Index Manager

功能描述

Index Manager 负责管理数据库中的索引，即索引的创建、删除以及对索引内容的查找、插入、修改和删除。

Index Manager 为Record Manager提供接口

数据结构

BPTreeNode

```
template<typename T>
class BPTreeNode{
public:
    bool isLeaf;//是否为叶子结点
    int degree;//结点的度数
    int cnt;//结点中当前的key的数量
    BPTreeNode *parent, *sibling;
    //成员函数省略
};
```

- `isLeaf` 用来表示该结点是否为叶子结点
- `degree` 用来表示该结点的度数
- `cnt` 用来表示该结点中当前有的key的数量
- `parent` 表示该结点的父结点
- `sibling` 表示叶子结点的兄弟结点
- `keys` 用来表示结点中存储的搜索码
- `children` 表示非叶子结点的子结点
- `keyOffset` 表示叶子结点指向的 `record` 在该文件的序号

Nodemap

```
template<typename T>
struct Nodemap {
    int index;
    BPTreeNode<T> *node;
};
```

- `Nodemap` 构建了一个对于每个结点的 `index` 的映射

BPTree

```
template<typename T>
class BPTree {
public:
    string fileName;
    TreeNode root, head;
    int sizeofKey, level, keyCount, nodeCount, degree;
    typedef BPTreeNode<T> *TreeNode;
    //成员函数省略
};
```

- `fileName` 代表该B+树对应的文件名
- `root` 代表该B+树的根
- `sizeofKey` 代表该B+树索引对应的搜索码属性的size
- `level` 代表该B+树索引的层数
- `keyCount` 代表该B+树中子结点中key的总数
- `nodeCount` 代表该B+树中子结点的总数
- `degree` 代表该B+树的度数

_index_manager

```
template<typename T>
class _index_manager{
public:
    std::map<std::string, BPTree<T>* > tree_map;
};
```

`tree_map`代表该索引及其对应的索引名的映射

IndexManager

```
class IndexManager{
private:
    _index_manager<int> *int_im;
    _index_manager<double> *float_im;
    _index_manager<std::string> *char_im;
    //成员函数省略
};
```

对`int`，`double`和`string`三种属性类型，定义三种`index`的指针

接口定义

`Index Manager`为`Record Manager`提供接口

```
CreateIndex(const std::string index_name, Table & table, const std::string &
column_name)
FindIndex(const std::string index_name, Table & table, const std::string &
column_name, const Value &val)
InsertIndex(const std::string index_name, Table & table, const std::string &
column_name, const Value &val, int offset)
DeleteIndex(const std::string index_name, Table & table, const std::string &
column_name, const Value &val)
AlterIndex(const std::string index_name, Table & table, const std::string &
column_name, const Value &val_before, const Value &val_after, int offset)
DropIndex(const std::string index_name, Table & table, const std::string &
column_name)
```

