

电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

Linux 环境高级编程

LINUX ENVIRONMENT ADVANCED PROGRAMMING



大作业题目 存储表的设计

专 业	电子信息
学 号	202222080625
姓 名	乔翱
授课老师	李林
学 院	计算机科学与工程学院

目 录

第一章 需求分析	1
1.1 总体需求	1
1.2 存储需求	2
1.3 添加需求	2
1.4 搜索需求	2
1.5 索引需求	2
1.6 并发需求	2
1.7 测试需求	3
1.8 其他需求	3
第二章 总体设计	4
2.1 系统整体架构	4
2.2 存储表核心部分设计	5
2.2.1 存储表添加数据功能设计	5
2.2.2 存储表搜索数据功能设计	5
2.3 B+ 树设计	5
第三章 详细设计与实现	7
3.1 基本数据结构的设计与实现	7
3.2 B+ 树的设计与实现	8
3.3 存储表的设计与实现	10
3.3.1 存储表添加数据功能详细设计与实现	12
3.3.2 存储表搜索数据功能详细设计与实现	12
3.4 API 封装及多线程的设计与实现	13
第四章 测试	15
4.1 测试环境	15
4.2 测试用例及结果	15
4.2.1 添加功能测试	15
4.2.2 搜索功能测试	16
4.2.3 索引功能测试	17
4.2.4 并发功能测试	18
第五章 总结	20

第一章 需求分析

本章主要分析了系统的需求，包括系统的总体需求，以及从不同的方面全面地对系统的需求进行了分析，包括存储需求、搜索需求、并发需求等。

1.1 总体需求

本次大作业的总体需求是编写针对一个表进行存储，将表的相关数据存储到文件系统中，并且可以针对表进行添加、查询等一系列的操作，对这些针对表的功能进行封装，以 API 的形式提供给应用程序使用。本系统的用例图如图1-1所示。该用例图描述了存储表系统的整体需求，在后面将详细介绍本系统的一些具体需求。

