电 子 科 技 大 学
作 业 报 告

学生姓名：　　学 号：　　指导教师：

学生 E-mail：

Linux 下表的实现与应用

第一章 需求分析

1. 总体要求
   (1) 在 Linux 环境下，采用 C 或 C++，应用现代程序设计思想。
   (2) 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
   (1) 利用已学的文件操作 API，在文件系统中存储一张表。
   (2) 该表有 100 个属性，每个属性都是 8 字节大小。
   (3) 需要支持的最大行数为 1 百万行（可看作支持行数没有上限限制）。
3. 添加要求
   (1) 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
   (1) 提供 API 函数，实现对表格的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。
   (2) 用户可以指定在哪一个属性上进行搜索。
   (3) 当搜索结果包含的行数过多时，可以只返回一小部分，如 10 行等。
5. 索引要求
   (1) 提供 API 函数，为表格的某一个属性建立索引结构，以实现快速搜索。
   (2) 自行选择使用哪种数据结构，建立索引结构，比如 B+树等。
   (3) 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
   (1) 应用程序可以以多线程的方式，使用上述 API。
   (2) 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
   (1) 表中的数据随机生成
   (2) 测试用例要覆盖主要的需求
8. 其他要求
   (1) 要求使用 C 或 C++语言，应用现代程序设计思想。

第二章 总体设计

1. 程序总体架构

程序总体架构分为三个部分，B+树（索引）、record_table（表格）以及一个线程管理器，如下图所示。

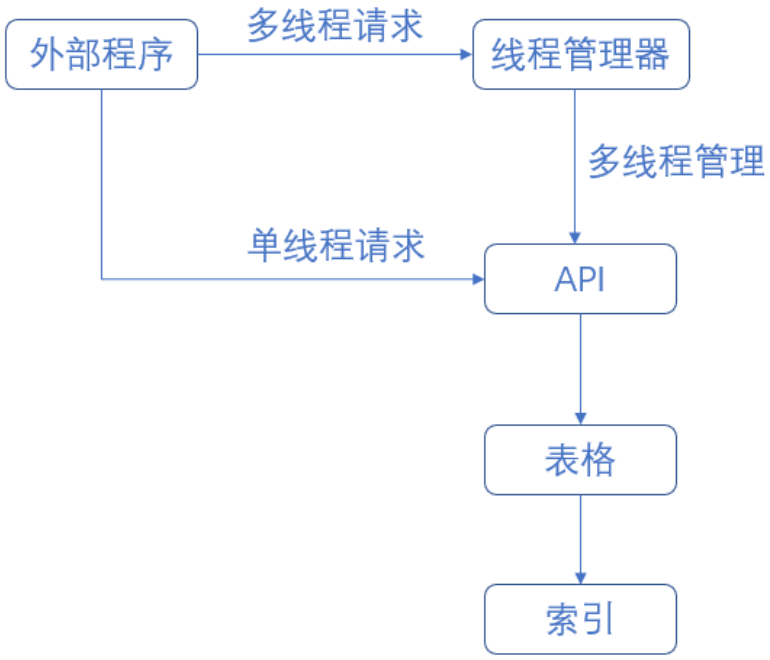

图 2-1

2. 主要流程分析

首先是表格的 API，根据需求分析我们可以知道，API 需要提供三个接口给外部程序或线程管理器，分别是插入函数，查询函数以及对指定属性的创建索引函数，因此这三个函数在表格类的设计中以 public 属性公开给外部程序访问。

第二部分是表格的索引，由于索引是由用户指定是否创建的，因此最开始的时候表格内的所有属性都没有索引，一段时间后表格内的属性也不一定都存在索引，因此表格需要记录索引标记数组，这个数组代表了某一个属性是否已经创建了索引，这对于后续的索引更新十分重要。

索引本体采用 B+树实现，采用 B+树的好处是其一它的深度不会太深，不会像二叉树那样可能需要几十次 IO 操作，其二是 B+树是自平衡的，也就是说 B+树是一棵平衡二叉树，不会出现因为插入顺序不同而导致的树不平衡的情况，其三是 B+树对于节点的读取是以块为单位的，块的大小称为节点的阶数，这十分契合计

算机磁盘按块读入的特性，这是其他数据结构所不具有的优点

可以看到外部程序调用时分为两种情况，第一种情况是并发环境（多线程环境），第二种情况是无并发环境（单线程环境），两种环境主要的不同即在于对表的操作是否会产生并发的情况。

