IndexManager

交叉创新平台1604 3160104714 熊苗

一、模块概述

IndexManager的具体要求如下：

- 负责 **B+树索引** 的实现
  - B+树索引的创建
  - B+树索引的删除
  - 等值查询
  - 插入键值
  - 删除键值
  - ***提供相应的接口***
- 索引文件
  - B+树中 **节点大小应与缓冲区的块大小相同**
  - B+树中节点中的数据按照顺序存储
  - B+树的**叉数由节点大小与索引键大小计算得到**
- 优化
  - 基于记录的**B+树重建**
  - 给定双边界或左边界或右边界的**范围查询**

二、主要功能及对外接口

1. 创建索引:

```
bool createIndex(const string & table_name, const string & attribute_name,
const string &index_name, int data_type =  TYPE_INT, int posinRecord=-1,
bool isBeginning = true)throw(Error);
```

传入参数为`table_name`, `attribute_name`, `index_name`, `data_type`, `posinRecord`, `posinRecord`, `isBeginning`. 返回值为bool变量。

若语句执行成功，则输出执行成功信息;若失败，通过throw Error集中管理错误最后打印失败信息。

2. 删除索引

```
bool dropIndex(const string & index_name);
```

传入参数为 `index_name`,返回值为bool变量。

若语句执行成功，则输出执行成功信息;若失败，通过throw Error集中管理错误最后打印失败信息。

3. 删除键值

```
int remove(const string & index_name, const char * raw_data, int
data_type)throw(Error);
```

传入参数为`index_name`,`raw_data`,`data_type`,返回值为欲删除的键值的ID。

若语句执行成功，则输出执行成功信息;若失败，通过throw Error集中管理错误最后打印失败信息。

4. 等值查找

```
int find(const string & index_name, const char * raw_data, int data_type);
```

传入参数为`index_name`,`raw_data`,`data_type`,返回值为查找到的键值的ID。

若语句执行成功，则输出执行成功信息;若失败，通过throw Error集中管理错误最后打印失败信息。

5. 插入键值

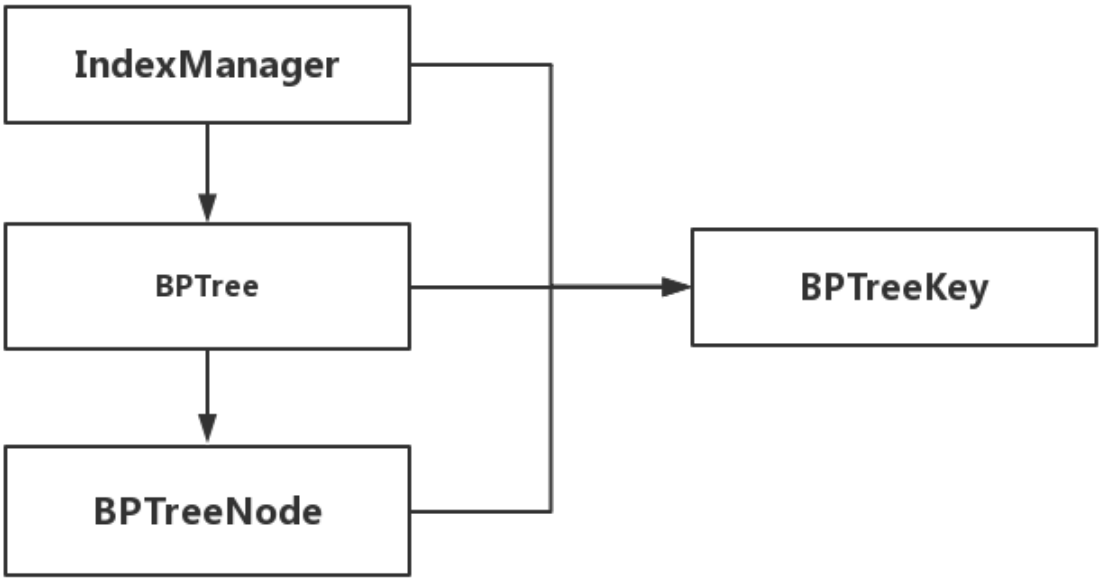
```
bool insert(const string & index_name, const char * raw_data, int
data_type, int recordID)throw(Error);
```

