电 子 科 技 大 学

作 业 报 告

学生姓名：张浩　　学 号：202021080306　　指导教师：刘杰彦

学生 E-mail：986136244@qq.com

Linux 下表的实现与应用

第一章 需求分析

1. 总体要求：

   在 Linux 环境下，采用 C 或 C++，存储一张表，然后能对该表进行查询、添加等操作。上述功能以 API 的形式提供给应用使用。

2. 存储要求：

   利用文件操作 API，在文件系统中存储一张表。该表有 100 个属性，每个属性都是 8 字节大小。需要支持的最大行数为一百万行（可看作支持行数没有上限限制）。

3. 添加要求：

   提供 API 函数，实现向表格添加一行的功能（添加到表格的末尾）。

4. 搜索要求：

   提供 API 函数，实现对表格某一个属性进行范围查找或精确查找的功能。

   例如：查找在属性 A 上，大于等于 50，小于等于 100 的所有行，当上下限相等时，即为精确查找。

   用户可以指定在哪一个属性上进行搜索。当搜索结果过多时，可以返回一小部分，如 10 行。

5. 索引要求：

提供 API 函数，为表格里的某一个属性建立索引结构，以实现快速搜索。

自行选择使用哪种数据结构，建立索引结构，比如 B+树等。

建立的索引结构，需要保存到一个文件中（索引文件），下次重启应用程序，并执行搜索任务时，应先检查是否已为相应属性建立了索引结构。即，搜索功能实现时，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。