在非并发的状态下，外部程序的请求直接提交给表格，由于不存在线程互斥的问题，直接进行写入、读取、创建索引的操作即可，如图 2-2，图 2-3，图 2-4 所示。

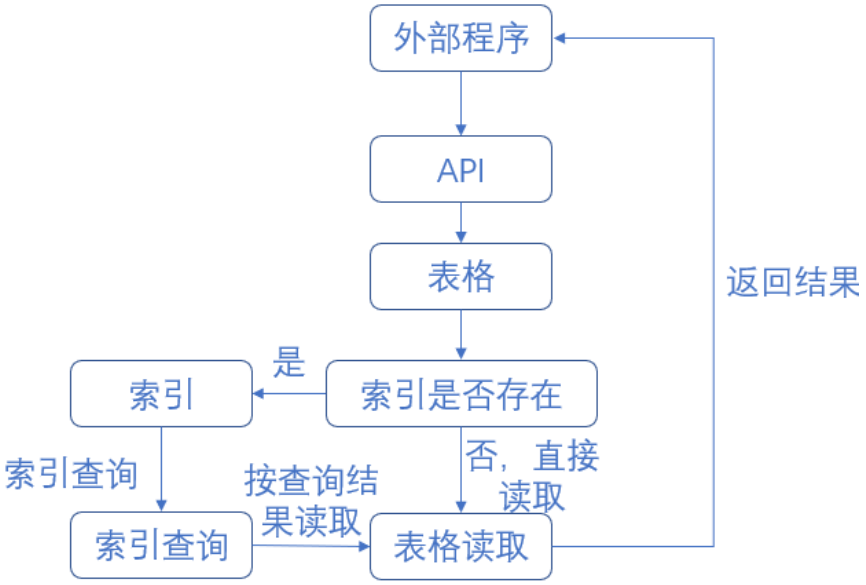

图 2-2 无并发环境读取

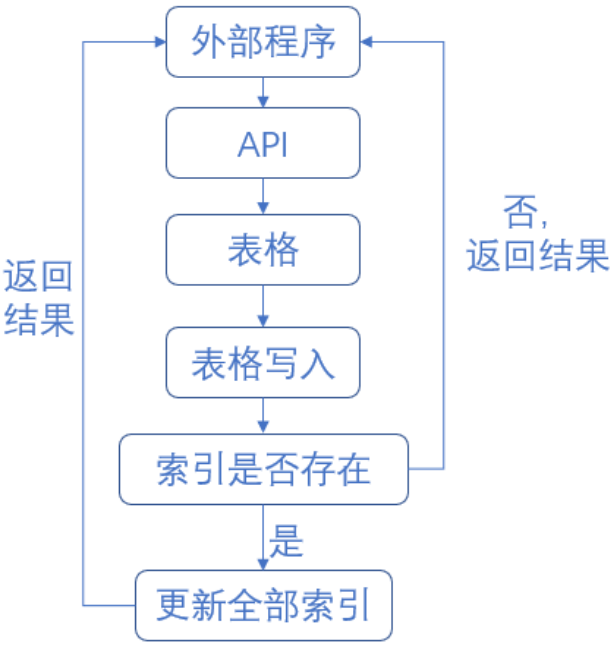

图 2-3 无并发环境写入

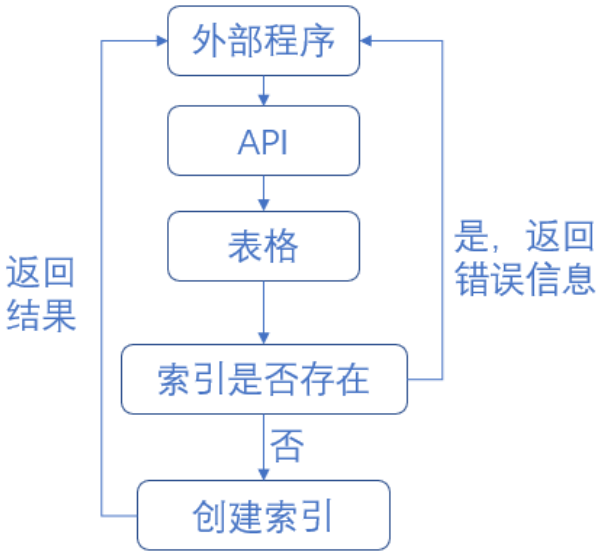

图 2-4 无并发环境创建索引

在并发的情况下需要有一个线程管理器来分配线程以实现并发操作，此时，用户的请求提交给线程管理器，线程管理器创建线程并将请求通过调用 API 的方式提交给表格，需要注意的是线程管理器并不负责表格的互斥访问，对于并发操作时的互斥由表格内部的读写锁完成，图 2-5，图 2-6，图 2-7 展示了并发情况下读取、写入、创建索引三个 API 的流程。