传入参数为`index_name`,`raw_data`,`data_type`,欲插入记录的`recordID`，返回值为bool变量。

若语句执行成功，则输出执行成功信息;若失败，通过throw Error集中管理错误最后打印失败信息。

三、设计思路及框架

`IndexManager`实现的是与文件交互的B+树，每一个节点都是存放在文件里4K大小的`block`。其主要由4个类：`IndexManager`，`BPTree`,`BPTreeNode`,`BPTreeKey`实现，其调用关系如下：



1. 基本名词解释

`entry`：由一个记录的**键值**和它对应的**文件指针**封装成的结点。

`header`：一个单独的固定的模块，用来记录B+树的基本信息。

2. 操作执行路径

1. 当API调用IndexManager时，IndexManager会将传入信息封装成一个`entry`，然后生成一个临时的`BPTree`读取文件进行操作。
2. `BPTree`通过`Index`文件的`headerBlock`获取基本信息，之后从根结点开始一层层向下搜索，直至搜索到叶结点。
3. 每进入一个`Node`，都会先从文件里把结点的相应信息和所有的`entry`读到内存里，然后执行相应的删除查询或者插入操作，最后写会内存。

3. 模块介绍

> ### IndexManager
>
> API和B+树之间的交互平台，整个B+树的对外接口。
>
> 主要用于生成临时的B+树和调用相应的操作。

1. **insert**

```
bool IndexManager::insert(const string & index_name, const char * raw_data,
int data_type, int recordID) throw(Error)
{
    //初始化entry，存储了rawdata的信息
    BPTreeKey entry(raw_data, recordID, data_type);
    //临时建立B+树
```

```
    BPTree* tree = new BPTree("index/" + index_name, data_type);
    //调用树的删除操作
    int ret = tree->insertKey(entry);
    //释放树的空间
    delete tree;

    //错误管理
    if(ret  == BPRepeat)
    {
        Error repeat("This record has exsited in the index!");
        throw repeat;
    }
    return true;
}
```

2. find

```
int IndexManager::find(const string &index_name, const char *raw_data, int
data_type) {

    //调用树的查找操作
    int ret = tree->findKey(entry);
    //错误管理，因为在recordManager查找之前会调用IndexManager去询问是否存在记录
    //所以并不throw而是返回信号
    //如果为-1则记录不存在，否则返回记录的地址
    if(ret  == BPEmpty)
        ret = BPEmpty;
    else
        ret = entry.getPointer();
    return ret;
}
```

3. createIndex

```
bool IndexManager::createIndex(const string & table_name, const string &
attribute_name, const string &index_name, int data_type, int posinRecord,
bool isBeginning) {

    //得到attribute的长度
    int attriLength = Method::getLengthFromType(data_type);
    //调用函数创建文件
    createFile("index/" + index_name);

    //isbegining用来标记建立索引的时间
    //如果isbegining为false，则需要在已有的B+树上进行重建
    if(! isBeginning )
    {
```

```
        BPTree* tree = new BPTree("index/" + index_name, data_type);
        tree->initialIndex(table_name, attribute_name,attriLength,
posinRecord);
    }
    return true;
}
```

4. dropIndex

```
bool IndexManager::dropIndex(const string &index_name) {
    //直接清理掉index所属的文件
    Method::deleteFile("index/" + index_name);
    return true;
}
```

5. remove

```
int IndexManager::remove(const string &index_name, const char *raw_data,
int data_type)throw(Error) {
    //准备返回值
    int recordPointer = 0;
    int* retPointer = &recordPointer;
    //调用树的操作
    int ret = tree->removeKey(entry, retPointer);

    return recordPointer;
}
```

> ### BPTree
>
> 实现B+树的各种操作。

结构图：header，保留block ID = 0；

nodeCount	firstEmptyBlock	firstLeftBlock	mostRightBlock	RootID
记录了B+树的block数目	维护Index文件中第一个为空的block的ID	记录B+树叶节点中最左边一个的blockID，方便范围查询	记录B+树叶节点中最右边一个的blockID，方便范围查询	根结点的Block ID，查找的起点

- 成员变量

```
    string fileName;
    static const int HeaderID = 0;
    int dataType;
```