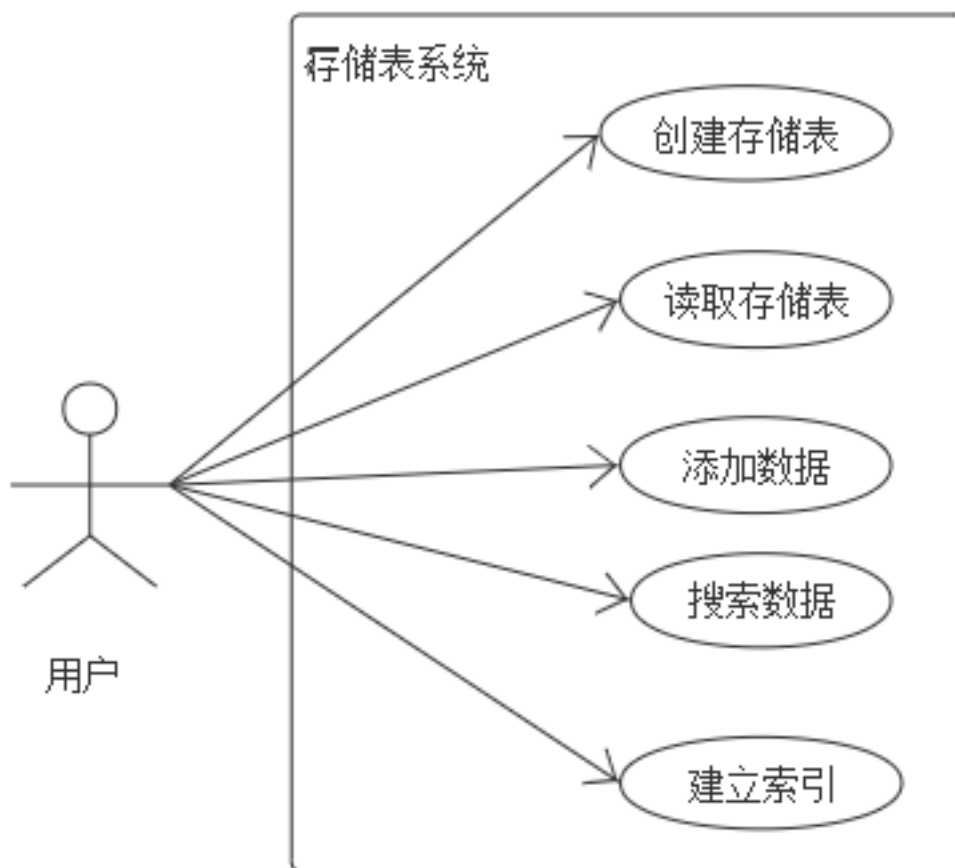


图 1-1 存储表系统用例图

1.2 存储需求

利用学习过的 Linux 系统中的文件操作 API，例如 `create`、`leek` 等文件操作 API，把表的相关数据存储到文件系统中。

该存储表的每一个数据项都具有 100 个属性，并且每个属性的类型都是 `int64_t` 类型，即每条数据的每个属性占 8 个字节的大小。

另外，该存储表需要支持的最大存储行数为 1 百万行。

1.3 添加需求

随机产生一条数据，插入到表格末尾，即写入存储表的文件的末尾，并且对该功能进行封装，提供 API 函数，供应用程序使用。

1.4 搜索需求

对表格中的数据按条件进行搜索，搜索条件可以由用户指定，用户可以指定在哪一个属性上进行搜索，并且指定搜索的上界和下界，实现对表格的某一个属性进行范围查找的功能。例如：查找在属性 A 上，大于等于 50，小于等于 100 的所有行。当搜索结果包含的行数过多时，可以只返回整个搜索结果的一小部分，如 10 行等。

需要对存储表的范围搜索功能进行封装，封装为 API 函数，以供应用程序调用。

1.5 索引需求

选择一种数据结构，例如 B+ 树等，为存储表的某一个属性建立索引结构，以实现快速搜索，加快搜索效率。

为存储表建立的索引结构需要作为索引文件保存到文件系统中。在每次执行搜索功能的时候，需要先检查搜索的属性是否存在索引文件，再继续搜索。即，搜索功能执行时，需要查找是否有索引文件存在，若有，则使用该文件加速搜索；否则，为该属性建立索引，并且把索引结构存储到索引文件中。

提供 API 函数，实现该功能，以便方便应用程序调用。

1.6 并发需求

应用程序可以以多线程的方式，使用提供的上述 API，需要采取某种方式保证并发程序的正确性。要保证多线程环境下，表、索引结构、索引文件的一致性。

1.7 测试需求

对于整个系统的测试数据随机生成，另外测试用例需要覆盖系统的主要需求，即添加需求、搜索需求、索引需求和并发需求等。

1.8 其他需求

系统的基本功能要完成插入和搜索功能，删除和修改表中的数据等功能不做要求。

系统的实现需要使用 C 或 C++ 语言，并且需要应用现代程序设计思想。

第二章 总体设计

本章针对整个系统的总体设计思路进行介绍，从整体上分析了系统，对系统的架构进行了设计，并且针对系统中的核心模块进行了设计。

2.1 系统整体架构

如图2-1所示，是整个系统的架构，可以看到本系统最终目标的封装两个 API 供上层应用程序使用，一个是对于存储表数据的增加，增加一条数据在表的末尾，并且写入文件系统。另一个则是对存储表的数据的查询，给定查询范围和查询的列，对存储表进行查询，返回查询结果给调用的应用程序。

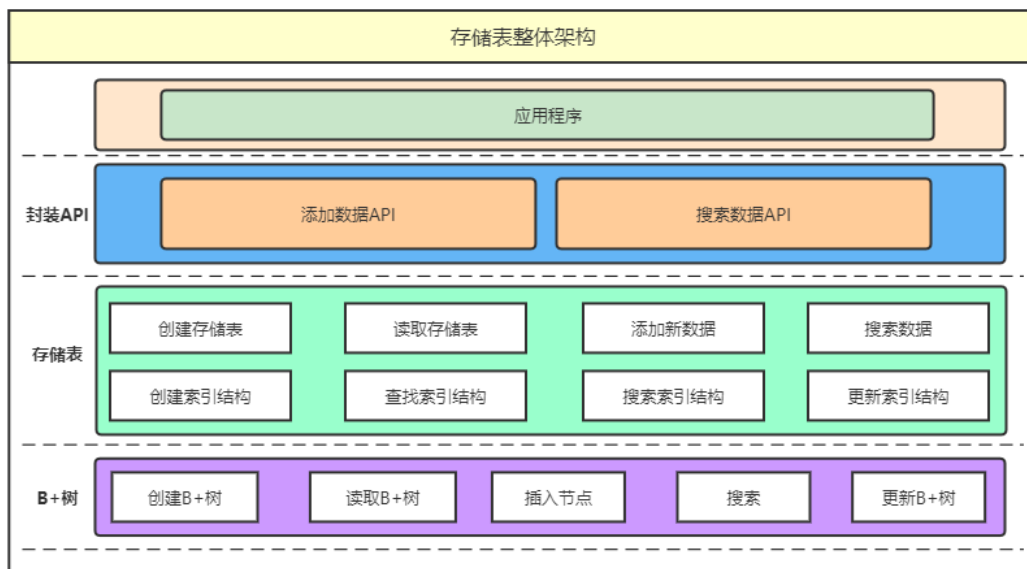


图 2-1 存储表系统架构图

本系统的核心部分在于对整个存储表的设计，本系统采用面向对象的思想进行构建，利用 Linux 提供的文件读写相关的 API，把表的数据存储到文件系统并从文件系统读取表的数据，并且对于索引结构也进行存储和读取。