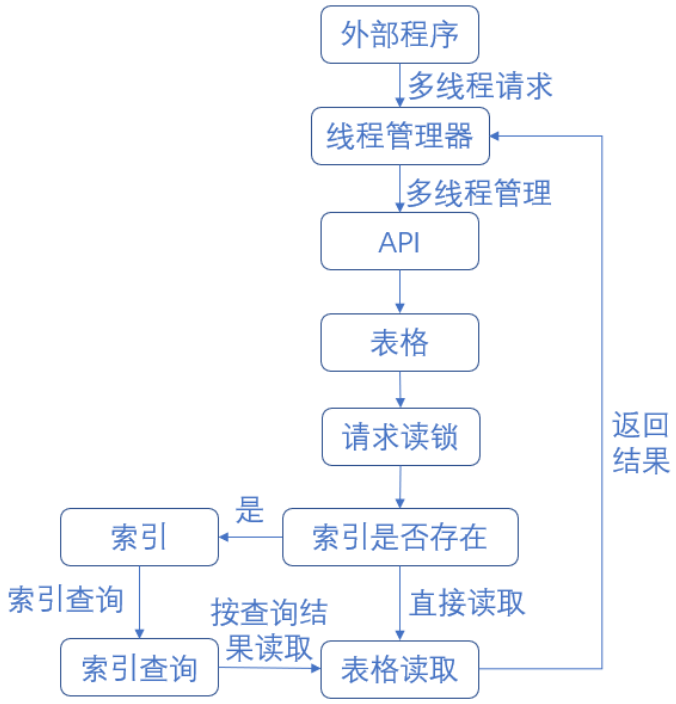

图 2-5 并发环境读取

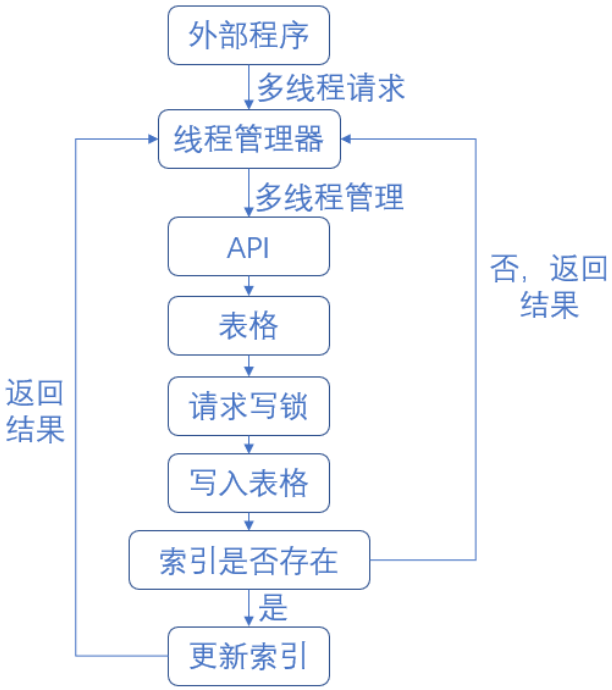

图 2-6 并发环境写入

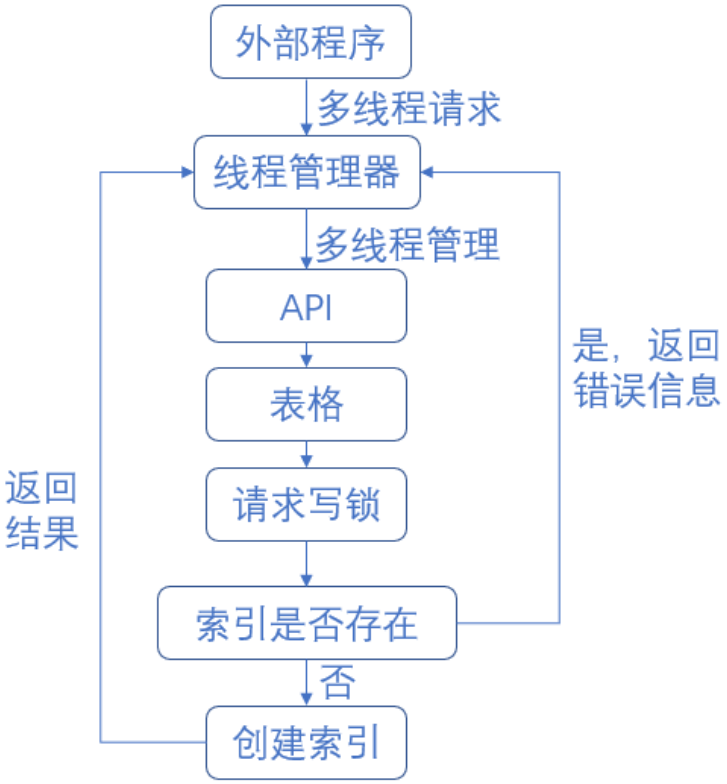

图 2-7 并发环境创建索引

创建索引的过程同样需要申请写锁，因为如果有线程在索引没有创建完成的时候就通过索引进行搜索，就会导致索引创建出错或者索引返回错误的结果，两种情况都无法保证数据一致性，因此在创建索引的过程中仍然需要申请写锁。

3. 总结

本章主要介绍了实验程序的总体架构和主要流程。

程序主体架构由索引、表格以及线程管理器组成，其中索引和表格是实验的主体部分，线程管理器则是单独的负责管理并发访问的部分。

程序主要流程有三种，分别是读取记录、写入记录和创建索引，根据情况的不同又分为并发情况和非并发情况共六种，本章对六种情况都进行了介绍。

第三章 详细设计与实现

1. 常量列表

```
const int BPLUE_LEVEL=4;
//B+树阶数
const int LENGTH_OF_RECORD=100;
//记录属性数量
const int BYTES_OF_RECORD=800;
//记录字节大小
const int SHOW_MAX_ROW=20;
//最大展示行
const int BYTES_OF_INT=4;
//int字节数量
const int BYTES_OF_BOOL=1;
//bool字节数量
const int BYTES_OF_INT64=8;
//属性字节数量
const int BYTES_OF_TREENODE=101;
//树节点字节数量
const char TABLE_NAME[]="./bin/table.txt";
//表格地址
const char INDEX_NAME[]="./index/i";
//索引地址
const char INDEX_EXISTS_NAME[]="./bin/indexe.txt";
//索引标记地址
```

图 3-1 常量列表

对应源代码中的 constant_val.h，主要定义了所有项目中需要用到的常量，包括 B+数阶数，表格长度，最大展示行数，各个文件在资源管理器中的地址以及一部分数据结构的字节树，主要用来根据序号确定数据在文件中的偏移量。

2. 基础数据结构

(1) 记录（record）

```
struct record{
    int32_t id;
    int64_t data[LENGTH_OF_RECORD];
    record();
    record(const record &);
};
```

图 3-2 记录的数据结构

定义于 table.h 文件中，主要描述了表格中每一条记录的属性，其中 data 数组是每一条记录的具体数据，id 变量则对应了记录在文件中的偏移量，该偏移量不写入文件，但需要保存在数据结构中方便创建索引。

(2) 树节点(tree_node)

```
struct tree_node{
    int64_t key[BPLUE_LEVEL+1]; //键值，5*8字节
    int32_t son_offset[BPLUE_LEVEL+2]; //儿子节点偏移量，6*4字节
    int32_t index[BPLUE_LEVEL+1];  //数据，5*4字节
    int32_t prev_offset,nxt_offset,key_num,id; //前驱，后继，键数量，id，4*4字节
    bool is_leaf; //叶子标记，1字节
};
```

图 3-3 树节点的数据结构

定义于 btree.h 中，主要描述了 B+树中每个节点需要存储的信息，其中包括：键值(key)、儿子节点在索引文件中的偏移量(son_offset)、对应的记录序号(index)，叶子节点的前驱、后继(prev/nxt)、节点中键值的数量，节点在索引文件中的偏移量，以及叶子标记。