`HeaderID` 是Index文件的第一个block，地位类似于系统中操作系统的保留寄存器，主要用于存储B+树的各种信息。

`dataType`是API传递给B+树的参数，通过`dataType`可以得到`keyLength`。

- 对外接口

```
public:
    //基本接口
    int insertKey(BPTreeKey & entry);
    int findKey(BPTreeKey & entry);
    int removeKey(BPTreeKey& entry, int * retPointer);

    //范围查询
    void scaleFindLeftEnd(BPTreeKey & entry, vector<ptr>& res);
    void scaleFindRightEnd(BPTreeKey & entry, vector<ptr>& res);
    void scaleFind(BPTreeKey &sta_entry, BPTreeKey &end_entry,vector<ptr>
&res);

    //重建B+树
    void initialIndex(const string & table_name,const string &
attribute_name, int attriLength, int posinRecord);
```

- 辅助函数

```
    int insert(int nodeID,  BPTreeKey& entry);
    int remove(int nodeID, BPTreeKey &entry, int siblingID, bool isLeftSib,
int * retPointer, BPTreeKey & parentKey );
    int getFirstEmptyBlock();
    void updateHeader();
```

`insert` `remove`都是递归函数。实现中对B+树的插入、删除两种基本操作都是基于递归来执行。**递归的使用使得对树结构的维护变得非常方便**。具体实现见第四部分分析。

`getFirstEmptyBlock` 借用了FreeList的思想，通过`firstEmptyBlock`记录了第一个空的`block`，而在第一个空的block的友记录了下一个block的ID。

对B+树的各种信息的维护集成在了`updateHeader`这个函数里。这样做的目的是一次更新所有信息，方便且不易出错。

> ### BPTreeNode
>
> 1个BPTreeNode是B+树的一个结点，实现中采用了4096Byte即一页来存储一个node 。在构造函数时将会通过buffer读取一个完整的node到文件里，析构函数会判断node是否被修改过或者remove，然后把结点写回文件。支持对B+树的结点进行查询删除等各种操作。

isLeaf	nodeSize	nextNodeID	Entry1	Entry2	.....
是否为叶节点	结点中entry的个数	对叶节点，为下一个结点的block ID，方便范围查询；对内子结点，存储第一个指针	第一个entry	第二个entry	

- 成员变量

```
    //利用这两个来标记数据是否需要重新写回buffer
    bool isDirty;
    bool isRemoved;
    //记录key
    vector<BPTreeKey> keys;
```

`nodeCapability`指的是一个node可以存的key的大小， 主要是用来判断是否underflow。

`isDirty`和`isRemoved`用于记录B+树结点的状态，如果结点被修改过或者被删除了，才会选择写回文件或者删除节点，这样方便B+树的IO操作。

`keys`是一个存储`BPTreeKey`的vector，存储了该结点所有的entry，插入和删除记录的操作都是基于内存的操作。

- 构造函数

```
    //用于根据BufferManager传来的Node构建已有的B+树结点
    BPTreeNode(const char *_filename, int _id, int data_type);
    //空结点的初始化
    BPTreeNode(const char *_filename, int _id, int data_type, bool isLeaf,
int nextNodeId);
```

分别为基于非空结点和空结点的初始化。

基于非空结点的初始化将会通过buffer读取一个完整的node到内存里。

基于空结点的初始化将会用过传入参数初始化一个结点，包括对它是否为叶节点等信息赋值。

```
    //结点内查询和维护操作
    int findPosition_UpperBound(const BPTreeKey& key);
    int findPosition_LowerBound(const BPTreeKey& key);
```

查询操作是基于二分搜索的插入，上界函数找到的是比给定key大的entry的位置。下界函数找到的是小于或等于entry的key在vector的位置索引值。

- 插入和删除

```
    int insertEntry(BPTreeKey& entry, int pos);
    int deleteEntry(int pos);
```

`deleteEntry` `insertEntry`主要是基于vector的插入和删除操作。

- 结点判断操作

```
    bool isoverflow() { return nodeSize > nodeCapability; }
    bool isUnderflow(int rootID) ;
```

这两个函数用于判断函数是否需要`split`、`borrow`或者`merge`。

- 核心维护操作

```
    void split(BPTreeKey& entry , int nodeID);
    int borrow(BPTreeKey &entry, BPTreeNode *sibling, bool isLeftSib,
BPTreeKey &parentKey);
    void mergeRightNode(bool isLeftSib, BPTreeNode *sibling, const
BPTreeKey& parentKey, BPTreeKey& entry) ;
```

```
    //范围查询操作
    void mergeVec(vector<ptr>& res, int pos);
    void mergeVecRight(vector<ptr>& res, int pos);
```