以下三个属性对于每个函数的意义均相同

`index_name`为要操作的`index`的名称，为唯一识别该`index`的属性

`table`为该`index`对应的数据表的名称

`column_name`为该`index`对应的数据项的属性名

插入操作：

`val`为需要插入`index`的记录的属性值

`offset`为需要插入`index`的记录的偏移值

删除操作：

`val`为需要删除`index`的记录的属性值

修改操作：

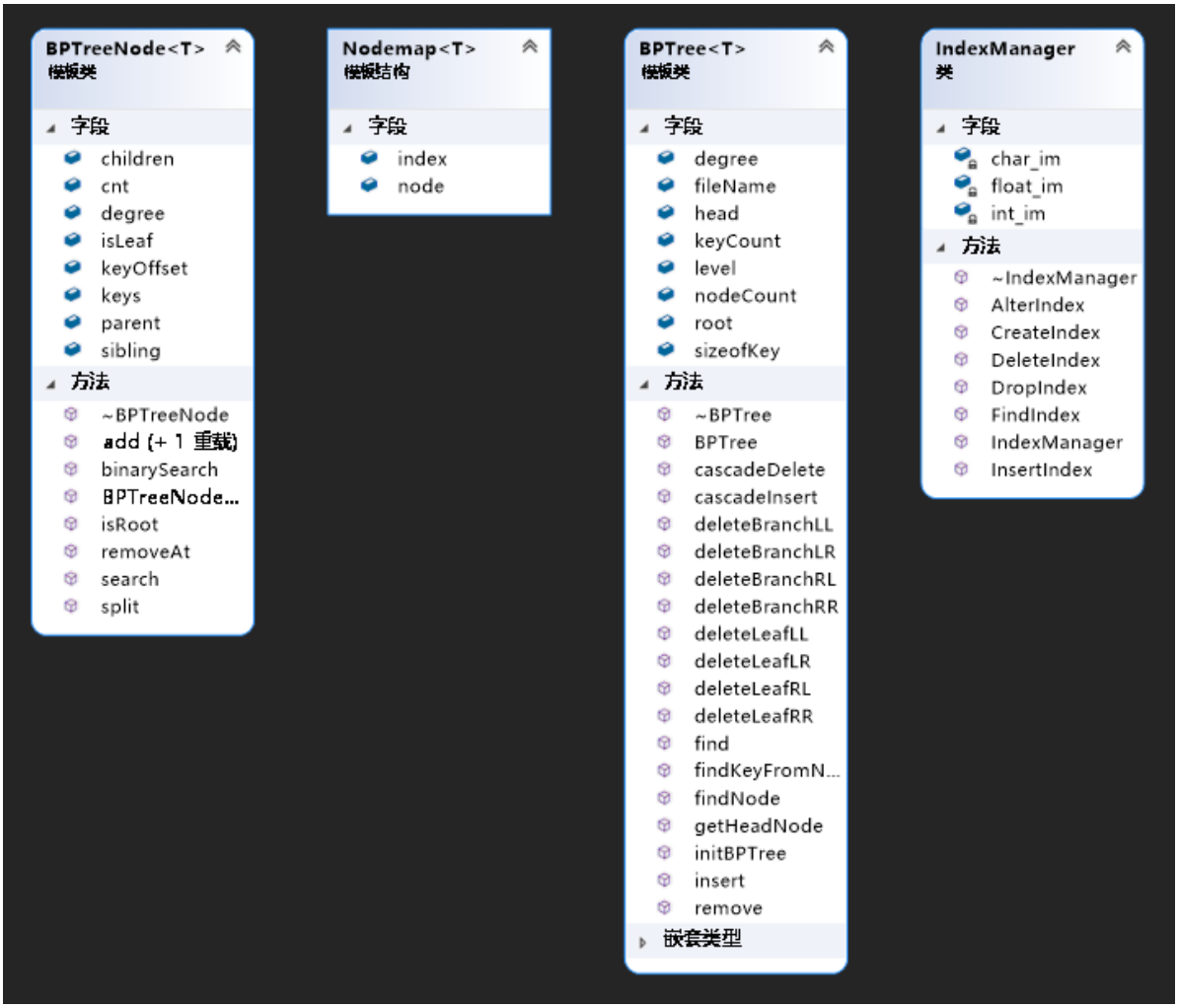
`val_before` 为需要修改的 `index` 的记录的原先的属性值

`val_after` 为需要修改的 `index` 的记录的修改后的属性值

`offset` 为需要修改的 `index` 的记录的偏移值

模块架构

类图



类内关系

`BPTreeNode` 包含了B+树结点的定义，以及结点层面的搜索、分裂、新增、删除等操作

`Nodemap` 构建了一个对于每个B+树结点的 `index` 的映射

`BPTree` 包含了B+树的定义，使用 `BPTreeNode` 和 `Nodemap` 实现了 `BPTree` 的初始化、查找、插入、删除等功能

设计思路

`Index Manager` 主要负责管理数据库中的索引，即索引的创建、删除以及对索引内容的查找、插入、修改和删除。因此需要实现B+树，对 `record` 建立B+树索引，实现创建、删除以及对索引内容的查找、插入、修改和删除等功能，并对不同的索引建立索引名和索引的map映射，进行管理