由于在 B+树运行的过程中会出现节点键值数量多于阶数的情况，这种情况下需要分裂，因此在声明键值数组的时候需要多声明一个，儿子指针（偏移量）需要多声明两个。

3. 索引

```
struct btree{
private:
    int node_cnt,pid;
    std::string i_name;
    //文件名称
    int read_node(int id,tree_node &a);
    //读取一个节点
    int write_node(tree_node &x);
    //存储一个节点
    tree_node lower_bound(int64_t val);
    //查询第一个存储了大于等于val的键值的节点
    void split(tree_node&,tree_node&);
    //分裂函数，第一个参数为parent,第二个参数为left_chile,分裂对象是left_child
    int insert_dfs(int64_t val,int idx,tree_node &parent,int id);
    //递归的插入数据
    int insert_tonode(int64_t val,int idx,tree_node &now,int new_offset);
    //将数据直接插入节点中而不考虑分裂
public:
    void create(int id);
    //建树，有文件读文件，没文件创建文件
    std::vector<int> range_search(int64_t left,int64_t right);
    //区间搜索，返回record偏移量
    int insert(int64_t val,int idx);
    //插入函数，第一个参数为键值，第二个参数为record偏移量
    void print();
    //调试函数
};
```

图 3-4 树节点的数据结构

索引本体由 B+树实现，主要提供的功能有三部分，其一是查询部

分，根据对应的键值查询满足某种条件的记录在表中的偏移量，也即记录在表中的序号，其二则是插入一个键和其对应的属性，其三即创建接口，表格类通过调用创建接口来读取一个已经存在的索引或者创建不存在的索引，其对应类数据结构的定义如图 3-4 所示，位于 btree.h 文件中。

对应的三个接口分别为创建/读取索引：create()函数，区间查询：range_search()函数，insert()插入函数。

(1) 查询

B+树查询原理图如图 3-5 所示。

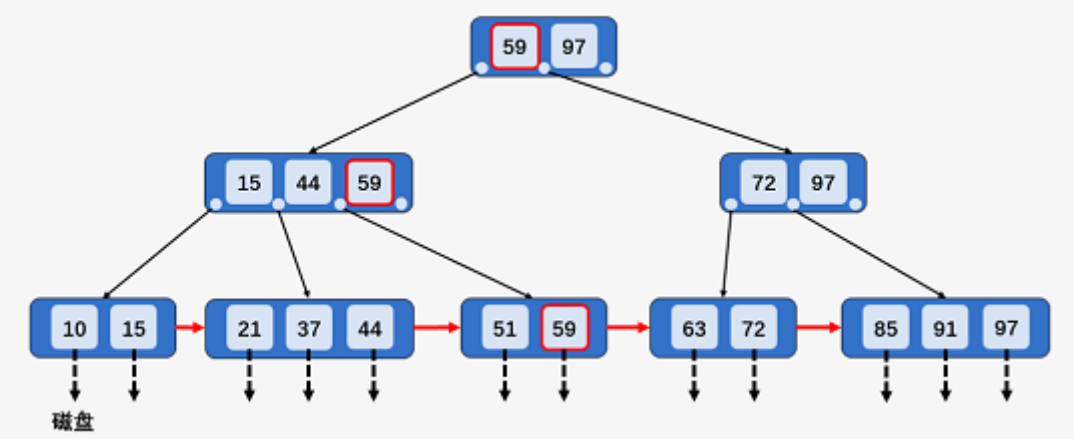

图 3-5 B+树查询过程

```
tree_node btree::lower_bound(int64_t val){
    //找到保存第一个大于等于val的key的叶子节点
    tree_node now;
    int id=0;
    read_node(id,now);
    while(!now.is_leaf){
        for(int i=0;i<now.key_num;i++){
            if(val< now.key[i]){
                //如果小于当前键，意味着一定要往左儿子走
                id=now.son_offset[i];
                break;
            }
        }
        //按id读取下一个节点
        read_node(id,now);
    }
    return now;
}
```

图 3-6 找到第一个大于等于查询值的键所对应节点

假设我们需要查找[59,72]区间内的数据，我们首先需要找到第一个大于等于 52 的键值所在的节点，根据 B+树特性，键值的左子

树对应小于等于键值的数据，右子树对应大于键值的数据，因此我们首先在 B+树上执行深度优先搜索，在节点内遍历键值，一旦发现一个键值大于查询值的键，就进入他所对应的左儿子节点，如果没有查到，则进入最右端的儿子节点，dfs 过程对应索引类的 lower_bound 函数，其实现如图 3-6 所示。

找到之后遍历节点所有值，只要满足要求就加入结果中，知道键值超出范围或者当前节点没有后继叶子节点。Range_search 函数实现如图 3-7 所示。

```
std::vector<int> btree::range_search(int64_t left,int64_t right){
    //区间查询，返回一个int的容器表示对应记录在表中的偏移量
    std::vector<int> res;
    tree_node now=lower_bound(left);
    int id=now.id;
    while(1){
        bool ok=0;
        for(int i=0;i<now.key_num;i++){
            if(now.key[i]>=left&&now.key[i]<=right){
                //满足要求则加入
                res.emplace_back(now.index[i]);
            }else if(now.key[i]>right){
                ok=1;
                break;
                //超过查询范围 退出
            }
        }
        if(ok)break;
        if(now.nxt_offset!=-1){
            //判断后继叶子节点是否存在
            id=now.nxt_offset;
            read_node(id,now);
        }else{
            break;
        }
    }
    return res;
}
```