6. 并发要求：

   应用程序可以以多线程的方式，使用上述 API。

   要保证多线程环境下，表、索引结构、索引文件的一致性（考虑互斥的要求）。

7. 测试要求：

   表中的数据随机生成。

   测试用例要覆盖主要的需求。

第二章 总体设计

1. 程序总体架构

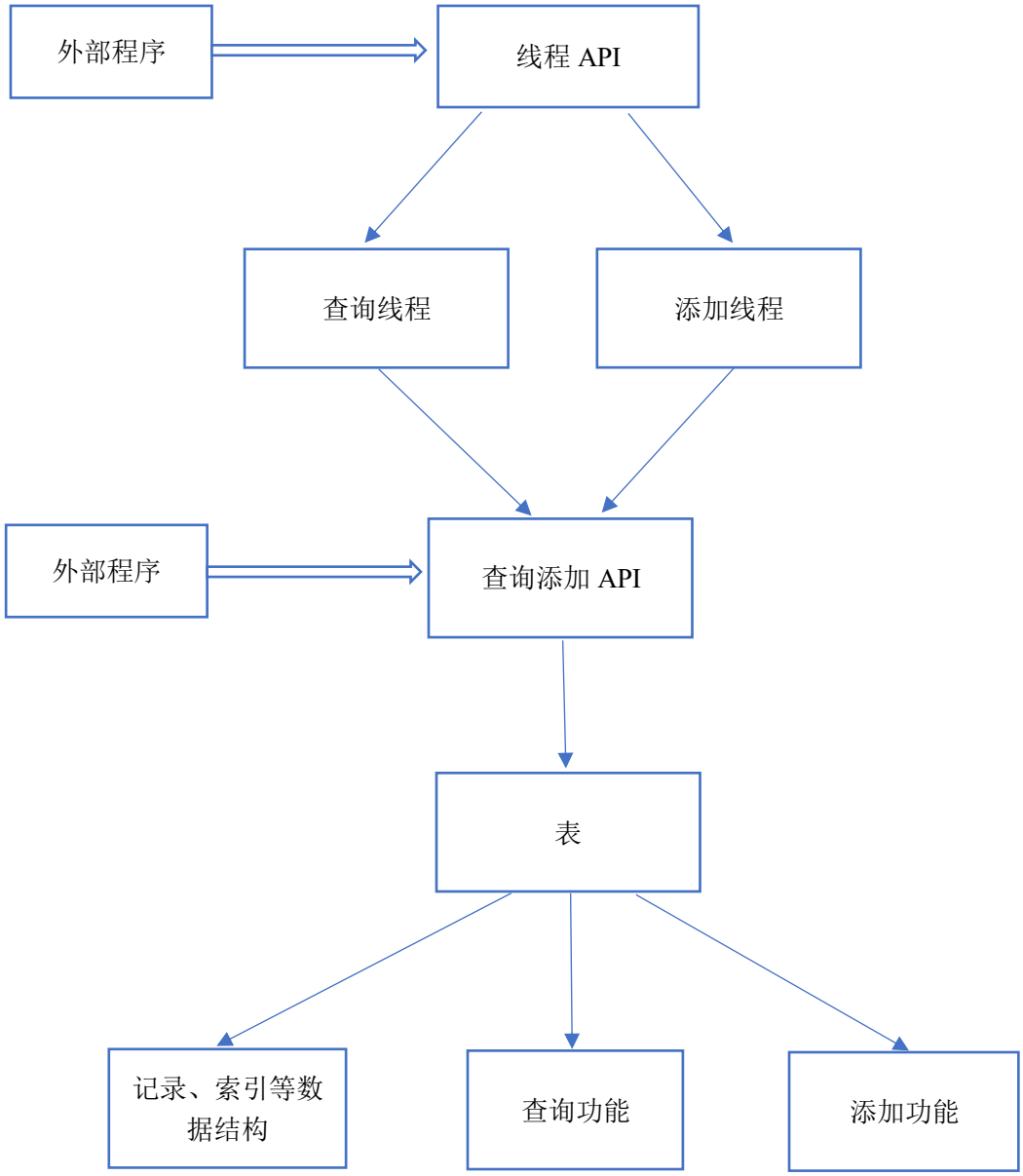

图 2-1 程序总体架构

图 2-1 所示是程序的总体架构。可以看到，在本次实验中，首先是完善表的基本结构和功能。其中记录的存储与读取，索引结构的构造，索引文件的存储与读取，是最先完善的。接着实现表的查询和添加功能。除此之外，为了使用的方便性，本实验封装了两个 API 供外部程序使用。第一个是将表的查询和添加进行封装，

外部程序可直接调用；考虑到多线程的环境，又封装了创建线程的 API，调用该 API 可以创建用于查询和添加的线程，实现并发，提高了效率。

2. 关键流程分析

在本次实验中，为了提高查询速度，查询功能是基于索引结构的，而添加记录会引起索引结构的更新，因此，索引结构的实现是关键。本次实验选择了 B+树作为表的索引结构，执行查询功能时，会先检查索引文件是否存在，若不存在则创建索引结构进行查询，并将索引结构保存到索引文件中，供下次查询使用；若存在索引文件，则从中读取出索引结构进行查询。

为了实现并发功能，本实验封装了创建线程的 API，用于创建查询线程和添加线程，通过调用操作表格的 API，执行相应的功能。由于在多线程环境下，表格属于共享资源，因此必须考虑到互斥的要求，保证索引文件和表格中记录的一致性。

第三章 详细设计与实现

1. 基本的数据结构

在本实验中，每条记录有 100 个属性，为了区分开每条记录，将其在文件中的序号作为主键，如下所示：

```
17  typedef struct Record {
18      int64_t primary_key; //主键，即该记录的序号
19      int64_t record_array[RECORD_LENGTH];//属性值
20  } Record;
```

构建某一属性的索引结构时，需要把所有记录的该属性值作为关键字构建 B+树，并将记录的序号一同保存，便于寻找该条记录。因此，B+树中的关键字定义如下：

```
22  //索引节点结构
23  typedef struct IndexNode {
24      int64_t primary_key;//主键
25      int64_t value;//属性值
26  } IndexNode;
```

为了提高查询效率，本实验选择构建四阶 B+树作为记录属性的索引，树节点定义如下：

```
28  //B+树节点结构
29  typedef struct BPlusTreeNode {
30      IndexNode index_nodes[2 * M - 1];//关键字，即为索引节点
31      struct BPlusTreeNode* childs[2 * M]; //子节点数组
32      int num;//子节点数
33      bool is_leaf;
34      struct BPlusTreeNode* prev;
35      struct BPlusTreeNode* next;
36  } BPlusTreeNode;
```

2. B+树的实现

```
class BPlusTree {
public:
    //创建一个节点对象
    BPlusTreeNode* BPlusTreeNode_new();

    //创建B+树
    BPlusTreeNode* BPlusTree_create();

    //创建一个索引节点
    IndexNode IndexNode_new(const Record& record, int col);

    //插入未满的节点中
    void BPlusTree_Insert_NotFull(BPlusTreeNode* node, const IndexNode& indexNode);

    //节点分裂
    int BPlusTree_Split_Child(BPlusTreeNode* parent, int pos, BPlusTreeNode* child);

    //插入节点s
    BPlusTreeNode* BPlusTree_Insert(BPlusTreeNode* root, const Record& record, int col);

    //搜索特定属性值
    void SearchValueEqual(BPlusTreeNode* root, int value, int64_t *result, int &num);

    //搜索属性值在[left,right]范围内的记录
    void SearchValueRange(BPlusTreeNode* root, int left, int right, int64_t *result, int &num);

    //保存B+树
    void WriteBPlusTree(BPlusTreeNode* root, int col);

    //将B+树节点写入文件
    void WriteBPlusTreeNode(int &fd, BPlusTreeNode *node);

    //读取B+树
    BPlusTreeNode *ReadBPlusTree(int col);

    //读取B+树节点
    BPlusTreeNode *ReadBPlusTreeNode(int &fd,BPlusTreeNode* &leaf_node_pre);
};
```

上面的代码为 B+树构建了一个类，里面是各种成员函数，包括构建一棵 B+树所需的一些基本函数，以及实现查询、插入、保存、读取的函数，下面依次展示它们的实现。

B+树的插入：

```
104  //插入节点
105  BPlusTreeNode* BPlusTree::BPlusTree_Insert(BPlusTreeNode* root, const Record& record, int col) {
106      IndexNode indexNode = IndexNode_new(record, col);
107      if (!root) {
108          return NULL;
109      }
110      if (root->num == 2 * M - 1) {
111          BPlusTreeNode* node = BPlusTreeNode_new();
112          if (!node) {
113              return NULL;
114          }
115          node->is_leaf = false;
116          node->childs[0] = root;
117          BPlusTree_Split_Child(node, 0, root);
118          BPlusTree_Insert_NotFull(node, indexNode);
119          return node;
120      } else {
121          BPlusTree_Insert_NotFull(root, indexNode);
122          return root;
123      }
124  }
```