对于非常大数据量的搜索，采用索引结构可以加快搜索的速度，常用的索引结构有 Hash、红黑树、B 树、B+ 树等，Hash 只适合等值查询，不适合范围查询。红黑树，是一种特化的平衡二叉树，数据量很大的时候，索引的体积也会很大，内存放不下的而从磁盘读取，树的层次太高的话，读取磁盘的次数就多了。B 树叶子节点和非叶子节点都保存数据，相同的数据量，B+ 树更矮壮，也就是说，相同

的数据量，B+ 树数据结构，查询磁盘的次数会更少。综合考虑，本系统选择 B+ 树作为索引结构。

最后，为了方便上层应用程序调用，对于增加数据和搜索数据进行封装，方便用户进行调用。

2.2 存储表核心部分设计

2.2.1 存储表添加数据功能设计

对于增加数据功能的设计，首先要满足程序并发的要求，防止不一致的读写的产生，那么就需要一个时刻只能有一个进程对整个存储表进行操作。在对存储表进行数据添加的时候，首先是随机产生一条新数据，之后把这条数据写入存储表的文件的末尾，同时假如之前已经由构建好的索引文件，那么就需要对所有之前已经构建好的索引文件进行更新。

2.2.2 存储表搜索数据功能设计

同样，对于存储表进行搜索也需要对存储表互斥访问，不能在搜索的同时对存储表进行更新，那样会导致不一致的数据产生。在对存储表进行范围搜索的时候，首先判断是否有索引结构文件，如果没有则直接进行搜索，并且为搜索的列创建索引文件。如果有的话，则读取索引文件中的 B+ 树结构，使用 B+ 树进行搜索。最终，返回搜索结果。为了判断是否真正使用了索引结构，在程序执行的过程中也记录了程序执行的时间。

2.3 B+ 树设计

如图2-2所示，是 B+ 树的一个结构图。在 B+ 树中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储 key 值信息，这样可以大大加大每个节点存储的 key 值数量，降低 B+ 树的高度。

通常在 B+ 树上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。因此可以对 B+ 树进行两种查找运算：一种是对主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

根据本系统的需求分析，在对 B+ 树设计的时候，需要能够对 B+ 树进行节点插入、节点更新、节点搜索等功能，其中最主要的是对 B+ 树进行搜索，而在执行范围搜索的时候会存在两种情况，一种是对特定值进行搜索，也就是搜索的上界等于搜索的下界；另外一种就是范围搜索。在设计 B+ 树的时候需要考虑这两种情

况，实现的时候分别实现。

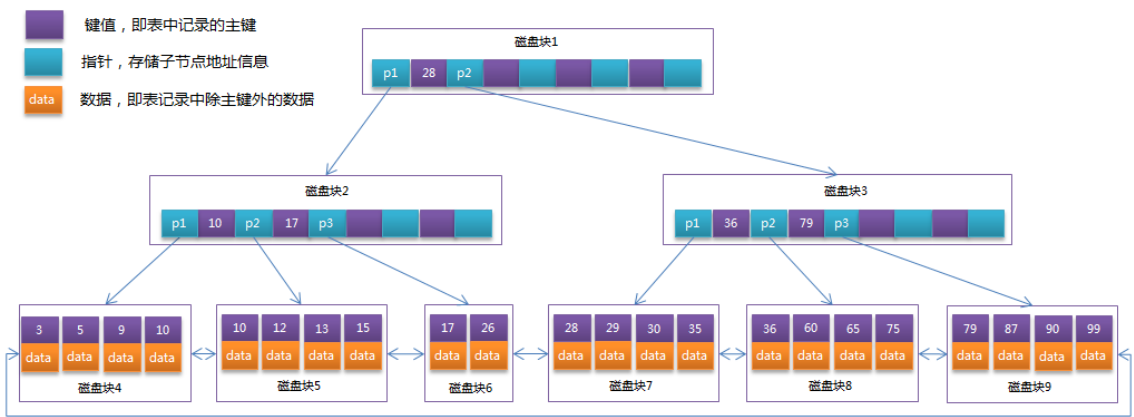


图 2-2 B+ 树结构

第三章 详细设计与实现

本章介绍了系统的详细设计，以及系统的核心部分的编码实现细节。

3.1 基本数据结构的设计与实现

如图3-1所示，是存储表中数据项的结构体的定义，该结构体中 itemID 是唯一标识每一条数据项的字段，而数组 items 则是这条数据具有的属性，预先设置了每个数据项的属性的大小为 100，即每条数据有一百个属性。

```
1 //数据项
2 typedef struct Item {
3     int64_t itemID;
4     int64_t items[ITEM_LENGTH];
5 } Item;
```

图 3-1 存储表中数据项定义

图3-2是对于 B+ 树中的节点定义，另外还定义了索引节点。而在 B+ 树节点中可以看到，定义了该节点的前驱节点、后继节点、孩子节点等构建 B+ 树必不可少的一系列数据项。

```
1 //index
2 typedef struct Index {
3     int64_t key;
4     int64_t value;
5 } Index;
6
7 #define M 5
8 //BPT
9 typedef struct BPTNode {
10     Index index[2 * M - 1];
11     struct BPTNode* children[2 * M];
12     struct BPTNode* prev;
13     struct BPTNode* next;
14     int children_num;
15     bool leaf;
16 } BPTNode;
```