图 3-7 对外接口范围查询函数

(2) 插入功能

插入功能涉及节点的分裂，B+树节点分裂如图 3-8 所示

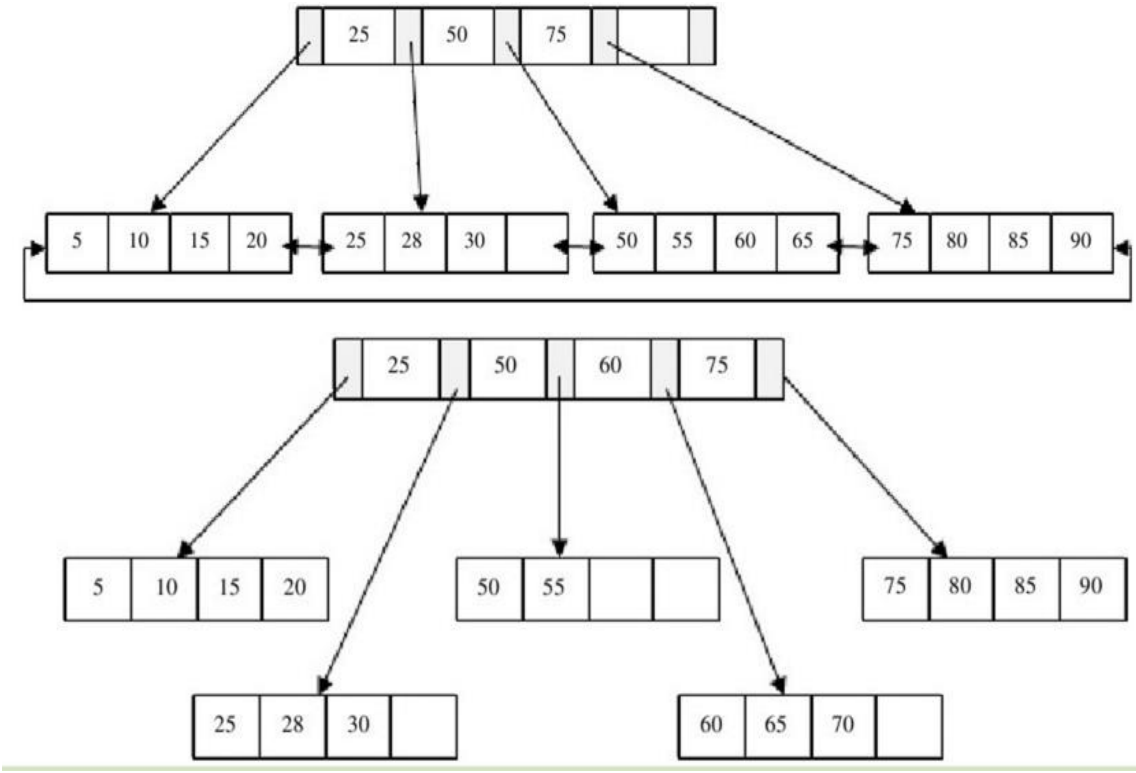

图 3-7 B+树分裂

如图 3-7 所示，当我们插入 70 这个数据到第三个儿子节点时，儿子节点的键值多于其阶数，需要分裂，我们从中拿出中间数据作为键值插入到父亲节点中，即 60，再将儿子节点分裂为左右两个，分别继承原先儿子的左半数据和右半数据，对应下图中的 3，4 叶子节点。

因此我们明确了一点，就是对于 B+树的插入是自下而上的递归的操作，首先会发生变化的是叶子节点而不是祖先节点，叶子节点插入、分裂之后才会引起祖先节点的变化，因此我们需要考虑利用递归来实现索引的插入，对应代码如图 3-8 所示。

```
int btree::insert_dfs(int64_t val,int idx,tree_node &parent,int id){
    //插入一个数据后，树的调整是自下而上的，我们需要先将数据插入叶子节点
    //再逐步调整父亲节点，因此采用dfs的方式调整
    tree_node now;
    read_node(id,now);
    if(now.is_leaf){
        insert_tonode(val,idx,now,-1);
    }else{
        int nxt=now.key_num;
        //找到插入的位置
        for(int i=0;i<now.key_num;i++){
            if(val<now.key[i]){
```

```
                nxt=i;
                break;
            }
        }
        //递归的插入
        insert_dfs(val,idx,now,now.son_offset[nxt]);
    }
    if(now.key_num>BPLUE_LEVEL){
        //插入后如果需要分裂，则进行分裂
        //父亲节点的分裂会在其对应的节点进行
        split(parent,now);
    }
    write_node(now);
    return 1;
}
```

图 3-8 递归插入

值得注意的一点是不能直接调用根节点进入该函数，因为我们无法指定根节点的父亲节点，因此我们在最外层又包装了一层 insert 函数，该函数是真正对外暴露的接口。

此外由于我们设定根节点的偏移量必定是 0，因此根节点的分裂也需要特殊处理而不能直接调用 split 函数。

Insert 函数的实现如图 3-9 所示。

```
int btree::insert(int64_t val,int idx){
    tree_node root;
    read_node(0,root);
    if(root.is_leaf){
        insert_tonode(val,idx,root,-1);
    }else{
        int insert_pos=root.key_num;
        for(int i=0;i<root.key_num;i++){
            if(val<root.key[i]){
                insert_pos=i;
                break;
            }
        }
        insert_dfs(val,idx,root,root.son_offset[insert_pos]);
    }
    //需要保证root节点的偏移量为0
    //因此root节点的分裂需要特殊处理
    if(root.key_num>BPLUE_LEVEL){
        //root的分裂操作
        //...
```

图 3-9 对外接口 insert 函数

4. 表格

```
struct record_table{
public:
    static record_table *get_instance();
    //单例模式，获取实例
    int read_record(record &,int);
    //读取一条记录
    std::vector<record> search_record(int64_t,int64_t,int);
    //区间搜索
    void create_index(int);
    //对attr属性创建索引
    int insert_record (record &);
    //插入一条记录
    void print();
    //调试函数
    void* save();
    //保存索引标记
private:
    int pid;
    //文件标识符
    int num_of_record;
    //记录数量
    static record_table* m_instance;
    //单例模式的实例
    bool is_index_exists[LENGTH_OF_RECORD];
    //记录索引是否存在
    btree index[LENGTH_OF_RECORD];
    //索引
    pthread_rwlock_t rwlock;
    //读写锁，用以实现同步
    std::vector<record> direct_search_record(int64_t,int64_t,int);
    //直接搜索
    std::vector<record> search_record_with_index(int64_t,int64_t,int);
    //索引搜索

    record_table();
    ~record_table();
};
```