对树进行维护操作的几个函数，具体框架和伪代码见第四部分。

> ### BPTreeKey
>
> 对一个Attribute的封装，由key和它对应的pointer组成。

raw_data	pointer
存储了char*数组，长度为keyLength	存储记录所在的blockID

封装的目的

为了避免实现模版类，通过把Key封装在一个类里，重载各种逻辑比较符号，简化B+树里基于key的各种操作，如搜索查询等。

接口

```
class BPTreeKey {
public:
    int getPointer();
    const char* getKeyRawData();
    void setPointer();
    void setKey();

    bool operator > (const BPTreeKey& entry );
```

```
    bool operator < (const BPTreeKey& entry );
    bool operator == (const BPTreeKey& entry );
    bool operator != (const BPTreeKey& entry );
    BPTreeKey& operator = (const BPTreeKey& entry );

private:
    char * rawData;
    int dataType;
    int pointerID;

private:
    int compareKey(const BPTreeKey& key);
};
```

四、关键函数伪代码分析

整体的B+树实现主要有几个特点：

1. 充分利用了递归函数的特性，基于函数出栈进栈FirstInLastOut的特性在返回时修改B+树的内子结点。
2. 充分利用返回值进行状态判断

```
#define BPInsert 1
#define BPDeleteFail 0
#define BPNormal 2
#define BPDelete (-1)
#define BPChange (-2)
#define BPEmpty (-1)
#define BPRepeat (-2)
#define BPChangeEnd (-4)
```

> **B+树查找**
>
> 主要通过`findPosition_LowerBound`这个函数找到对应结点的pointer，通过while循环向下到叶节点，将返回值存储在传进来的`entry`（BPTreeKey&）里。

```
while(true)
{
    //找到记录在当前结点的指针
    //对内子结点，找到的是指向下一层的指针
    //对叶子结点找到的是记录所在的recordID
    pos = node->findPosition_LowerBound(entry);

    //判断是否为叶节点
```

```
    if(node->isleaf() )
    {
        if(find the entry)
        {
            copy the pointer to the returnEntry;
            set mode to Normal;
        } else
            set mode to BPEmpty;
        break;
    } else
        set nodeID to the next level blockID;
}
```

> ### B+树的插入
>
> 递归调用insert函数直到叶节点，执行插入，再通过修改传入的`entry`对内子结点的值进行修改。

针对叶节点和内子结点

1. 未溢出，调用insertEntry，设置返回值BPNormal
2. 溢出，调用split函数，设置返回值BPInsert
3. 重复插入将会报错，设置返回值BPRepeat

```
int BPTree::insert(int nodeID, BPTreeKey& entry) {

    //递归调用
    //如果是叶节点，进入插入状态，如果是内子结点，进入下一层结点的插入函数、
    //到递归回到当前状态即开始修改内子结点的值
    res = node->isleaf() ? BPInsert : insert(next level entry);

    // Check for duplicate
    if (BPRepeat)
        set mode to BPRepeat;
    //插入
    else if (res == BPInsert)
    {
        //把记录插入具体的结点
        res = node->insertEntry(entry, pos+1);
        //判断是否overflow
        if(isoverflow)
        {
            //找到第一个为空的节点
            newID = getFirstEmptyBlock();
            //分裂
            node->split();
            //因为保证分裂出来的node在右侧的
            //在split里面已经执行过新的node的赋值
```

```
            //只需要维护mostRightBlock的记录
            if(nodeID == mostRightBlock)
                mostRightBlock = newID;

            deal specialcase for node == rootID;
            set mode to BPInsert;
        } else if(BPRepeat)
            ret = BPRepeat;
        else ret = BPNormal;
    }
    return ret;
}
```

> ### B+树的删除
>
> 和插入的思路基本相似，主要利用递归和返回值实现。

针对叶节点

1. 未underflow，调用deleteEntry，设置返回值BPNormal
2. underflow：调用兄弟结点
    1. 如果兄弟结点的值等于下界，merge兄弟结点
        1. merge默认是merge右侧结点
        2. 将传进来的entry设置为删除节点的nodeID
        3. 设置返回值为BPDelete
    2. 如果兄弟结点的值大于下界，向兄弟结点借一个entry
        1. 如果是左兄弟，将会修改该结点的父亲值，设置返回值为BPInsertEnd
        2. 如果是右兄弟，将会修改右结点的第一个值，因此也必须修改父节点的值，直接修改传进来的父节点的entry，设置返回值为BPInsertEnd