如上述代码所示，插入一条记录时，我们首先为其构建关键字结构。由于插入一个关键字有时会引起 B+树节点的分裂，所以我们在插入之前首先应检查当前节点关键字是否已满，如果已满，应先调用 BPlusTree_Split_Child 函数将其分裂成两个节点，然后再进行插入，以保证每次插入时当前节点都处于非满状态，此时调用 BPlusTree_Insert_NotFull 函数，插入该未满节点中。

节点分裂的实现：

```
//节点分裂
int BPlusTree::BPlusTree_Split_Child(BPlusTreeNode* parent, int pos, BPlusTreeNode* child) {
    BPlusTreeNode* new_child = BPlusTreeNode_new();
    if (!new_child)
        return -1;
    new_child->is_leaf = child->is_leaf;
    new_child->num = M - 1;
    for (int i = 0; i < M - 1; i++)
        new_child->index_nodes[i] = child->index_nodes[i + M];
    if (!new_child->is_leaf) {
        for (int i = 0; i < M; i++)
            new_child->childs[i] = child->childs[i + M];
    }
    child->num = M - 1;
    if (child->is_leaf)
        child->num++;//如果是叶节点，保留中间的关键码
    for (int i = parent->num; i > pos; i--)
        parent->childs[i + 1] = parent->childs[i];
    parent->childs[pos + 1] = new_child;
    for (int i = parent->num - 1; i >= pos; i--)
        parent->index_nodes[i + 1] = parent->index_nodes[i];
    parent->index_nodes[pos] = child->index_nodes[M - 1];
    parent->num++;

    //叶节点情况，需要更新指针
    if (child->is_leaf) {
        new_child->next = child->next;
        child->next->prev = new_child;
        child->next = new_child;
        new_child->prev = child;
    }
    return 1;
}
```

上述函数有三个参数，第一个是待分裂节点的父节点指针，第二个是分裂的位置，第三个是待分裂节点的指针。首先创建一个新的节点，将待分裂节点的后 M-1 个关键码和后 M 个孩子指针放入新建的节点，更新节点的成员值，并将待分裂节点最中间的关键码放入父节点，并调整父节点的孩子指针，使其指向正确的子节点。上述代码保证了每次分裂节点时，待分裂节点的父节点都处于非满状态，因此不会引起二次分裂。

插入非满节点：

```
//插入一个未满的节点中
void BPlusTree::BPlusTree_Insert_NotFull(BPlusTreeNode * node, const IndexNode& indexNode) {
    if (node->is_leaf) {
        int pos = node->num;
        while (pos >= 1 && indexNode.value < node->index_nodes[pos - 1].value) {
            node->index_nodes[pos] = node->index_nodes[pos - 1];
            pos--;
        }
        node->index_nodes[pos] = indexNode;
        node->num++;
    } else {
        int pos = node->num;
        while (pos > 0 && indexNode.value < node->index_nodes[pos - 1].value) {
            pos--;
        }

        if (2 * M - 1 == node->childs[pos]->num) {
            BPlusTree_Split_Child(node, pos, node->childs[pos]);
            if (indexNode.value > node->index_nodes[pos].value)
                pos++;
        }
        BPlusTree_Insert_NotFull(node->childs[pos], indexNode);
    }
}
```

需要判断该节点是否为叶节点，若是则直接插入相应位置，若不是，则根据节点中的关键字判断应插入的子节点，同样的，为了避免二次分裂，我们在插入子节点时，都应先判断子节点是否已满，然后递归地调用此函数，最终将关键字插入叶节点中。

B+树的插入采取的是先检查再插入的策略，而不是先插入再判断的策略，这样做的好处是避免了二次分裂，无需考虑子节点分裂后引起父节点分裂的情形。

B+树的搜索：

根据设计需求，既要实现精确查找，也要实现范围查找，编写相应函数分别实现这两个功能。由上述代码可知，构建出来的 B+树的所有叶节点从左到右形成了一个链表，且叶节点中的关键字从小到大排列，因此查询时先找到对应的叶节点，然后再顺着叶节点的 next 指针寻找。

精确查找：

```
/** 搜索特定属性值
  * root B+树根节点
  * value 属性值
  * result 结果数组 存放记录的主键
  * num 结果数
  */
void BPlusTree::SearchValueEqual(BPlusTreeNode* root, int value, int64_t *result, int &num) {
    num = 0;
    if (!root)
        return;
    BPlusTreeNode *node = root;
    while (!node->is_leaf) { //不是叶节点，向下搜索
        int pos = 0;
        while (pos < node->num && value > node->index_nodes[pos].value)
            pos++;
        node = node->childs[pos];
    }
    //到达叶节点，顺着next指针往前搜索
    while (node) {
        for (int i = 0; i < node->num && num < MAX_RESULT_NUM; i++) {
            if (node->index_nodes[i].value > value)
                return;
            if (node->index_nodes[i].value == value)
                result[num++] = node->index_nodes[i].primary_key;
        }
        if (num == MAX_RESULT_NUM)
            return ;
        node = node->next;
    }
}
```