关键函数

结点的搜索（search）

```
template<typename T>//index的意思是这个key应该放在这个Node里的第几个位置上
bool BPTreeNode<T>::search(const T &key, int &index) const {
    if (cnt == 0) {//结点目前没有key，它的位置该是0
        index = 0;
        return false;
    }
    if (key < keys[0] ) {//比keys[0]还小，那位置该是0
        index = 0;
        return false;
    }
    if (key > keys[cnt - 1]){ //比keys[cnt-1]（目前有的最后一个）还大，那位置该是
cnt
        index = cnt;
        return false;
    }
    return binarySearch(key, index);
}
```

- 如果该结点当前没有`keys`，那么这个`key`应该被插入到第0个位置
- 如果该`key`比`keys[0]`还要小，那么这个`key`应该被插入到第0个位置
- 如果该`key`比`keys[cnt-1]`还要大，那么这个`key`应该被插入到第`cnt`个位置
- 若不是上述三种情况，则调用二分查找法，找到key的位置（若存在）或key应被插入到的位置

```
template<typename T>
bool BPTreeNode<T>::binarySearch(const T &key, int &index) const {
    int left = 0, right = cnt - 1, pos;
    while (left <= right) {
        pos = left + (right - left) / 2;
        if (keys[pos] < key) {
            left = pos + 1;
        } else {
            right = pos - 1;
        }
    }
    index = left;
    return keys[index] == key;
}
```

结点的新增（add）

如果新增的记录在结点中原先就已存在，则违反了唯一性规则，给出错误信息

在本函数中只负责完成新增工作，分裂等工作在`split`函数中完成

叶子结点新增记录

```
template<typename T>
int BPTreeNode<T>::add(const T &key, int offset) {//叶子结点增加记录
    int index;
    bool keyExists = search(key, index);
    if (keyExists) {
        cerr << "Key is not unique!" << endl;
        exit(10);
```

```
    }
    for (int i = cnt; i > index; i--) {
        //把key和keyoffset都向后移动一个
    }
    在index处增加这条记录
    cnt++;
    return index;
}
```

非叶子结点新增记录

```
template<typename T>
int BPTreeNode<T>::add(const T &key) {//非叶子结点增加记录
    int index;
    bool keyExists = search(key, index);
    if (keyExists) {
        cerr << "Key is not unique!" << endl;
        exit(10);
    }
    for (int i = cnt; i > index; i--) {
        把key和children都向后移动一个
    }
    //在index处增加这条记录
    //在index处增加一个指向空的孩子
    cnt++;
    return index;
}
```

结点的分裂（`split`）

```
template<typename T>
BPTreeNode<T> *BPTreeNode<T>::split(T &key) {
    //key用来往上层传
    BPTreeNode<T> *newNode = new BPTreeNode<T>(degree, isLeaf);
    int minimal = (degree - 1) / 2;//[n/2]-1
    if (isLeaf) {//叶子结点分keys
        //叶子的元素数量为[n/2]~n-1
        //把从[n/2]开始的key赋值给新的
        newNode->sibling = this->sibling;
        this->sibling = newNode;
        this->cnt = minimal + 1;
    }
    else {//非叶子结点分children和keys
        //非叶子的元素数量为[n/2]~n，且children比key多一个
        //把从[n/2]开始的key和children赋值给新的
        this->cnt = minimal;
    }
    newNode->parent = this->parent;
    newNode->cnt = degree - minimal - 1;
    return newNode;
}
```

- 叶子结点分keys,元素数量为[n/2]~n-1,将从[n/2]开始的key赋值给分裂出的新的结点
- 非叶子结点分children和keys，元素数量为[n/2]~n，且children比key多一个，将从[n/2]开始的key和children赋值给分裂出的新的结点

结点的删除（`remove`）

在本函数中只负责完成减少工作，合并等工作在`BPTree`模块中完成

```
template<typename T>
void BPTreeNode<T>::removeAt(int index) {//删除node中第index的key和children
    //把keyoffset向前移动一个
    if (isLeaf) {//叶子结点
        //把keyoffset向前移动一个
        //多出来的空位的key和keyoffset都设成0
    }
    else {//非叶子结点
        //把children向前移动一个
        //多出来的空位的key设成0，children设为指向空
    }
    cnt--;
}
```

