IndexManager 设计报告

--计算机科学与技术学院 计算机科学与技术专业 1405 班 吴陈奥 3140103809

一、 模块概述：

MiniSQL 要求自动对用户创建的主键进行一个 B+树的索引同时也可以在这个表从创建到删除的整个过程的每个时候对一个声明为 unique 的属性进行索引。这保证了 B+树中的值肯定是不会有重复的。

IndexManager 这个模块实现了 B+树索引的创建，查找，插入和删除等多个功能。每次从根节点读取数据，通过数据的比较，每次往下一层读取一个新的块。因为每个块都有 4K 的空间，使得一个节点的儿子数量众多，而在内存中比较的速度十分快，这就大大节省了直接在 recordManager 中遍历数据把每个块都读取进内存的速度

注:本模块及本工程所用语言为 C++，本模块实现时所用编译器为 VS。

二、 主要功能

【创建索引】

用户在对一个表插入许多数据后对一个属性进行索引，（一开始建表对主键的索引不需要调用此函数，因为那个时候还没有数据，只需要在 catalog 里把根节点创建好即可）根据输入的数据在内存中建立相应的一棵 B+树，在对每个节点操作完成后及时调用 bufferManager 更新底层数据，完成创建。如果失败，通过 C ++异常机制告诉上层模块失败原因。

【单值查找】

对一个给定的数据进行查找，从根节点到叶节点，在相应叶节点没有找到数据则通过异常返回机制返回没有找到的信息最后反馈给用户。

【多值查找（条件查找）】

上层模块通过用户的条件给定一个最小值和一个最大值进行查找，搜索到叶节点再横向搜索把符合条件的数据通过数据类返回给用户，没有找到则通过异常处理机制返回给上层模块最终反馈给用户。

【单值插入】

上层模块给定一个值，递归进行插入，最终反馈给用户插入成功或者失败的相关信息。多值同时插入也只需要多次调用单值插入即可。

【多值删除（包含单值删除）】

上层模块根据用户要求给定一个最小值和最大值，如果是单值删除则只需要将俩个值变成相等即可。通过一个搜索的郭层，删除符合条件的这个区间的所有值，并将结果反馈给用户。

（删除索引）：只需要 catalog 模块调用 buffer 删除文件即可不需要 index 的模块来单独做这件事。

三、 对外提供的借口

【创建索引】

```
void createIndex(Index& indexinfor, IndexColumn DataofIndex);
```

需要传入一个有索引信息的 index 的类以及包含数据信息的 data 的类。

【单值查找】

```
int selectEqual(Index indexinfor, string key);
```

需要传入索引信息 index 的类以及需要搜索的 key 值。

【多值查找(条件查找)】

```
list<int>::iterator selectBetween(Index indexinfor, string keyFrom, string keyTo,
int& blockoffset);
```

需要传入 index 类以及最小的 key 值和最大的 key 值。

【单值插入】

```
void insertValue(Branch& Searchnode, IndexRow dataRecord, Index& indexinfor);
```

需要传入根节点的信息以及包含需要插入数据的信息的类 indexrow 和包含 index 数据的类 index。

【多值删除（包含单值删除）】

```
void deleteValue(Index indexinfor, IndexColumn dc, int&blockoffset);
```

需要传入 index 类以及包含需要删除哪些数据的信息的类 indexcolumn 和一个反馈信息的参数 int&。

(删除索引)：只需要 catalog 模块调用 buffer 删除文件即可不需要 index 的模块来单独做这件事。

四、 设计思路

需要设计一个 B+树的类用在内存里面进行操作。同时根据位置不同性质不同的情况衍生出 Branch 和 Leaf 俩种类，一种代表中间的节点一种代表叶子节点。(注意内点也包含有根节点所以其实需要知道一个中间节点是一个分支还是一个根）