精确查找的思路是：从根节点开始，从关键字数组的左端开始跟待查的关键字比较，遇到大于或者等于的关键字便停止，进入对应的子节点，重复该过程，直到到达叶节点，这样做是为了保证能够到达待查关键字可能出现的第一个叶节点，不会出现遗漏的情况。到达叶节点后，一直向右查找，保存与待查属性值相等的关键字的序号，遇到更大的属性值便停止，返回记录的序号数组。

范围查找：

```
/**搜索属性值在[left,right]范围内的记录
 */
void BPlusTree::SearchValueRange(BPlusTreeNode* root, int left, int right, int64_t *result, int &num) {
    num = 0;

    BPlusTreeNode *node_left = root;
    BPlusTreeNode *node_right = root;

    //往下搜索，到达两个端点所在的叶节点
    while (!node_left->is_leaf) {
        int pos = 0;
        while (pos < node_left->num && left > node_left->index_nodes[pos].value)
            pos++;
        node_left = node_left->childs[pos];
    }
    while (!node_right->is_leaf) {
        int pos = node_right->num;
        while (pos > 0 && right < node_right->index_nodes[pos - 1].value)
            pos--;
        node_right = node_right->childs[pos];
    }

    //移动node_left指针直到node_right
    while (node_left != node_right) {
        if(node_left==NULL)
            std::cout<<"error";
        for (int i = 0; i < node_left->num && num < MAX_RESULT_NUM; i++) {
            if (node_left->index_nodes[i].value >= left)
                result[num++] = node_left->index_nodes[i].primary_key;
        }
        if (num == MAX_RESULT_NUM)
            return ;
        node_left = node_left ->next;
    }

    //node_left和node_right相遇
    for (int i = 0; i < node_left->num && num < MAX_RESULT_NUM; i++) {
        if (left <= node_left->index_nodes[i].value && node_left->index_nodes[i].value <= right)
            result[num++] = node_left->index_nodes[i].primary_key;
    }
}
```

与精确查找类似，查找某个范围时，从左端和右端分别查找到叶节点，然后再从左端叶节点向右边逼近，保存搜索到的所有符合条件的记录。

B+树的保存：

```
//保存为第col列属性创建的B+树，创建或更新索引时调用此函数
void BPlusTree::WriteBPlusTree(BPlusTreeNode *root, int col) {
    char index_file_path[20];
    sprintf(index_file_path, "%s%d", INDEX_FILE_PATH, col);
    //删除原来的索引文件
    if (access(index_file_path, F_OK) == 0) {
        if (remove(index_file_path ) == -1) {
            std::cout << "删除失败";
            throw "In BPlusTree::WriteBPlusTree(),remove error";
        }
    }
    int fd = open(index_file_path, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    if (fd == -1)
        throw "In BPlusTree::WriteBPlusTree(),open error";
    WriteBPlusTreeNode(fd, root);
    close(fd);
}
```

利用文件操作 API，将 B+树的每个节点按照深度优先的顺序写到文件中。

B+树的读取：

```
229  //读取B+树
230  BPlusTreeNode* BPlusTree::ReadBPlusTree(int col) {
231      BPlusTreeNode* leaf_node_pre = NULL; //保存前一个叶节点
232      char index_file_path[20];
233      sprintf(index_file_path, "%s%d", INDEX_FILE_PATH, col);
234      if (access(index_file_path, F_OK) == -1)
235          return NULL;
236      int fd = open(index_file_path, O_RDONLY, S_IRUSR | S_IWUSR);
237      if (fd == -1)
238          return NULL;
239      BPlusTreeNode *node = ReadBPlusTreeNode(fd, leaf_node_pre);
240      close(fd);
241      return node;
242  }
```

B+树的读取和保存类似，按照深度优先的顺序读取，不过需要注意为了恢复之前叶节点形成的链表，需要额外引入一个 leaf_node_pre 指针变量，指向读取到的上一个叶节点，再次读取到叶节点时，将上一个叶节点的 next 指针指向新的叶节点，并将 leaf_node_pre 指向新的叶节点。读取完毕之后，所有的叶节点形成了一个链表，方便以后的查询操作。

3. 表的完善

下面是 table.h 的内容：

```
class table {

public:
    static table *GetTable();//获取table对象指针
    void InsertRecord();//添加记录
    void SearchRecord(int left, int right, int col); //查询第col列属性值位于区间(left,right)的所有记录


private:
    table();
    ~table();
    Record CreateRecord(); //创建一条随机记录
    bool AppendRecord(const Record &record); //保存一条记录
    BPlusTreeNode* CreateBPlusTree(int col);//为第col列的属性创建B+树
    void UpdateIndexFile(int col);//更新索引文件
    void CreateIndexFile(BPlusTreeNode* root, int col);
    void DisplayRecord(const Record &record);//显示记录
    bool Is_Index_File_Exists(int col);//判断索引文件是否存在
    void InitializeTable();//初始化表格
    static pthread_mutex_t *InitializeMutex();//初始化



private:
    int m_Fd;//文件描述符
    int64_t record_num;//表中现有记录数量
    Record* records;//存储记录

    BPlusTree* tree;//B+树
    pthread_mutex_t *m_pMutexForOperatingTable;//互斥访问表


    //用于table实例的创建，保证只创建一个table实例
    static pthread_mutex_t *m_pMutexForCreatingTable;

    static table *m_table;


};


#endif // TABLE_H
```