图 3-2 B+ 树节点定义

3.2 B+ 树的设计与实现

图3-3是本系统中定义的 B+ 树的类的定义，从中可以看到，包括对于 B+ 树的创建、对 B+ 树的读取和写入、搜索等方法。

```

1  class BPlusTree {
2
3  public:
4
5      BPTNode* NewBPTNode();
6
7      BPTNode* CreateBPT();
8
9      Index NewIndex(const Item& record, int col);
10
11     //read B+tree
12     BPTNode *ReadBPT(int col);
13     BPTNode *ReadBPTNode(int &fd, BPTNode* &leaf_node_pre);
14
15     //write B+tree
16     void WriteBPT(BPTNode* root, int col);
17     void WriteBPTNode(int &fd, BPTNode *node);
18
19     //insert node
20     BPTNode* Insert(BPTNode* root, const Item& record, int
        col);
21     void NotFullInsert(BPTNode* node, const Index& index);
22
23     //split node
24     void Split(BPTNode* parent, int pos, BPTNode* child);
25
26     //find
27     void EqualFind(BPTNode* root, int value, int64_t *result
        , int &num);
28     void RangeFind(BPTNode* root, int lower, int upper,
        int64_t *result, int &num);
29 };

```

图 3-3 B+ 树类定义

由于本系统的核心功能之一，也就是搜索功能主要使用了 B+ 树创建的索引，下面针对 B+ 树种的两个搜索方法 EqualFind 和 RangeFind 的设计思路和核心代码进行分析。由之前介绍的 B+ 树的特点可以知道，B+ 树的所有叶子节点都是相互链接的，形成了一个链表，而且叶子节点的关键字是按照从小到大进行排序的。所以在进行等值搜索的时候，需要从根节点出发找到找到对应的叶子节点，完后再沿着叶子节点形成的链表进行查找，并且为了方便展示这里只要找到 10 条数据就终止查找。

图3-4是等值搜索函数 SearchValueEqual 的核心代码。从根节点开始，从索引节点数组的左端开始与需要查询的数据进行比较，直到查到的索引节点大于或者等于需要查询的数据，就停止比较，进入对应的子节点，一直重复整个过程，直到到达叶子节点。之后，一直沿着链表向右查询，保存与需要查询的数据相等的索引节点的 ID，假如遇到不相等的索引节点，那么就停止，返回查询到的数据。

```

1 void BPlusTree::EqualFind(BPTNode* root, int value, int64_t *
  result, int &num) {
2     num = 0;
3
4     if (root==NULL)
5         return;
6     BPTNode *node = root;
7
8     //find leaf node
9     while (node->leaf==false) {
10         int pos = 0;
11         while (pos < node->children_num && value > node->
            index[pos].value)
12             pos++;
13         node = node->children[pos];
14     }
15
16     //find value
17     while (node!=NULL) {
18
19         //only find 10 items
20         if (num == 10)
21             return ;
22
23         for (int i = 0; i < node->children_num && num < 10; i
            ++){
24             if (node->index[i].value > value)
25                 return;
26
27             //finding results
28             if (node->index[i].value == value)
29                 result[num++] = node->index[i].key;
30         }
31
32         node = node->next;
33     }
34 }

```

图 3-4 B+ 树中的等值搜索函数

图3-5 是范围查找函数 RangeFind 的核心代码。范围查找和等值查找的思路差不多，唯一不同的一点是从根节点向叶子节点搜索的时候，从左端和右端分别查

找叶节点，之后再从左端叶节点向右端叶子节点逼近即可，并且保存满足条件的数据，最后返回结果。

```

1 void BPlusTree::RangeFind(BPTNode* root, int lower, int upper
  , int64_t *result, int &num) {
2     num = 0;
3     BPTNode *node_lower= root;
4     BPTNode *node_upper = root;
5     //find leaf from left
6     while (!node_lower->leaf) {
7         int pos = 0;
8         while (pos < node_lower->children_num && lower >
9             node_lower->index[pos].value)
10             pos++;
11         node_lower = node_lower->children[pos];
12     }
13     //find leaf from right
14     while (!node_upper->leaf) {
15         int pos = node_upper->children_num;
16         while (pos > 0 && upper < node_upper->index[pos - 1].
17             value)
18             pos--;
19         node_upper = node_upper->children[pos];
20     }
21     while (node_lower != node_upper) {
22         if (node_lower == NULL) {
23             for (int i = 0; i < node_lower->children_num && num <
24                 10; i++) {
25                 if (node_lower->index[i].value >= lower)
26                     result[num++] = node_lower->index[i].key;
27             }
28             if (num == 10)
29                 return ;
30             node_lower = node_lower ->next;
31         }
32         for (int i = 0; i < node_lower->children_num && num < 10;
33             i++) {
34             if (lower <= node_lower->index[i].value && node_lower
35                 ->index[i].value <= upper)
36                 result[num++] = node_lower->index[i].key;
37         }
38     }
39 }