图 3-10 表格类的成员列表

表格需要向外提供的接口有四个，第一个是获取表格实例，第二个是区间搜索，第三个是创建索引，第四个是插入数据。

(1) 获取表格实例

为了避免冲突，表格类采用单例模式实现，获取表格实例的函数为图 3-10 中的 get_instance 函数，对应的实例为 m_instance 变量，具体的代码如图 3-11 所示。

```
record_table* record_table::get_instance()
{
    if (nullptr == m_instance )
    {
        m_instance = new record_table();
    }
    return m_instance;
}
```

图 3-11 对外接口：获取实例

(2) 区间搜索记录

根据需求，区间搜索主要分为两个部分，第一个部分是直接搜索函数，第二个部分是根据索引搜索函数，当收到用户请求后，根据当前属性是否存在索引进行搜索，若存在，则用索引加速搜索，其实现如图 3-12 所示。

此外，为了满足并发需求，在搜索之前需要请求读锁，如图 3-12 所示。

```
std::vector<record> record_table::search_record(int64_t left, int64_t right,int attr){
    //面向用户的接口
    //针对第attr条属性，搜索[left,right]区间内的所有记录
    pthread_rwlock_rdlock(&rwlock);
    std::vector<record> res;
    if(is_index_exists[attr]){
        //有索引，用索引搜索
        res=search_record_with_index(left,right,attr);
    }else{
        //如果没有索引，则顺序搜索
        res=direct_search_record(left,right,attr);
    }
    pthread_rwlock_unlock(&rwlock);
    return res;
}
```

图 3-12 对外接口：区间搜索函数

```
std::vector<record> record_table::search_record(int64_t left, int64_t right,int attr){
    //面向用户的接口
    //针对第attr条属性，搜索[left,right]区间内的所有记录
    pthread_rwlock_rdlock(&rwlock);
    std::vector<record> res;
    if(is_index_exists[attr]){
        //有索引，用索引搜索
        res=search_record_with_index(left,right,attr);
    }else{
        //如果没有索引，则顺序搜索
        res=direct_search_record(left,right,attr);
    }
    pthread_rwlock_unlock(&rwlock);
    return res;
}
```

图 3-13 直接搜索函数

其中直接搜索函数 direct_search_record 实现即根据表格中记录的数量逐一读取并判断是否满足条件，实现如图 3-13 所示

(3) 插入记录

根据需求，我们需要实现向表格添加一行记录到最末尾的功能，其具体实现如图 3-14 所示，在添加之前我们需要请求写锁。

插入完成后我们根据索引标记判断索引是否存在，如果存在则需要更新索引。

```
int record_table::insert_record(record &a){
    //插入一条记录到表的末尾
    pthread_rwlock_wrlock(&rwlock);
    //lseek(this->pid,0,SEEK_END);
    int x=write(this->pid,a.data,BYTES_OF_RECORD);
    a.id=this->num_of_record;
    this->num_of_record ++;
    save();
    //如果存在索引则需要更新索引
    for(int i=0;i<LENGTH_OF_RECORD;i++){
        if(is_index_exists[i]){
            index[i].insert(a.data[i],a.id);
        }
    }
    pthread_rwlock_unlock(&rwlock);
    return x==BYTES_OF_RECORD?1:-1;
}
```

图 3-14 对外接口：插入函数

(4) 创建索引

根据需求，我们需要实现用户指定创建索引的功能，具体实现方法为先初始化一个索引类，即成员中的 index 变量，调用其 create 函数创建，之后将表格中的数据插入到索引中，如图 3-15 所示。在创建索引之前同样需要请求写锁。

```
void record_table::create_index(int idx_id){
    //创建索引
    pthread_rwlock_wrlock(&rwlock);
    //此时上写锁
    if(is_index_exists[idx_id]){
        return;
    }
    record a;
    is_index_exists[idx_id]=1;
    index[idx_id].create(idx_id);
```

```
    for(int i=0;i<num_of_record;i++){
        //将表中数据逐条插入
        read_record(a,i);
        index[idx_id].insert(a.data[idx_id],i);
    }
    save();
    pthread_rwlock_unlock(&rwlock);
}
```

图 3-15 对外接口：创建索引函数

5. 线程管理器

线程管理器针对三个不同的功能提供了多线程的函数接口，如图所 3-16 所示。

```
void* range_search(void *arg){
    //搜索业务函数
    search_arg *para= (search_arg*) arg;
    done[para->thread_id]=1;
    auto table=record_table::get_instance();
    //获取实例
    auto res1=table->search_record(para->left,para->right,para->attr);
    //调用函数
    puts("");
    printf("thread: %dstart print\n",thd[para->thread_id]);
    //打印结果
    for(auto now:res1){
        for(int i=0;i<10;i++){
            printf("%lld ",now.data[i]);
        }
        puts("");
    }
    printf("thread: %dprint complete\n",thd[para->thread_id]);
    done[para->thread_id]=0;
    return nullptr;
}
```

```
void* insert_record(void *arg){
    //插入函数
    insert_arg *para= (insert_arg*) arg;
    done[para->thread_id]=1;
    auto table=record_table::get_instance();
    table->insert_record(para->a);
    printf("thread: %dinsert complete\n",thd[para->thread_id]);
    done[para->thread_id]=0;
    return nullptr;
}
```

```
void* create_index(void *arg){
    //创建索引函数
    create_arg *para= (create_arg*) arg;
    done[para->thread_id]=1;
    auto table=record_table::get_instance();
    table->create_index(para->attr);
    printf("thread: %dcreate index complete\n",thd[para->thread_id]);
    done[para->thread_id]=0;
    return nullptr;
}
```

图 3-16 线程控制器业务函数

线程管理器管理着一片线程池，每次有需求时会先寻找一个空置的线程，并分配给请求。

我们可以通过一定的输入格式来创建不同的请求，具体格式如图 3-17 所示。

```
//输入格式
//0查询：格式 0 left right attr,表示对第attr条属性查询位于[left,right]区间内的记录
//1插入：格式 1,表示插入一条随机的记录
//2创建：格式 2 attr,表示对第attr条属性创建索引
//3退出：格式 3，表示退出程序
```

