]);
                if(!read2.is_open()) {
                        cout << "run failed " << endl;
                        cin.get();
                        return 0;
                }
                int t = 0 , insert_num = 0 , read_num = 0;
                clock_t start , end;
                double time = 0;
                cout<<"Start running("<<i<<")..."<<endl;
                while(!read2.eof()){
                        read2 >> op >> key;
                        t++;

                        if(op[0]=='I') {   //“读取到‘INSERT’操作
                                insert_num++;
                                start = clock();
                                hash.insert(key , key);
                        }
```

```
                else {      //其余为‘SEARCH’操作
                        read_num++;
                        start = clock();
                        hash.search(key , value);
                }
                end = clock();
                time += end - start;//统计时间
                if(t==10000) break;  //eof的原因会多读一行
        }
        cout<<"Run("<<i<<") successfully!"<<endl;

        cout << "Operations size : " << t  << endl; //输出运行指标
        cout << "Total time cost : " << time/1000  << " ms " << endl;
        cout << "Insert_num      : " << insert_num << endl;
        cout << "Search_num      : " << t -  insert_num << endl;
        cout << "OPS             : " << t/(time/1000000) << endl;
        cout << "Avg_Latency     : " << time/t << " us " << endl;
        cout << endl<< endl;
        read2.close();
        cin.get();
        hash.destroy(); //清空
    }
    return 0;
}
```

测试结果

测试结果如图，可以看到每次运行测试所用`insert`和`search`的指令条数以及运行OPS和时延；

kate@kate-Legion-Y7000P-2019: ~/test

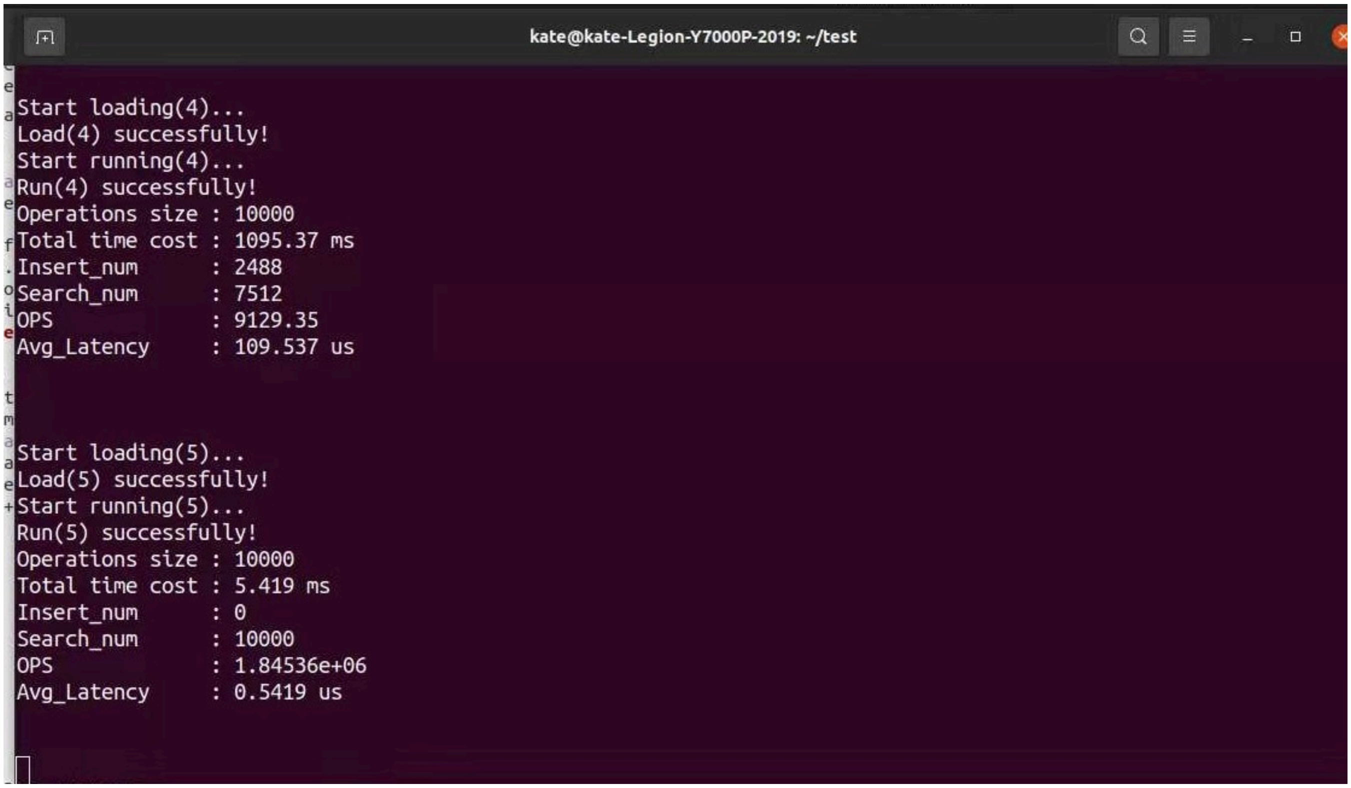
```
Start loading(1)...
Load(1) successfully!
Start running(1)...
Run(1) successfully!
Operations size : 10000
Total time cost : 7097.31 ms
Insert_num      : 10000
Search_num      : 0
OPS             : 1408.98
Avg_Latency     : 709.731 us



Start loading(2)...
Load(2) successfully!
Start running(2)...
Run(2) successfully!
Operations size : 10000
Total time cost : 3187.49 ms
Insert_num      : 7516
Search_num      : 2484
OPS             : 3137.27
Avg_Latency     : 318.748 us



Start loading(3)...
Load(3) successfully!
Start running(3)...
Run(3) successfully!
Operations size : 10000
Total time cost : 2254.4 ms
Insert_num      : 5102
Search_num      : 4898
OPS             : 4435.77
Avg_Latency     : 225.44 us
```


kate@kate-Legion-Y7000P-2019: ~/test
Start loading(4)...
Load(4) successfully!
Start running(4)...
Run(4) successfully!
Operations size : 10000
Total time cost : 1095.37 ms
Insert_num : 2488
Search_num : 7512
OPS : 9129.35
Avg_Latency : 109.537 us
Start loading(5)...
Load(5) successfully!
Start running(5)...
Run(5) successfully!
Operations size : 10000
Total time cost : 5.419 ms
Insert_num : 0
Search_num : 10000
OPS : 1.84536e+06
Avg_Latency : 0.5419 us