**Index Manager说明文档**

**作者：吴世奇 计算机科学与技术07-03**

**IndexManager模块设计：聂森 吴世奇**

**一．模块分析：**

Index manager是程序的索引部分,直接对buffer manager提供的内存索引块操作，主要是构建B＋树，负责实现API manager这一模块（提供函数接口）所需要的函数。功能有：存储\删除记录的索引，查找记录在table表中的相对位置等（由buffer负责写回磁盘）。

1. **设计思路：**

我们将B+树的扇出系数FAOUT定位3，即每个树节点可以存两个key值，3个指针，这和老师要求的不一样，老师要求我们将每个节点大小设为一个block。节点的构造如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pointer[blo][off] | KeyType\* key | Pointer[blo][off] | KeyType\* key | Pointer[blo][off] |

↑ ↑ ↑ ↑ ↑

指向左节点 指向第一个key值 指向中间节点 指向第二个key值 指向右节点

指向子节点，我们是通过一个二维数组实现的，一维记录它所在的block数，二维记录这个子节点在这个block中的偏移量，这样我们就很容易找到它的子节点们。当为叶节点时，此时的block和offset记录的是文件中记录的block数和offset，这样我们就可以通过叶节点找到记录。

和数据文件一样，我们专门为每个B+树建立一个文件，也是实行block化，即一个一个block。每个block中用offset记录偏移。每插入一个节点，我们就将新插入的节点写入到这个B+树文件中，记录下这个节点所在的block和它在这个block中偏移量。当我们需要索引时，就从这个表所对应的B+树中读入这些节点。读到buffer中，这样就很容易通过B+树找到我们要搜寻的记录。

1. **类和函数介绍：**

下面我们将介绍一些这个模块中比较重要的类和函数，以体现出indexmanager这个模块具有的功能。首先是Node这个类：

**class Node**

{

**bool m\_bLeaf;** //表示是否为叶子节点

**int m\_iCount;** //表示已经存入节点中的key数

**TKeyType\* m\_Key[FANOUT-1];** //表示已存入key的值

**int NodeSize;** //表示每一个节点的大小

**int m\_iPointer[FANOUT][2];** //表示节点的各个子节点地址

**int m\_iFatherPosition[2];** //表示节点的父节点地址

**int m\_iSelfPosition[2];** //表示节点自己的地址

**int m\_iKeyTypeCount;** //每个key值的分块数

**CFile\* m\_file;** //表示节点所在的文件

**BTree\* m\_BTree;** //表示节点所在的树

FANOUT定义为3.。数组m\_Key中存放key的值，所以可以存放两个key值。m\_iKeyTypeCount表示数据类型的长度，对于int和float类型，长度即为1，但是对于char类型，则根据建表时的情况而定。数组m\_iPointer表示子节点的地址，所以共有3个子节点的地址，表示block数中的偏移量。数组m\_FatherPosition表示父节点的地址，m\_iSelfPosition表示节点自己在那个block中的偏移量。

一旦数据类型确定，那么这个节点大小NodeSize即确定。

NodeSize = **sizeof**(TKeyType) \* m\_iKeyTypeCount \* (FANOUT - 1) //节点的大小

**+ sizeof**(**int**) \* (FANOUT + 1) \* 2 + 1; //3个子节点加1个父节点

B+树的操作其实是建立在对节点的操作上的。

1. 节点构造函数

**template <class TKeyType>**

Node<TKeyType>::Node(**int TypeCount, BTree\* Tree)**

其中TypeCount表示数据长度，Tree表示这个节点所在的树。在这个构造函数中，我们将所有的数据都初始化，将m\_count值为-1，所属文件m\_file值为空，要等到建立文件时，再将文件名赋给它。将m\_iPointer、m\_iFatherPosition、m\_iSelfPosition的值都值为-1。因为我们在向B+树中插入一个节点时，首先要建立一个空的节点，并对其进行初始化，并为节点开辟存储空间：