针对内子节点

1. 未underflow，调用deleteEntry，设置返回值BPNormal
2. underflow：调用兄弟结点
    1. 如果兄弟结点的值等于下界，merge兄弟结点
        1. merge默认是merge右侧结点
        2. 将传进来的entry设置为删除节点的nodeID
        3. 设置返回值为BPDelete
    2. 如果兄弟结点的值大于下界，向兄弟结点借一个entry
        1. 对于内子结点的borrow实际是一个旋转的过程，将第一个结点的key作为父节点的key，将父亲结点的值与第一个指针配对形成新的第一个节点
        2. 如果是左兄弟，将会修改该结点的父亲值，设置返回值为BPInsertEnd
        3. 如果是右兄弟，将会修改右结点的第一个值，因此也必须修改父节点的值，直接修改传进来的父节点的entry，设置返回值为BPInsertEnd

```
int BPTree::remove(int nodeID, BPTreeKey &entry, int siblingID, bool
isLeftSib, int* retPointer, BPTreeKey & parentKey ) {

    //找到结点的位置
    int pos = findPosition_LowerBound(entry);

    //如果是叶节点，res =  BPDelete
    //如果不是叶节点，调用递归函数进入下一层
    int  res = node->isleaf() ? BPDelete : remove(nextLevel);

    Check for duplicate;
    if (res == BPDelete) {
        //开始删除entry
        //如果需要删除结点：总是删除右边节点
        //分成叶节点和非叶节点处理
        if (isleaf) {
            if (entry != the node to be deleted)
                ret = BPDeleteFail;

            //存储欲删除的key的pointer
            //方便recordManager删除，用于优化
            *retPointer = node->getPointer();
            res = node->deleteEntry;
        } else {
            //对内子结点，找到需要删除的entry
            pos = node->findPosition_LowerBound(entry);
            res = node->deleteEntry(pos);
        }

        //处理根结点的特殊情况
        if(nodeID == RootID)
        {
            if(B+Tree is empty ) {
                remove the tree to the initial state;
            } else if (the root is only a pointer){
                reset the root to its child;
            }
        }

        //删除没有出错并且underflow
        if(isUnderflow())
        {
            //borrow from siblings
            if(sibling NodeSize() > lowerBound)
            {
                ret = node->borrow();
            }else if(sibling NodeSize() == UpperBound())
            {
                node->mergeRightNode();
```

```
                removeBlock();

                check the mostRightBlock;
                ret = BPDelete;
            }

        } else if(DeleteFail)
            ret = BPDeleteFail;
        else if(leaf node delete influence the internal node )
        {
            if(isLeftSib)
            {
                parentKey.setKey();
                ret = BPChangeEnd;
            } else ret = BPNormal;
        }
        else
            ret = BPNormal;
    }else if(res == BPChange)
    {
       change internal node;
    }else if(res == BPChangeEnd)
    {
        node->setDirty();
        ret = BPNormal;
    } else if (DeleteFail)
    {
        ret = BPDeleteFail;
    }
    return ret;
}
```

五、测试情况

> 测试流程

1. 顺序插入100000-200000个结点
2. 利用`DebugPrint`函数打印整个B+树检测插入情况
3. 再随机插入，检测报错情况
4. 随机删除B+树，重点测试corner case
5. 批量删除整个B+树，打印结果
6. 再次插入100000个结点，检测删除是否使B+树回到初始状态

> 表现

目前测试情况直到200000仍然完全正确。

```
//递归打印整个B+树
```

```
void BPTree::debugPrint(int id)
{
    BPTreeNode* node = new BPTreeNode(fileName.c_str(), id, keyLength);
    //打印header信息
    cerr << "nodeSize =  " << node->getNodeSize() << endl;
    cerr << "Block id = " << id << ", isLeaf = " << node->isleaf() << endl;
    //keys
    cerr << "Keys:";
    for (int i = 1; i <= node->getNodeSize(); i++)
    {
        const char* k = node->getEntry(i).getKeyRawData();
        cerr << (k) << " - ";
    }
    cerr << endl;
    //打印Pointer
    cerr << "Pointers: ";
    for (int i = 0; i <= node->getNodeSize(); i++)
    {
        cerr << " " << node->getEntry(i).getPointer();
    }
    cerr << endl;

    if (!node->isleaf())
        for (int i = 0; i <= node->getNodeSize(); i++)
            debugPrint(node->getEntry(i).getPointer());

    delete node;
}
```