如上述代码所示，该头文件中声明了对表进行操作的所有函数，其中InsertRecord()和 SearchRecord()是供外部程序使用的接口，其他则是内部使用的函数，下面依次分析其作用和实现。

表的初始化：

```
table::table() {
    m_Fd = open(RECORD_FILE, O_RDWR | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR);
    if (m_Fd == -1)
        throw "In table::table(),open error";

    //读取文件中已有的记录数量
    records = new Record[MAX_RECORD_NUM];
    //判断表格是否为空
    if (lseek(m_Fd, 0, SEEK_END) == 0) {
        record_num = 0;
        InitializeTable();
    } else {
        record_num = lseek(m_Fd, 0, SEEK_END) / RECORD_SIZE_BYTE;
        lseek(m_Fd, 0, SEEK_SET);
        read(m_Fd, records, record_num * RECORD_SIZE_BYTE);
    }
    tree = new BPlusTree;
    srand((unsigned) time(NULL));//随机数种子

    //初始化mutex
    m_pMutexForOperatingTable = new pthread_mutex_t;
    if (pthread_mutex_init(m_pMutexForOperatingTable, 0) != 0 ) {
        delete m_pMutexForOperatingTable;
        close(m_Fd);
        throw "In table::table(),pthread_mutex_init error";
    }
}
```

上述代码是 table 类的构造函数，涉及一些基本成员的初始化，例如文件描述符的初始化，把它放在构造函数里的原因是：在创建表实例时就把表文件打开，这样就不需要每次插入记录时都进行打开文件的操作，节省了时间，提高了效率。接着是读取表中所有记录，如果是第一次使用表，,即记录数为 0，便调用 InitializeTable()函数为表创建 10000 条记录，方便实验。然后是 tree 的初始化，用于对索引的操作。

表实例的获取：

```
62  //获取table实例
63  table *table::GetTable() {
64
65      /**
66      只允许构建一个table实例，因为对表的操作是互斥的
67      **/
68
69      if (m_table != 0) {
70          return m_table;
71      }
72      if (pthread_mutex_lock(m_pMutexForCreatingTable) != 0) {
73          return 0;
74      }
75      if (m_table == 0) {
76          try {
77              m_table = new table;
78          } catch (const char *) {
79              pthread_mutex_unlock(m_pMutexForCreatingTable);
80              return 0;
81          }
82      }
83      if (pthread_mutex_unlock(m_pMutexForCreatingTable) != 0) {
84          return 0;
85      }
86      return m_table;
87  }
```

如上述代码所示，获取表实例的唯一方法是调用 table 类的静态函数 GetTable()，将表实例的地址保存在静态变量 m_table 中，每次外部程序要获取 table 实例时，GetTable()函数都会先检查 m_table，若空则创建，非空则返回。同时注意到将 table 类的构造函数设为私有，只可供类内部使用。这样做的原因是：由于设计到对表的操作时，都需要先创建一个表实例，再调用其中的方法，如此一来，每使用一次表操作，都会创建一个表实例，资源消耗十分严重，因此我们考虑在全局公用一个 table 实例。通过静态函数 GetTable()获取。

互斥量的使用：

在多线程环境下，表属于共享资源，为了避免错误，需要引入互斥量，同时在使用 GetTable()函数获取表实例时，也需要引入创建表的互斥量，防止两个

线程同时创建表实例。

```
45      pthread_mutex_t *m_pMutexForOperatingTable;//互斥访问表
46
47
48      //用于table实例的创建，保证只创建一个table实例
49      static pthread_mutex_t *m_pMutexForCreatingTable;
50
```

查询插入功能的封装：

为了外部程序的使用，我们封装了如下两个函数实现查询、插入的功能。

```
137 //插入记录
138 void table::InsertRecord() {
139     //P(mutex)
140     if (pthread_mutex_lock(m_pMutexForOperatingTable) != 0)
141         throw "In table::InsertRecord(),lock error";
142     Record record = CreateRecord();
143     //DisplayRecord(records[record_num-1]);
144     if (!AppendRecord(record))
145         throw "In table::InsertRecord(),insert error";
146     std::cout << "已成功添加一条记录: " << std::endl;
147     DisplayRecord(record);
148     //更新索引
149     for (int col = 1; col < RECORD_LENGTH + 1; col++) {
150         if (Is_Index_File_Exists(col))
151             UpdateIndexFile(col);
152     }
153     //V(mutex)
154     pthread_mutex_unlock(m_pMutexForOperatingTable);
155 }
```