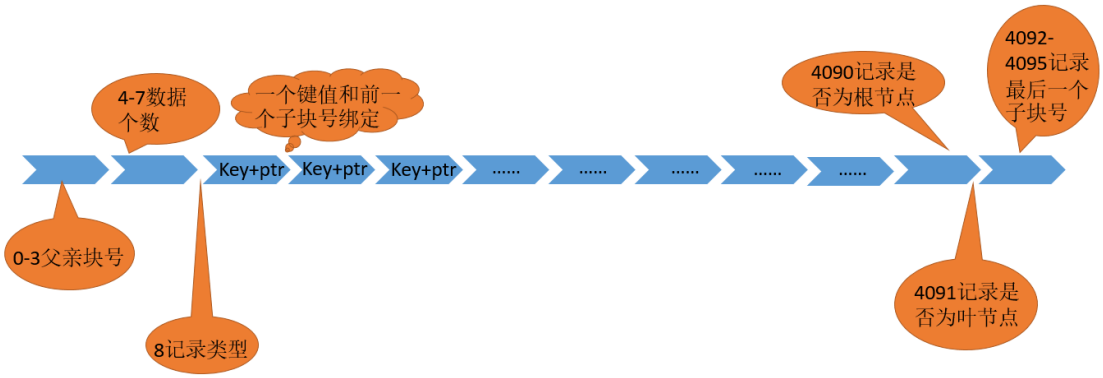
分支节点的工作是区分不同大小的分支，而叶子节点是直接指向当前键值的块号和在块中的偏移量。所以在对树的插入删除等动作的操作其实都是在内存中对抽象出来的这个类进行操作，在每个节点的操作完成之后再通过转化函数通过 buffer 存到硬盘就可以了。

Buffer 中转存到硬盘的一个媒介是一个数组，也就是通过数组调用了硬盘的空间到内存里面，在这里我们就需要对这个数组如何存放作出一些约定以便于在后面的翻译函数和反翻译的时候不会出错。在这个转换过程中，该模块都是把一个 32 字节的 int 类型的数据转换成同样空间大小的 4 个 char 类型的数据，当然读 int 的时候也是一样的。首先，这个 4K 大小的数组的头 4 个 char 是存放的他指向的父亲的块号，紧接着存放的是这个记录有多少条记录。而 key 值得长短存在了 catalog 里面所以我们就没有必要在每个节点里面都存放一遍 key 值得大小了。

在 Branch 的块里面，每个 key 和它前面一个子块号绑定在一起，顺序往后存放，

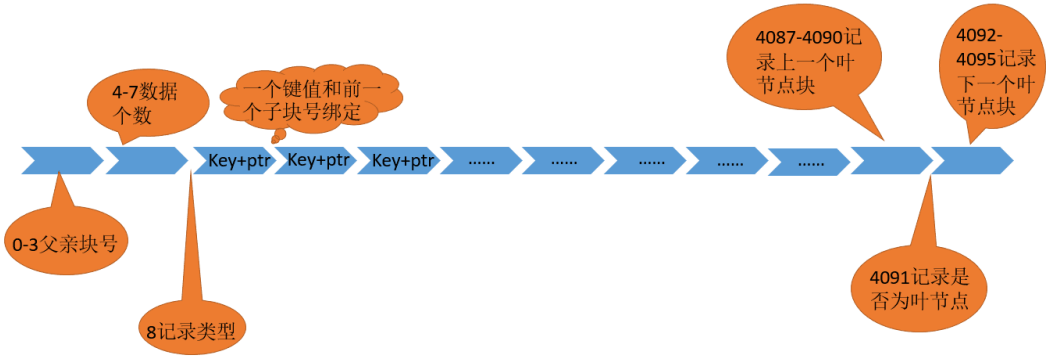
这样会多出一个 lastchild 存放在最后面，同时需要两个标志位表明当前这个块里面的数据是 INT 型还是 STRING 型以及标志当前这个块是一个 Leaf 还是一个 Branch。

中间块 Branch 的结构设计如下：



在 Leaf 块里面，每个 key 值和存放这个 key 值的块号和在块里面的偏移量刚好存放在了一起，所以不需要一个 lastchild。不过叶子节点是需要一个指针指向前一个块和后一个块以便于后续操作的，所以在块的最后有一个 nextsibling 和 lastsibling 的记录以及当前块是 Leaf 还是 Branch 的记录。