图 3-17 输入格式

需要注意的是线程控制器仅负责分配线程和调用 API，并发控制由表格中的锁实现。

第四章 测试

1. 测试代码

测试代码如图 4-1 所示，其输入格式与第三章最后所述格式相同，我们通过该输入格式输入数据从而向表格提交请求。

```
if(op==0){
    search_arg arg;
    scanf("%lld %lld %d",&arg.left,&arg.right,&arg.attr);
    arg.thread_id=now;
    pthread_create(&thd[now],NULL,range_search,&arg);

}else if(op==1){
    insert_arg arg;
    arg.a=gen_record();
    arg.thread_id=now;
    pthread_create(&thd[now],NULL,insert_record,&arg);
}else if(op==2){
    create_arg arg;
    scanf("%d",&arg.attr);
    arg.thread_id=now;
    pthread_create(&thd[now],NULL,create_index,&arg);
}else if(op==3){
    break;
}
sleep(1);
```

图 4-1 测试代码

如图 4-1 所示，其输入格式与第三章最后所述格式相同，我们通过该输入格式输入数据从而向表格提交请求。

数据生成器如图 4-2 所示，采用 mt19937 随机数，随机种子由时间决定

```
std::mt19937 rnd(time(NULL));
record gen_record(){
    record a;
    for(int i=0;i<LENGTH_OF_RECORD;i++){
        a.data[i]=rnd()%2000;
    }
    return a;
}
```