插入记录时，为了实现互斥，使用 pthread_mutex_lock 对互斥量上锁，然后随机生成一条记录，插入记录文件，同时更新索引文件。

```cpp
//搜索记录
void table::SearchRecord(int left, int right, int col) {
    std::cout << "正在搜索第" << col << "列，范围: [" << left << "," << right << "]" << std::endl;
    //P(mutex)
    pthread_mutex_lock(m_pMutexForOperatingTable);

    //存放搜索结果
    int64_t *result = new int64_t[MAX_RESULT_NUM];
    int num = 0;
    BPlusTreeNode *root;//存放B+树
    //首先判断是否有索引文件
    if (Is_Index_File_Exists(col)) {
        std::cout << "已查找到索引文件，正在读取..." << std::endl;
        root = tree->ReadBPlusTree(col);
        if (!root) {
            std::cout << "读取索引文件失败" << std::endl;
            return ;
        }

    } else {
        std::cout << "未查找到索引文件，正在创建..." << std::endl;
        root = CreateBPlusTree(col);
        if (!root) {
            std::cout << "创建索引文件失败";
            return;
        }
        //创建索引文件
        CreateIndexFile(root, col);
        std::cout << "已为第" << col << "列创建索引文件" << std::endl;
    }

    if (left == right) {
        tree->SearchValueEqual(root, left, result, num);
    } else {
        tree->SearchValueRange(root, left, right, result, num);
    }
    std::cout << "搜索结果为: " << std::endl;
    for (int i = 0; i < num; i++)
        DisplayRecord(records[result[i] - 1]);
    //V(mutex)
    pthread_mutex_unlock(m_pMutexForOperatingTable);
}
```

搜索记录时，先对互斥量上锁，然后先查找对应属性是否有索引文件，没有则创建，然后查找；有索引文件便直接读取，然后查找相应的记录。

4. 线程执行体的封装

本实验涉及到两种线程——查询线程和插入线程，由于执行的业务逻辑不同，考虑从基于接口的程序设计思想对线程的创建进行封装。

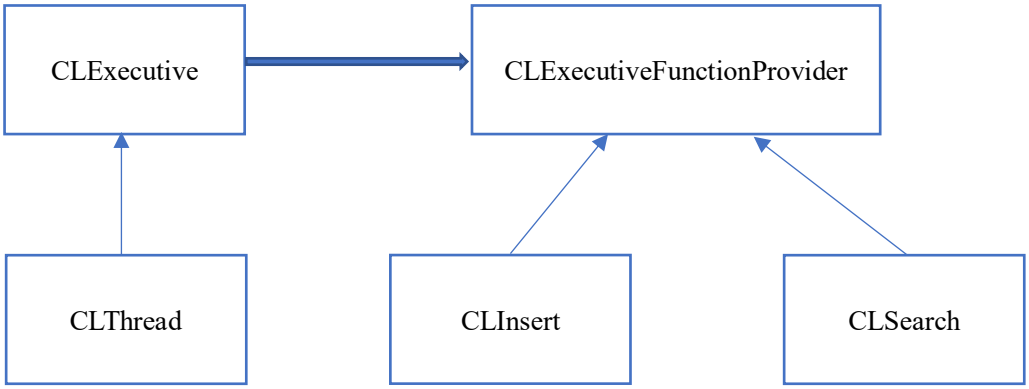

图 3-1 基于接口程序设计思想的类图

由图 3-1 的类图所示，CLExecutive 和 CLExecutiveFunctionProvider 分别是线程创建和业务逻辑的接口类，CLThread 是 CLExecutive 类的派生类，是线程创建的具体实现，CLInsert 和 CLSearch 是具体的业务逻辑实现。

```
#ifndef CLEXECUTIVE_H
#define CLEXECUTIVE_H

#include "CLExecutiveFunctionProvider.h"

//执行体的接口类
class CLExecutive{
public:
    explicit CLExecutive(CLExecutiveFunctionProvider* pExecutiveFunctionProvider){
        m_pExecutiveFunctionProvider=pExecutiveFunctionProvider;
    }
    virtual ~CLExecutive(){}
    virtual void Run()=0;
    virtual void WaitForDeath()=0;

protected:
    CLExecutiveFunctionProvider* m_pExecutiveFunctionProvider;

};


#endif // CLEXECUTIVE_H
```

```cpp
#include "CLThread.h"

CLThread::CLThread(CLExecutiveFunctionProvider* pExecutiveFunctionProvider):CLExecutive(pExecutiveFunctionProvider) {
}

CLThread::~CLThread() {
}

void* CLThread::StartFunctionOfThread(void *pThis){

    CLThread* pThreadThis=(CLThread *)pThis;
    std::cout<<std::endl<<"线程ID: "<<pThreadThis->m_ThreadID<<std::endl;
    pThreadThis->m_pExecutiveFunctionProvider->RunExecutiveFunction();
    return 0;
}

void CLThread::Run(){
    int r=pthread_create(&m_ThreadID,0,StartFunctionOfThread,this);

    if(r!=0)
    {
        std::cout<<"pthread_create error"<<std::endl;
        return ;
    }
}

void CLThread::WaitForDeath(){
    int r=pthread_join(m_ThreadID,0);
    if(r!=0)
    {
        std::cout<<"In CLThread::WaitForDeath(),pthread_join error"<<std::endl;
        return ;
    }
}
```

```cpp
#ifndef CLEXECUTIVEFUNCTIONPROVIDER_H
#define CLEXECUTIVEFUNCTIONPROVIDER_H


//执行体功能的接口类，线程业务逻辑的提供者
class CLExecutiveFunctionProvider{
public:
    CLExecutiveFunctionProvider(){}
    virtual ~CLExecutiveFunctionProvider(){}
public:
    virtual void RunExecutiveFunction()=0;
};

#endif // CLEXECUTIVEFUNCTIONPROVIDER_H
```