叶块 Leaf 的结构设计如下：



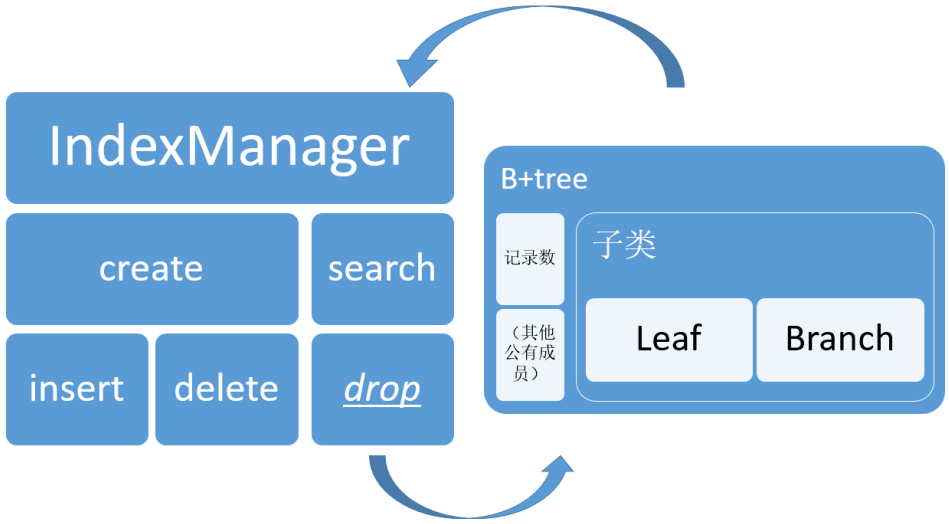
五、 整体架构

IndexManager 类的公有函数对外提供接口，在对输入的数据进行相应需要操作的格式规范之后传入私有函数，也就是说，私有函数真正实现了插入，删除等真正意义上的操作。这个操作其实是对 B+tree 这个类进行操作的，前面已经说了，这是一个把硬盘中的数据抽象到内存中数据结构的一个类，在 indexManager 中的私有函数真正要进行相关操作的时候，将硬盘中的数据通过 bufferManager 读入得到 B+tree 这一类的结构，然后进行完相关操作最终写通过 bufferManager 写回了硬盘。

B+tree 是一个抽象在内存方便进行操作的数据结构，公有域包含一些叶子和分支节点都有的数据比如当前块记录的数量等，私有函数根据不同有一些变换。同时 B+tree 的构造函数和重构的函数实现了把一个 char 转换成自己内部便于操作数据的过程，所有 IndexManager 每次得到一个由数据块的 char 的时候传给 B+tree 就会得到一个可以使用的 B+tree 的类，在使用完成后又可以翻译回 char 的数组最终写回硬盘这样一来对于一