图 4-2 随机数生成器

2. 表格的创建和数据插入

新建表格时，表格为空并且未实例化，其工作目录如下图 4-3 所示为空

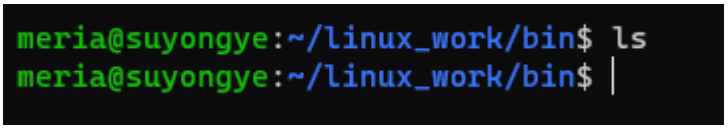

图 4-3 创建表格前

插入指令执行中：

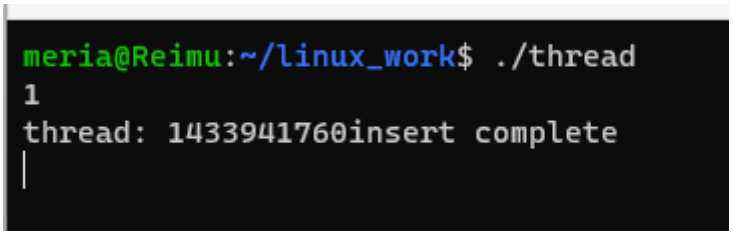

图 4-4 插入表格执行中

插入完成后，表格文件成功创建：

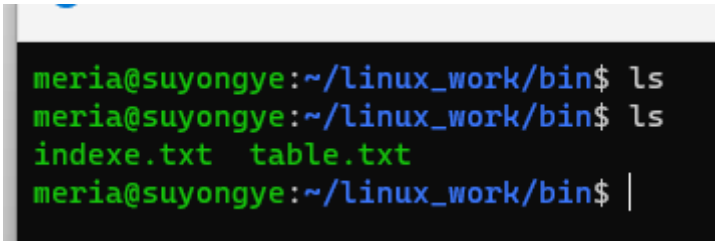

图 4-5 表格成功创建

此时查询表格中所有数据：

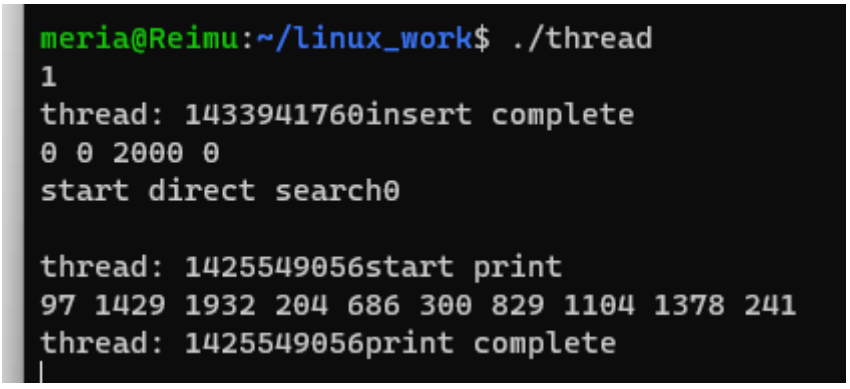

图 4-5 表格成功创建

成功查询，说明数据已经存入表格，这里为了方便只显示了记录的前 10 个数据。

3. 索引的创建和利用索引查询

在测试之前我们预先向表格中插入若干数据：

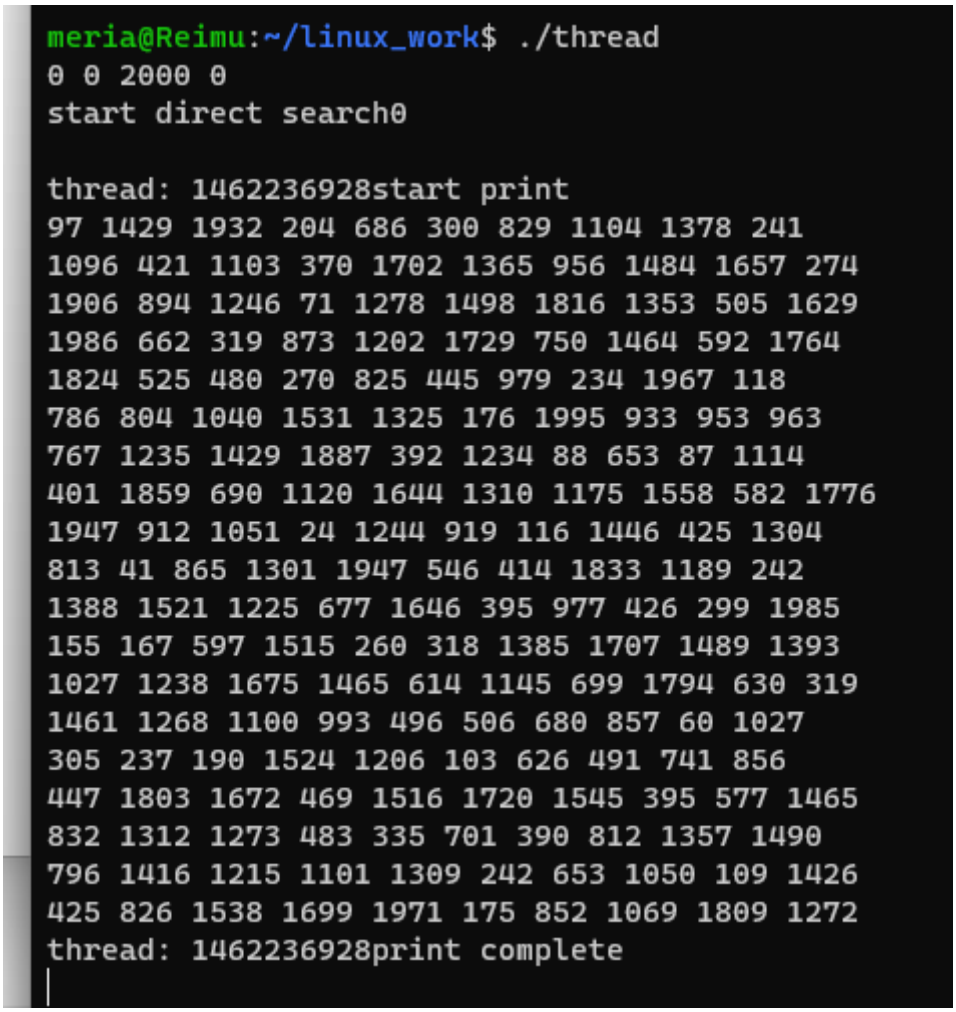

图 4-6 预先插入若干数据

为了验证查询正确性，在查询之前我们先用直接搜索功能查询第 0 个属性，值在 0 到 600 间的所有记录。

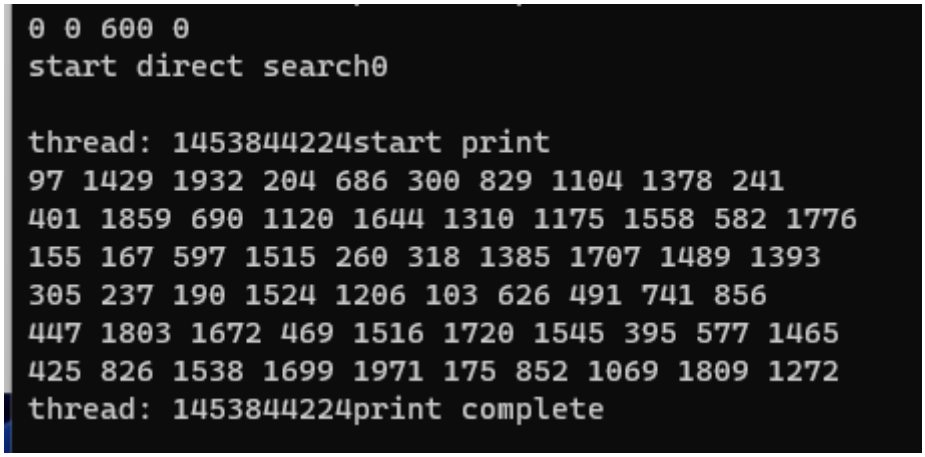

图 4-7 创建索引前查询

然后我们创建索引：

```
2 0
thread: 1445451520create index complete
```

图 4-8 创建索引

创建索引后再次查询：

```
0 0 600 0
start search with index 0

thread: 1437058816start print
97 1429 1932 204 686 300 829 1104 1378 241
155 167 597 1515 260 318 1385 1707 1489 1393
305 237 190 1524 1206 103 626 491 741 856
401 1859 690 1120 1644 1310 1175 1558 582 1776
425 826 1538 1699 1971 175 852 1069 1809 1272
447 1803 1672 469 1516 1720 1545 395 577 1465
thread: 1437058816print complete
```

图 4-9 创建索引后查询

可以看到查询的顺序和直接查询有所不同，因为直接查询中，记录是按偏移量升序排列，而索引查询中则是按键值升序排列。

创建的新索引文件如图 4-10 所示

```
meria@suyongye:~/linux_work/index$ ls
meria@suyongye:~/linux_work/index$ ls
i0
meria@suyongye:~/linux_work/index$
```

图 4-10 索引文件

4. 多线程测试

```
1
1
thread: 1308604160insert complete
1
thread: 1300211456insert complete
0 0 600 0
thread: 1291818752insert complete
thread: 1283426048insert complete
thread: 1275033344insert complete
start search with index 0

thread: 1266640640start print
97 1429 1932 204 686 300 829 1104 1378 241
155 167 597 1515 260 318 1385 1707 1489 1393
263 1039 626 994 709 1169 726 70 1566 19
288 1658 510 795 962 1158 1761 1262 1953 1081
305 237 190 1524 1206 103 626 491 741 856
401 1859 690 1120 1644 1310 1175 1558 582 1776
425 826 1538 1699 1971 175 852 1069 1809 1272
447 1803 1672 469 1516 1720 1545 395 577 1465
thread: 1266640640print complete
```

图 4-11 多线程测试

我们快速提交多个线程给表格，可以看到表格进行了并行处理而非串行处理，如图 4-11 所示
其中的线程 id 也说明了程序支持并行处理。

附录

代码见 https://github.com/Infinempty/linuxdatabase