由上述代码所示，这种基于接口的思想，使线程的创建与业务逻辑的实现耦合度降低了，当使用线程时，把 CLExecutiveFunctionProvider 类作为 CLThread 类构造函数的参数，然后调用 CLThread 的 run 方法便实现了多线程。

```cpp
    CLExecutiveFunctionProvider* Search=new CLExecutiveFunctionSearch(10,1000,8);
    CLExecutive *pThread=new CLThread(Search);
    pThread->Run();
    pThread->WaitForDeath();
```

第四章 测试

1. 测试环境

```
zhanghao@zhanghao-virtual-machine:~$ cat /proc/version
Linux version 5.4.0-53-generic (buildd@lcy01-amd64-007) (gcc version 9.3.0 (Ubun
tu 9.3.0-17ubuntu1~20.04)) #59-Ubuntu SMP Wed Oct 21 09:38:44 UTC 2020
```

2. 测试方案：

首先测试表的插入，显示出插入的记录，然后测试搜索操作，搜索刚才插入的记录，看是否成功。接着测试索引文件，保存、读取、更新分别测试。最后测试多线程。

表的插入：

测试代码

```
table* m_table=table::GetTable();
m_table->InsertRecord();
```

测试结果：

```
zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ g++ main.cpp BPlusTree.cpp CLT
hread.cpp table.cpp -o main -lpthread
zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ ./main
已成功添加一条记录:
-------------------------------------------------
379  1260 1022 1848 1113 320  550  132  540  751
zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$
```

为方便展示，只显示每条记录的前 10 个属性（实际有 100 个）。

接下来搜索第一个属性为 379 的记录，测试是否插入成功。

测试代码

```
table* m_table=table::GetTable();
m_table->SearchRecord(379,379,1)
```

测试结果

```
zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ g++ main.cpp BPlusTree.cpp CLT
hread.cpp table.cpp -o main -lpthread
zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ ./main
正在搜索第1列，范围：[379,379]
未查找到索引文件，正在创建...
已为第1列创建索引文件
搜索结果为：
-------------------------------------------------
379  592  998  1309 1095 555  1526 4    1478 1006
-------------------------------------------------
379  578  1735 1336 1378 1784 1577 785  1592 399
-------------------------------------------------
379  889  1403 326  288  339  213  753  90   945
-------------------------------------------------
379  1330 935  1673 627  293  1819 655  537  437
-------------------------------------------------
379  1517 1333 154  999  244  220  1505 732  334
-------------------------------------------------
379  1448 31   1986 945  67   812  1822 1082 725
-------------------------------------------------
379  1500 1113 1453 739  86   1069 1271 239  1607
-------------------------------------------------
379  1260 1022 1848 1113 320  550  132  540  751
```

与之前的插入结果对比，发现上次插入的记录出现在了结果集中，可说明插入成功，同时也测试了搜索功能。

索引文件的创建与读取测试：

首先查看源代码所在文件夹：

如上图所示，目前文件夹中只有保存记录的文件 record.bat 和第一列属性的索引文件 index_1，现在搜索第二列属性值，由于第二列属性值没有索引文件，程序会创建。

测试代码

```
table* m_table=table::GetTable();
m_table->SearchRecord(30,1000,2);
```

测试结果

```
zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ g++ main.cpp BPlusTree.cpp CLT
hread.cpp table.cpp -o main -lpthread
zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ ./main
正在搜索第2列，范围：[30,1000]
未查找到索引文件，正在创建...
已为第2列创建索引文件
搜索结果为：
-------------------------------------------------
1093 30   114  1471 321  79   829  166  1515 1378
-------------------------------------------------
466  30   1663 641  1355 1105 1951 185  490  593
-------------------------------------------------
1777 30   1682 175  69   511  15   1571 817  124
-------------------------------------------------
1205 30   1402 498  1284 12   1247 1803 1216 1711
-------------------------------------------------
1684 30   1812 649  399  1265 1879 1864 468  527
-------------------------------------------------
655  30   769  269  162  1227 917  380  1109 1716
-------------------------------------------------
786  30   601  406  677  45   1523 1276 891  1831
-------------------------------------------------
1186 30   1970 333  1475 522  796  342  1682 1880
-------------------------------------------------
422  31   186  67   1760 1317 1181 1150 1542 1623
-------------------------------------------------
1108 32   1401 1119 1700 1950 338  739  690  1845
```

可观察到搜索结果的第二列属性值在区间[30.1000]（这里只展示了 10 个记录），接下来再看源代码所在文件夹，可发现多了一个名为 index_2 的索引文件，这便是程序为第二列属性值所创建的索引文件。