```

图 3-5 B+ 树中的范围搜索函数

3.3 存储表的设计与实现

整个存储表封装为一个类，存储表中相关的成员变量和成员函数类型如图3-6所示，其中定义了相关的变量和方法。需要注意的一点是，在该类中定义了一个 bigtable 指针类型的静态成员变量，这样做的目的是为了防止每一次对存储表

进行操作的时候，都重新创建一个对象，那样会严重浪费资源，所以声明了一个静态类型的变量，在初始化时候为其分配存储空间，而在 Get 方法会返回这个对象。

另外在 bigtable 类的定义中，只有 Get、Add、Find 是 public 的，也就是只有这三个方法可以供外部调用，其余的变量和方法都是私有的，只能内部调用。

```
1  class bigtable {
2  private:
3      bigtable();
4      ~bigtable();
5      int fd;
6      int64_t item_num;
7      Item* items;
8      static bigtable *btable;
9
10     //create an item
11     Item NewItem();
12
13     //print items
14     void Print(const Item &item);
15
16     //mutex for operating bigtable
17     pthread_mutex_t *mutex_op;
18     //mutex for creating bigtable
19     static pthread_mutex_t *mutex_create;
20     static pthread_mutex_t *mutex_init();
21
22     //B+tree
23     BPlusTree* bpt;
24     BPTNode* CreateBPT(int col);
25     void UpdateIndex(int col);
26     void CreateIndex(BPTNode* root, int col);
27     bool FindIndex(int col);
28
29 public:
30     //get bigtable
31     static bigtable *Get();
32
33     //add a item to bigtable
34     void Add();
35
36     //find items >=lower <=upper
37     void Find(int lower, int upper, int col);
38 };
```

图 3-6 存储表类定义

3.3.1 存储表添加数据功能详细设计与实现

图3-7是存储表添加数据功能的核心代码，在添加数据的时候首先要判断当前是否有进程在对存储表进行操作，也是判断能否进行加锁。只有能够加锁，才能进行接下来的操作。利用随机数随机生成一条数据，并把该数据写入存储表的文件的末尾，之后对之前建立的所有索引进行索引文件的更新操作，最后释放锁，表明操作结束。

```

1  //add a new item
2  void bigtable::Add() {
3      if (pthread_mutex_lock(mutex_op) != 0)
4          throw "lock, can not add";
5
6      Item item =NewItem();
7
8      if(write(fd, &item, ITEM_SIZE) == -1){
9          throw "write error";
10     }
11     cout << "insert success" << endl;
12     cout<<"new item:"<<endl;
13
14     Print(item);
15
16     cout<<"item_num: " <<item_num<<endl;
17
18     //update index
19     for (int col = 1; col < ITEM_LENGTH + 1; col++) {
20         if (FindIndex(col))
21             UpdateIndex(col);
22     }
23
24     pthread_mutex_unlock(mutex_op);
25 }

```

图 3-7 存储表添加函数

3.3.2 存储表搜索数据功能详细设计与实现

图3-8是存储表搜索功能的核心代码。同增加数据函数一样，搜索的时候也需要判断是否有进程在访问存储表，需要对存储表进行互斥访问。在搜索的时候需要先判断是否存在搜索的列的索引文件，存在的话就读取索引文件，否则就为该列创建索引文件。之后根据上界和下界判断执行等值搜索还是范围搜索，打印搜索的结果。最后操作完毕后，释放锁。并输出搜索操作执行的时间。

```

1 void bigtable::Find(int lower, int upper, int col) {
2     int64_t *find_result = new int64_t[10];
3     int num = 0;
4     clock_t start, finish;
5     start=clock();
6     pthread_mutex_lock(mutex_op);
7     BPTNode *root;
8     if (FindIndex(col)) {
9         cout << "find index ,reading...." << endl;
10        root = bpt->ReadBPT(col);
11    } else {
12        root = CreateBPT(col);
13        CreateIndex(root, col);
14    }
15    if (lower == upper) {
16        bpt->EqualFind(root, lower, find_result, num);
17    } else {
18        bpt->RangeFind(root, lower, upper, find_result, num);
19    }
20    cout << "search results:  " <<endl;
21    for (int i = 0; i < num; i++){
22        Print(items[find_result[i] - 1]);
23    }
24    finish=clock();
25    cout<<"time: "<<double(finish-start)/CLOCKS_PER_SEC<<"(s)
26        "<<endl;
27    pthread_mutex_unlock(mutex_op);
28 }