BPTree的查找

```
template<typename T>
bool BPTree<T>::findKeyFromNode(TreeNode node, const T &key, Nodemap<T>
&res) {
    int index;
    if (node->search(key, index)) {//如果在这个node里找到了key
        if (node->isLeaf) {//这个node是叶子
            res.index = index;
        }
        else {//这个node不是叶子
            node = node->children[index + 1];//往下一层之后，一定会是最左边那个
            while (!node->isLeaf) { node = node->children[0]; }
            res.index = 0;
        }
        res.node = node;
        return true;
    }
    else {//node里没找着这个key
        if (node->isLeaf) {
            //把这个key放到res的map里
            return false;//没有
        }
        else{//向子结点进行查找
            return findKeyFromNode(node->children[index], key, res);
        }
    }
}
template<typename T>
int BPTree<T>::find(const T &key) {//返回的是key所对应的keyoffset
    Nodemap<T> res;
    if (root==nullptr) { return -1; }
    if (findKeyFromNode(root, key, res)) { return res.node-
>keyOffset[res.index]; }
    else { return -1; }
}
```

BPTree的插入

```
template<typename T>
bool BPTree<T>::insert(const T &key, int offset) {
    //往里加，度数在这个函数里判断
    Nodemap<T> res;
    if (root==nullptr) { initBPTree(); }
    //如果找到了就说明重复了，输出错误信息
    //直接调用add函数往里加
    //到达度数时需要调用cascadeInsert函数分裂
    }
    keyCount++;
    return true;
}
```

处理需要分裂的插入

```
template<typename T>
void BPTree<T>::cascadeInsert(BPTree::TreeNode node) {//需要分裂的插入情况
    T key;
    TreeNode sibling = node->split(key);//分裂
    nodeCount++;
    if (node->isRoot()) {
     //要分裂的node是root，加一层
    }
    else {//要分裂的不是root
        //给父结点加上key
        //加一个兄弟结点
        //如果父结点度数满了，则需要继续处理需要分裂的插入
    }
}
```

BPTree的删除

```
template<typename T>
bool BPTree<T>::remove(const T &key) {
    NodeSearchParse<T> res;
    if (res.node->isRoot()) {//要删的结点是根的话
        //删除根结点中的一个index
        //递归删除
        return cascadeDelete(res.node);
    }
    else {//要删的结点不是根
        if (res.index == 0 && head != res.node) {
            // 递归地更新父结点
            //找到第一个有被删掉的那个node的key的非叶子的父亲
            //把这个父结点的那个key给删了
            return cascadeDelete(res.node);
        }
        else{
            res.node->removeAt(res.index);
            keyCount--;
            return cascadeDelete(res.node);
        }
    }
```

```
}
```

```
template<typename T>
bool BPTree<T>::cascadeDelete(BPTree::TreeNode node) {
    int minimal = degree / 2, minimalBranch = (degree - 1) / 2;
    if ((node->isLeaf && node->cnt >= minimal) // leaf node
        || (node->isRoot() && node->cnt) // root node
        || (!node->isLeaf && !node->isRoot() && node->cnt >= minimal) //
branch node
            ) {
        return true; //不需要更新了
    }
    if (node->isRoot()) {
        if (root->isLeaf) {//根是叶子，整棵树就一个node
            // 把树删了
        }
        else {
            // 减一层
        }
        //把这个node给删掉
        return true;
    }
    //非根的情况
    TreeNode currentParent = node->parent, sibling;
    int index;
    if (node->isLeaf) {
        // 是叶子结点
        currentParent->search(node->keys[0], index);
        if (currentParent->children[0] != node && currentParent->cnt ==
index + 1) {
            //node是最右边但不是第一个的话，跟左边的兄弟合并
            sibling = currentParent->children[index];
            if (sibling->cnt > minimal) {
                //把左边兄弟的最右边转移到最左边
                return deleteLeafLL(node, currentParent, sibling, index);
            } else {
                return deleteLeafLR(node, currentParent, sibling, index);
            }
        }
        else {
            //跟右边兄弟合并
            if (currentParent->children[0] == node) {
                //这个node是最左边的那个
                sibling = currentParent->children[1];
            }
            else {
                sibling = currentParent->children[index + 2];
            }
            if (sibling->cnt > minimal) {
                // add the leftest of sibling to the right
                return deleteLeafRL(node, currentParent, sibling, index);
            }
            else {
                // merge and cascadingly delete
                return deleteLeafRR(node, currentParent, sibling, index);
            }
        }
```

```
    }
    else {
        //是非根非叶子结点
        currentParent->search(node->children[0]->keys[0], index);
        if (currentParent->children[0] != node && currentParent->cnt ==
index + 1) {
            //只能跟最左边的兄弟合并
            sibling = currentParent->children[index];
            if (sibling->cnt > minimalBranch) {
                return deleteBranchLL(node, currentParent, sibling, index);
            } else {
                // 删掉然后合并
                return deleteBranchLR(node, currentParent, sibling, index);
            }
        }
        else {
            //跟右边兄弟合并
            if (currentParent->children[0] == node) {
                sibling = currentParent->children[1];
            } else {
                sibling = currentParent->children[index + 2];
            }
            if (sibling->cnt > minimalBranch) {
                //把第一个key加到右边兄弟
                return deleteBranchRL(node, currentParent, sibling, index);
            } else {
                // merge the sibling to current node
                return deleteBranchRR(node, currentParent, sibling, index);
            }
        }
    }
}
```