下面继续测试索引文件的读取。还是以第二个属性为例，在第二列属性值上进行搜索，这次程序不会创建新的索引文件，而是读取已有的索引文件再进行搜索。

测试代码与上一次相同

```
table* m_table=table::GetTable();
m_table->SearchRecord(30,1000,2);
```

测试结果

```
zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ g++ main.cpp BPlusTree.cpp CLT
hread.cpp table.cpp -o main -lpthread
.zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ ./main
正在搜索第2列，范围：[30,1000]
已查找到索引文件，正在读取...
搜索结果为：
-------------------------------------------------
1093 30   114  1471 321  79   829  166  1515 1378
-------------------------------------------------
466  30   1663 641  1355 1105 1951 185  490  593
-------------------------------------------------
1777 30   1682 175  69   511  15   1571 817  124
-------------------------------------------------
1205 30   1402 498  1284 12   1247 1803 1216 1711
-------------------------------------------------
1684 30   1812 649  399  1265 1879 1864 468  527
-------------------------------------------------
655  30   769  269  162  1227 917  380  1109 1716
-------------------------------------------------
786  30   601  406  677  45   1523 1276 891  1831
-------------------------------------------------
1186 30   1970 333  1475 522  796  342  1682 1880
-------------------------------------------------
422  31   186  67   1760 1317 1181 1150 1542 1623
-------------------------------------------------
1108 32   1401 1119 1700 1950 338  739  690  1845
```

可观察到搜索结果与上次相同，可以说明索引文件的保存和读取均成功。

索引文件的更新测试：

当插入了一条记录后，应该及时更新已有的索引文件，保持同步。

首先插入一条记录

```
table* m_table=table::GetTable();
m_table->InsertRecord();
```

插入结果

```
zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ g++ main.cpp BPlusTree.cpp CLT
hread.cpp table.cpp -o main -lpthread
./zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ ./main
已成功添加一条记录：
-------------------------------------------------
705  1815 640  622  1393 1542 315  139  1835 922
```

插入一条记录后，发现插入的记录第二个属性值为 1815，接下来在第二列属

性上搜索 1815，若找到了刚才插入的记录，即可说明索引文件更新成功。

测试代码

```
table* m_table=table::GetTable();
m_table->SearchRecord(1815,1815,2);
```

搜索结果

```
zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ g++ main.cpp BPlusTree.cpp CLT
hread.cpp table.cpp -o main -lpthread
./zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ ./main
正在搜索第2列，范围：[1815,1815]
已查找到索引文件，正在读取...
搜索结果为：
-------------------------------------------------
672  1815 152  431  1615 354  1077 1683 1298 504
-------------------------------------------------
1227 1815 284  1077 112  1151 1846 1797 1211 1050
-------------------------------------------------
141  1815 676  1476 950  1005 191  768  1368 1645
-------------------------------------------------
705  1815 640  622  1393 1542 315  139  1835 922
zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$
```

可观察到之前插入的记录出现在了搜索结果中，可说明索引文件更新成功。

多线程的测试

测试代码

```
CLExecutiveFunctionProvider* m_insert=new CLExecutiveFunctionInsert();
CLExecutiveFunctionProvider* Search=new CLExecutiveFunctionSearch(10,1000,8);
CLExecutive *pThread1=new CLThread(Search);
pThread1->Run();
pThread1->WaitForDeath();
CLExecutive *pThread2=new CLThread(m_insert);
pThread2->Run();
pThread2->WaitForDeath();
```

创建一个插入线程和搜索线程

测试结果

```
zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ g++ main.cpp BPlusTree.cpp CLThread.cpp table.cpp -o main -lpthread
,zhanghao@zhanghao-virtual-machine:~/Desktop/projects/DataBase$ ./main

线程ID: 139897684141824
正在搜索第8列，范围：[10,1000]
未查找到索引文件，正在创建...
已为第8列创建索引文件
搜索结果为：
-------------------------------------------------
509  234  1805 1188 363  1495 1332 10   932  1347
-------------------------------------------------
1594 165  992  1531 1028 1940 1429 10   1140 1137
-------------------------------------------------
1502 501  812  1503 519  1658 1653 10   1718 849
-------------------------------------------------
1038 1197 1371 120  1441 1355 736  10   561  274
-------------------------------------------------
866  863  842  478  257  483  181  10   336  492
-------------------------------------------------
777  1972 1094 858  1606 641  788  10   846  1449
-------------------------------------------------
370  394  1315 495  748  446  1220 10   969  308
-------------------------------------------------
550  1529 1066 203  1631 807  911  10   55   169
-------------------------------------------------
423  544  116  922  1970 1874 646  10   607  598
-------------------------------------------------
1012 344  945  1557 1042 736  1988 10   1646 639

线程ID: 139897684141824
已成功添加一条记录：
-------------------------------------------------
1996 985  1448 1969 1311 1205 1065 212  1282 1511
```

通过显示的线程 ID，可说明多线程创建成功。

附录

本次实验由本人独立完成，完整代码地址：https://github.com/zhanghao731/linux-