个在硬盘中实际存在的 char 的更新就完成了。

IndexManager 和 B+tree 类的关系结构图如下：



六、 关键函数和代码

IndexManager 的插入函数：

```
void IndexManager::insertValue(Branch& , IndexRow, Index&) {//接受上层信息并转换函数
    IndexLeaf readyinsert(key,fileoffset,blockoffset);//通过数据创建叶子的元
    insertvalue(node, readyinsert, index, Fp, valid, ptrChild, formal);//执行插入
    if (/*下一层节点多分列了一个元*/) {
        if(/*根节点没有满*/)
            //插入
        else
            //创建俩个新的叶子，根节点不变，这样就不用维护catalog里的根节点的信息了
    }
}

void IndexManager::insertvalue(...) {//真正插入的函数
    if (根节点没有元) {
        //创建一个叶子并插入
        block1 = Buffer.create_newblock_index(indexinfor);
        blockoffset1 = Buffer.malloc_block(indexinfor.fileoffset, block1);
        Tleaf1.createLeaf(indexinfor.columnLength, indexinfor.type);
        Tleaf1.insert(data);
        //维护ptrfather
        translateLeaf(Buffer.b[blockoffset1].block, Tleaf1);//写回
        //维护lastchild

    }
```

```
    else {
        if (读进的节点为分支节点) {
            //插入
            insertvalue(Tbranch1, data, indexinfor, fp, next, nextptrch, nkey);
            if (下层分裂了一个节点) {
                if (本层不需要分裂) {
                    //直接插入同时维护valid
                    Tbranch1.insert(tbranch1);
                    translateBranch(Buffer.b[block1].block, Tbranch1);
                }
                else if (需要分裂) {
                    //create一个新的块
                    block2 = Buffer.create_newblock_index(indexinfor);
                    //将第一个的插入新的同时维护father和上一层当前元的值
                    //修改每个被改了父亲的节点的ptrfather
                    changefather(indexinfor, Tbranch2);
                }
            }
            else
                *valid = 0;
        }
        else if (是一个叶子节点) {
            //调用插入叶子节点的函数
            insertvalueL(Tleaf1, data, indexinfor, next, nextptrch, nkey);
            translateLeaf(Buffer.b[block1].block, Tleaf1);
            //反馈给上一层分裂信息
        }
    }
    }
```

IndexManager 创建索引函数：

```
void IndexManager::createIndex(Index& indexinfor, IndexColumn DataofIndex) {
    for (每一个需要插入的数据)
        //调用单值插入函数插入
        insertValue(root, DataofIndex.rows[i], indexinfor);
    //更新根节点信息
    blocknum = Buffer.malloc_block(indexinfor.fileoffset, indexinfor.blockoffset);
    Buffer.b[blocknum].isWritten = 1;
    translateBranch(Buffer.b[blocknum].block, root);
    }
```

IndexManager 的删除函数：

```
void IndexManager::deleteValue(Index indexinfor, IndexColumn dc, int& blockOffset) {
    for (所有需要删除的数据) {
```

```
        //转换成需要的类
    }
        while (类不为空) {
            //调用下层删除函数
            deletevalueL(indexinfor, temp);
        }
    //}
}

void IndexManager::deletevalueL(Index indexinfor, IndexLeaf &data) {
    //找到叶子节点并定位
    blockoffsetofleaf = indextoleaf(indexinfor, data, Dnode, location);
    if (当前叶子节点删除之后不满足1/2) {
        //先删除记录并转移元
        //调出父亲
        blocknum = Buffer.malloc_block(indexinfor.fileoffset, Dnode.ptrFather);
        if () {
            //删除父亲的元并递归对分支节点进行删除操作
            deletevalueB(indexinfor, Father, data);
        }
    }
    else {
        //直接在该叶子中删除该记录
    }
    }
```

IndexManager 的单值查找函数：

```
int IndexManager::selectEqual(Index indexinfor, string key) {
    //调出根节点
    tempblockoffset = Buffer.malloc_block(indexinfor.fileoffset,
indexinfor.blockoffset);
    while (还没调用到底层的叶节点) {
        //翻译当前块到Branch类
        MemorytoBranch(Buffer.b[tempblockoffset].block, tbranch);
        //找到在下一层的位置
        i = SearchlocaB(tbranch, get, valid);
        //调用下一层
        tempblockoffset = Buffer.malloc_block(indexinfor.fileoffset, nextblock);
    }

    //调出叶子节点
    MemorytoLeaf(Buffer.b[tempblockoffset].block, tleaf);
    //找到位置并返回信息
    j = SearchlocaL(tleaf, get, valid);
```

```
    return blockoffsetR;
    }
```

IndexManager 的多值查找函数：

```
list<int>::iterator IndexManager::selectBetween(...) {
    //前面和selectequaul一样一直找到叶节点
    while (当前值比最大值小) {
        if (当前值比最小值大)
            //纳入返回结果
            result.push_back((*j).offsetinBlock);
        j++;
        if (到了当前结点尾部) {
            //通过nextsibling找到下一节点
        }
    }
    fi = result.begin();
    return fi;
    }
```