**for**(**int j = 0; j < FANOUT - 1; ++j)**

**m\_Key[j] = new TKeyType[m\_iKeyTypeCount];**

同时，我们根据数据类型，计算出节点的大小。

2.获得节点第index个子节点的地址

**template <class TKeyType>**

**void Node<TKeyType>::GetPointer(int Index, int**& PositionBlock, **int**& PositionOffset)

知道一个节点，那么我们很有必要知道它的子节点的地址，因为树的插入和删除都要用到。这里index表示这个节点的第几个节点，index只能取0、1、2，PositionBlock和PositionOffset是我们想要得到的值，通过引用来改变他们的值。我们已经知道，数组m\_iPointer是存放子节点地址的，所以只需将数组m\_iPointer中的值对应的赋给PositionBlock和PositionOffset即可。如果index的值不是0、1、2，那么我们就将PositionBlock和PositionOffset的值设为-1.

3.设置节点第index个子节点的地址

**template <class TKeyType>**

**void Node<TKeyType>::SetPointer(int Index, int PositionBlock, int PositionOffset)**

和前面相反，这是将传入的PositionBlock和PositionOffset赋给数组m\_iPointer，从而获得子节点的地址。

4.获得节点储存的key值

**template <class TKeyType>**

TKeyType\* Node<TKeyType>::GetKey(**int Index)**

我们需要得到节点中的key值，这里index只能取0或1。返回指向key的指针，其实是数据数组的头地址。

1. 设置节点储存的key值

**template <class TKeyType>**

**void Node<TKeyType>::SetKey(int Index, TKeyType\* pKey)**

有的时候，我们需要改变节点中key的值。这里index同理只能取0或1，pKey为新的值。

**for**(**int i = 0; i < m\_iKeyTypeCount; ++i)**

**m\_Key[Index][i] = pKey[i];** //设置key值为pKey

1. 获得节点前一个节点的地址

**template <class TKeyType>**

**void Node<TKeyType>::GetFrontNodePosition(int**& Block, **int**& Offset)

在进行插入、删除时，我们往往需要知道节点的前一个节点，以便于拆分节点或者合并节点。当我们找不到这样的节点时，则将block和offset都值为-1，表示不存在。这里block和offset都是引用，通过函数，我们来改变他们的值，方便快捷。

1. 获得拥有相同父节点的前或后一个节点的地址

**template <class TKeyType>**

**bool Node<TKeyType>::GetEqualLevelNodePosition(int**& Block, **int**& Offset)

同理，这个函数也是为我们的B+树的插入、删除服务。

1. 获得父节点中该子节点的index值

**template <class TKeyType>**

**int Node<TKeyType>::GetFatherLinkToMeIndex()**

这个函数是为上面找兄弟节点的函数服务的。我们要知道这个节点是他父亲的第几个孩子。只有这样，我们才能找到他的兄弟节点是第几个孩子。

1. 读取节点信息

**template <class TKeyType>**

**void Node<TKeyType>::Read()**

这个函数是非常重要的几个函数之一。它是将节点从B+树文件中读出来。

m\_file->Seek(m\_BTree->GetSize() + NodeSize \* Offset, CFile::begin); //文件 指针移至所要读取的节点地址

Offset是节点偏移个数，通过Offset\*NodeSize+b\_Btree->GetSize()，我们就很容易得到我们想要读的节点地址，将这个节点从文件中读出来。

10.组织节点信息

**template <class TKeyType>**

**char**\* Node<TKeyType>::OrganizeNode()

这个函数是将我们要读或写的信息依次组织好，比如先写节点的子节点地址，再写key值，最后写父节点的地址。将组织好的字符串的首地址返回。

1. 插入key值

**template <class TKeyType>**

**void Node<TKeyType>::InsertKey(TKeyType\* pValue, int Block, int Offset, bool AutoSave)**

这个函数在B+树插值中用到。例如一个节点只有一个key值，还可以再插入一个key值，或者向一个新开辟的节点中插值等等。pValue是要插入值的指针，Block和Offset是这个node的地址。