index的创建

对于`int`和`double`类型的变量使用统一的模板类型，对于`string`类型的变量进行模板具体化

`int`和`double`类型的变量

```
template<typename T>
void IndexManager<T>::create_index(BPTree<T> &tree,Table & table,std::string
& column_name){
    //计算每个block中record的条数
    //利用table.hpp中提供的indexOfCol获取该属性的序号index_col
    //利用table.hpp中提供的blockCnt，遍历blockCnt个block
    for(int i=0;i<table.blockCnt;i++){//找blockcnt个block
        //对于每个block，利用table.hpp中提供的bread函数读取table文件的第i个块
        //利用table.hpp中提供的baddr函数读取block中存储的数据
        for(int j=0;j<num_record;j++){
          //对于每一条record
          //如果record的首位为0，则代表该record不可用，跳到下一条record并continue
          //否则根据每个columns的size，找到column_name对应的那个地址起始值
          //读取需求column_name的那个搜索码值
            T key = *(T*)data;
          //调用BPTree的insert函数插入该record即可
            }
        }
```

```
    }
}
```

`string`类型的变量

进行模板具体化

```
template<>
inline void _index_manager<std::string>::create_index(const std::string
index_name, Table & table,const std::string & column_name){
    //计算每个block中record的条数
    //利用table.hpp中提供的indexOfCol获取该属性的序号index_col
    //利用table.hpp中提供的blockCnt，遍历blockCnt个block
    for(int i=0;i<table.blockCnt;i++){//找blockcnt个block
        //对于每个block，利用table.hpp中提供的bread函数读取table文件的第i个块
        //利用table.hpp中提供的baddr函数读取block中存储的数据
        for(int j=0;j<num_record;j++){
          //对于每一条record
          //如果record的首位为0，则代表该record不可用，跳到下一条record并continue
          //否则根据每个columns的size，找到column_name对应的那个地址起始值
          //读取需求column_name的那个搜索码值
          std::string key = std::string((char *)data);
          //调用BPTree的insert函数插入该record即可
            }
        }
    }
}
```

index的查找

调用`BPTree`的`find`函数查找该`record`即可

index的插入

调用`BPTree`的`insert`函数插入该`record`即可

index的删除

调用`BPTree`的`remove`函数删去该`record`即可

index的修改

调用`BPTree`的`remove`函数删去原先的`record`，再调用`BPTree`的`insert`函数插入修改后的`record`即可