```

图 3-8 存储表搜索函数

3.4 API 封装及多线程的设计与实现

为了方便上层应用程序进行调用，本系统对存储表添加数据和存储表搜索数据功能分别进行了封装，为上层应用程序提供了相关 API 可以直接调用，另外通过调用 Linux 中提供的相关多线程 API 可以编写多线程程序，对存储表进行增加数据和搜索数据。

```

1 pthread_mutex_t *mutex_op;
2 static pthread_mutex_t *mutex_create;
3 static pthread_mutex_t *mutex_init();

```

图 3-9 互斥量的使用

如图3-9所示，是在 bigtable 类中声明的互斥量及互斥量初始化方法，由于本系统支持多线程并发运行，那么为了保证数据的一致性，一个时刻只有一个进程可以对存储表进行访问，那么就需要使用互斥量对表就是加锁和解锁。在 bigtable

中声明的两个互斥量，一个是用于对存储表的操作，包括增加、搜索；另一个则是用于存储表的创建，保证只有一个存储表被创建。

第四章 测试

本章对构建好的系统进行了测试，测试了系统的主要核心功能，检验系统是否正确完成了之前提出的所有需求。

4.1 测试环境

本系统的测试环境如下所示：

Linux 内核版本号: Linux version: 5.15.0-53-generic (builddd@lcy02-amd64-091)。

gcc 编译器版本: gcc version: 9.4.0。

Ubuntu 版本号: Ubuntu 9.4.0-1ubuntu1 20.04.1。

4.2 测试用例及结果

整个存储表中的数据都是随机生成，另外为了使得输出结果方便查看，生成的随机数的范围控制在一千以内的正整数。对系统的主要核心功能编写测试用例进行测试，由于测试用例都是人工编写，由于时间原因等原因，并没有采用专业的相关测试工具对系统进行详尽测试，所以对于本系统的测试并不完善，在提交作业后，还需要对系统进行详尽的测试，改进系统，加强系统的健壮性。

在执行本程序的时候，需要先对程序进行编译，由于本系统使用了 Linux 提供的线程相关的 API，所以在编译时需要加上 `-pthread` 静态链接 `pthread` 库。使用命令 `g++ *.cpp -pthread` 即可对源代码文件进行编译，之后执行 `./a.out` 即可运行程序。

4.2.1 添加功能测试

如图4-1所示是添加数据的测试代码，首先获取存储表对象实例，之后执行其添加数据的成员方法。由于是第一次获取存储表，那么就会初始化构建存储表，并把存储表相关数据写入文件系统中。

```
1    bigtable* btable=bigtable::Get();  
2    btable->Add();
```

图 4-1 添加功能测试代码

图4-2是程序执行结果，随机生成了一条新数据，并插入到了存储表中，可以看到，存储表中的数据项的个数发生了变化，增加了 1（初始时，存储表中有十万条数据）。

```

qiaoao@qiaoao-virtual-machine: ~/Linux/bigtable
qiaoao@qiaoao-virtual-machine:~/Linux/bigtable$ g++ *.cpp -pthread
qiaoao@qiaoao-virtual-machine:~/Linux/bigtable$ ./a.out
item_num: 100000
init bigtable success!
insert success
new item:
    734    676    340    329    105    120    130    697
item_num: 100001
qiaoao@qiaoao-virtual-machine:~/Linux/bigtable$

```

图 4-2 添加数据测试结果

4.2.2 搜索功能测试

图4-4是搜索功能测试代码，在搜索时首先也是获取表格对象，之后调用存储表对象中的成员函数 Find 进行范围搜索，该方法的参数是搜索的上界、搜索的下界和搜索的列的编号，此处搜索的是第三列，值大于等于 10，小于等于 500 的数据。

```

1    bigtable* btable=bigtable::Get();
2    btable->Find(10,500,3);

```

图 4-3 搜索功能测试代码

搜索结果如图4-4所示，可以看到由于是第一次对第三列进行搜索，并没有找到索引结构，那么就会为第三列创建索引结构，并把索引结构写入文件系统中，返回了搜索结果和程序执行时间。

```

qiaoao@qiaoao-virtual-machine: ~/Linux/bigtable
qiaoao@qiaoao-virtual-machine:~/Linux/bigtable$ g++ *.cpp -pthread
qiaoao@qiaoao-virtual-machine:~/Linux/bigtable$ ./a.out
item_num: 100001
init bigtable success!
finding.....
not find index,creating.....
search results:
    424    418    500    744    690    318    143    501
    47     327    500    240    146    457    661    57
    461    551    500    517    603    981    246    659
time: 0.089857(s)

```

图 4-4 搜索功能测试结果

由于是第一次执行在第三列的搜索，所以存储表会为该列建立索引结构，并且把索引结构写入文件系统中。从图4-5可以看出，生成了索引文件 INDEX6，另外，图中的 bigtable 文件是保存存储表数据的文件。

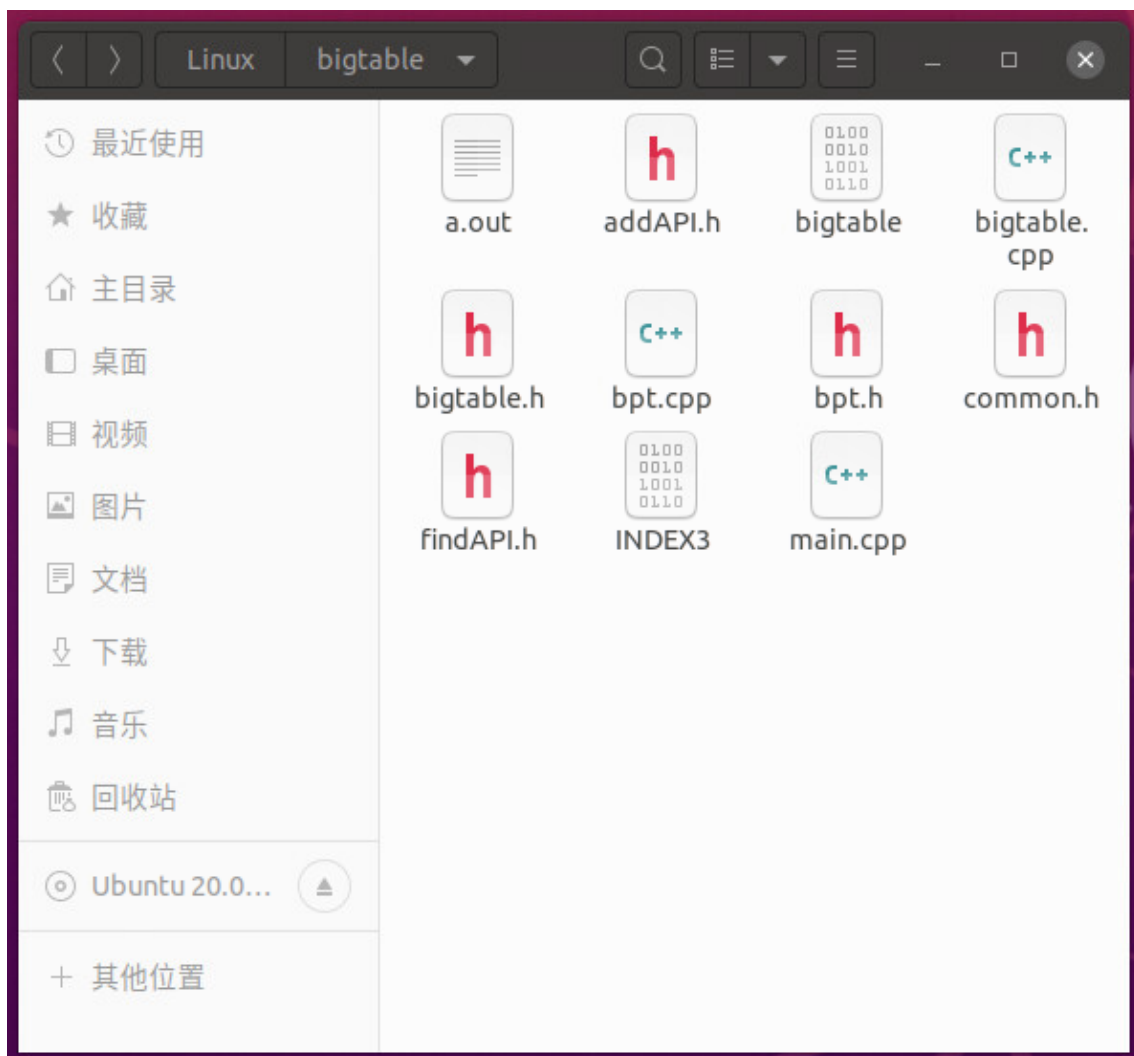


图 4-5 生成索引文件

4.2.3 索引功能测试

在第一次调用搜索功能后，重新再次调用搜索功能，还执行图4-4中相同的搜索代码，由于之前的第一次搜索已经为第三列建立了索引，所以正常情况下在第三列再次进行搜索会利用生成的索引文件加快搜索速度。

测试结果如图4-6所示。从图可以看出，一方面，两次执行的搜索结果是相同的，证明了搜索函数功能的正确性，也证明了存储表写入文件系统和从文件系统读取数据的正确性。另一方面，由于第一次搜索为第三列建立了索引文件，所以第二次相同的搜索通过索引加快了搜索效率，可以看到，第二次程序执行时间相较于第一次大大提高，说明第二次搜索是通过索引文件进行范围搜索，说明整个索引功能是正确的。

```

qiaobao@qiaobao-virtual-machine: ~/Linux/bigtable
qiaobao@qiaobao-virtual-machine:~/Linux/bigtable$ g++ *.cpp -pthread
qiaobao@qiaobao-virtual-machine:~/Linux/bigtable$ ./a.out
item_num: 100001
init bigtable success!
finding.....
not find index,creating....
search results:
    424    418    500    744    690    318    143    501
    47     327    500    240    146    457    661    57
    461    551    500    517    603    981    246    659
time: 0.089857(s)
qiaobao@qiaobao-virtual-machine:~/Linux/bigtable$ ./a.out
item_num: 100001
init bigtable success!
finding.....
find index ,reading....
search results:
    424    418    500    744    690    318    143    501
    47     327    500    240    146    457    661    57
    461    551    500    517    603    981    246    659
time: 0.025997(s)
qiaobao@qiaobao-virtual-machine:~/Linux/bigtable$

```

图 4-6 索引功能测试结果

4.2.4 并发功能测试

图4-7是并发功能测试的测试代码，在测试函数中创建多个线程调用封装好的API，并且查看结果，来判断多线程程序是否正确执行，是否会发生数据不一致性的情况。

```

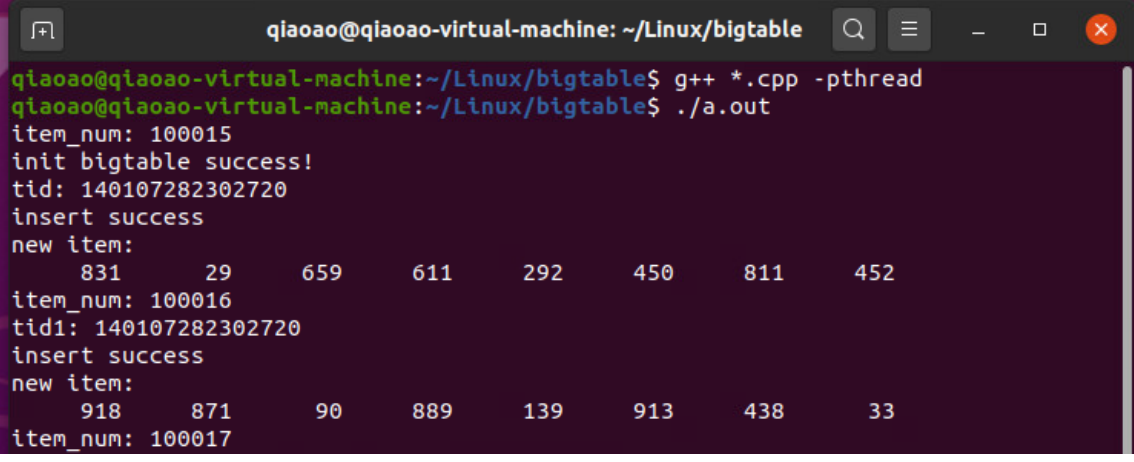
1      pthread_t tid;
2
3      int res=pthread_create(&tid,NULL,&AddAPI::RunAdd,NULL);
4      cout<<"tid:"<<tid<<endl;
5      pthread_join(tid,NULL);
6      pthread_t tid1;
7
8      int res1=pthread_create(&tid1,NULL,&AddAPI::RunAdd,NULL)
9      ;
10     cout<<"tid1:"<<tid1<<endl;
11     pthread_join(tid1,NULL);

```

图 4-7 并发功能测试代码

图4-8是并发功能测试结果，从测试结果中可以看到，成功创建了两个增加数据的线程，对存储表数据进行新增，由于互斥量的使用，多线程程序并没有造成数据的不一致性产生。两个线程都正确输出了线程 ID，返回了正确结果。该测试

结果也反映了程序中互斥量使用正确。



```
qiaobao@qiaobao-virtual-machine: ~/Linux/bigtable
qiaobao@qiaobao-virtual-machine:~/Linux/bigtable$ g++ *.cpp -pthread
qiaobao@qiaobao-virtual-machine:~/Linux/bigtable$ ./a.out
item_num: 100015
init bigtable success!
tid: 140107282302720
insert success
new item:
    831      29    659    611    292    450    811    452
item_num: 100016
tid1: 140107282302720
insert success
new item:
    918     871     90    889    139    913    438     33
item_num: 100017
```

图 4-8 并发功能测试结果

第五章 总结

通过本次大作业中学习到了很多知识，由于之前的学习过程中对于 C 和 C++ 语言并没有很熟练掌握，另外，在日常生活中编码也很少使用 C 和 C++，所以在本次大作业的过程中遇到了不少的麻烦，包括如何编写类对相关数据结构及方法进行封装、如何使用互斥量对临界资源互斥访问，以及最重要的对 Linux 中的文件读写、多线程等 API 等的使用。

在本次作业遇到的一个最大的困难，是对于 B+ 树的编码，对于 B+ 树这种数据结构理解起来不是很困难的，但是在编码实现过程中遇到了不少的麻烦，这一部分也参考了网络上其他人的代码。从这一点也可以看出，自己的编码能力有待提高，需要在今后多加练习。

整体来说，通过做这次大作业，自己收获颇丰，也从中发现自己有很大的不足，希望在之后的研究生生活中加紧学习，不断进步！