1. 删除key值

**template <class TKeyType>**

**void Node<TKeyType>::DeleteKey(int Index, bool AutoSave)**

这在B+树删除中用到，index表示要删除这个节点的第几个值，index只能取0或1.因为一个节点最多只有两个值。

1. 添加新的节点

**template <class TKeyType>**

**void Node<TKeyType>::AddNode()**

这个函数的功能是将已经建立的节点写进文件中，即储存节点信息

1. 删除节点中所有key

**template <class TKeyType>**

**void Node<TKeyType>::FreeNode()**

供其他函数调用。有的时候我们需要将整个节点清空，比如删除这个节点，需要将内存空间释放。

1. 删除节点

**template <class TKeyType>**

**void Node<TKeyType>::Drop()**

这个函数为B+树的删除节点所调用，同时处理和他周围的节点的关系。

1. 更新节点

**template <class TKeyType>**

**void Node<TKeyType>::UpdateNode()**

当节点中的值改变后，这个节点需要被重新写入文件中，即更新文件中节点的信息。

1. 更改节点中的key值

**template <class TKeyType>**

**void Node<TKeyType>::ChangeValue(TKeyType\* pSrcValue,TKeyType\* pNowValue)**

这个函数调用前面的的函数，将新值写入节点中。

**接下来是B+树的定义和操作，它们都将用到节点操作函数。**

首先是B+树类的声明：

**class BTree : public CObject**

{

**CString m\_sKeyType;** //树中key值的类型

**int m\_iKeyTypeCount;** //每个key值的分块数

**int m\_iRoot;** //根节点地址

**int m\_iFirstLeaf;** //首叶子节点地址

**int TreeSize;** //树的大小

**int MaxOffset;** //文件大小

**int EmptyOffset;** //第一个空闲段的地址

**CObArray m\_aReaders;** //

**CFile\* m\_file;** //树所在的文件指针

BTree类主要有以下几个函数，用来实现对B+树的操作。

1. 向B树中插入新的记录

**void BTree::InsertValue(void**\* pValue, **int Block, int Offset)**

这主要调用node类中insertKey函数，pValue是要插入值的首地址，Block和Offset是所对应文件记录的Block和Offset，不是节点的block和Offset，要区分清楚。

1. 删除B树中的记录

**void BTree::DeleteValue(void**\* pValue)

当从文件中删除一条记录时，我们要从B+树中对应的将这个key值删除，pVlaue是要删除的key值首地址。

1. 删除B树中的节点

**void BTree::DropNode(int Block, int Offset)**

当删掉某些key值之后，或者做完一些节点操作后，有些节点会变为空节点，我们有必要将这种节点从B+树中删除。

1. 删除B树

**void BTree::Drop()**

将整个B+树删除，比如删除表之后，我们需要将对应的索引一并删除。

1. 寻找文件中的空余位置

**void BTree::FindEmptyOffset(int**& Offset, **int NodeSize)**

这是为向文件中写入新节点做准备。找到之后，通过引用改变Offset的值，Offset即为要写入文件的首地址。

1. 创建B树

**void BTree::CreateBPlusTree()**

当为一个表创建索引时，我们需要首先建立一个B+树。将其中的所有变量都初始化。

1. 读取B树的信息

**void BTree::Read()**

将B+树的信息从文件中读出来，主要包括m\_iRoot(根节点地址)、m\_iFirstLeaf（首叶子节点地址）、MaxOffset（文件大小）、EmptyOffset（第一个空闲段的地址）。

1. 查找B树中key值相同的记录地址

**bool BTree::FindValue(void**\* pValue, **int**& Block, **int**& Offset)

这个函数主要用来支持select \* from table where value=？语句。即在B+树中查找给定的值，当查找到以后，改变Block和Offset的值，指向record。为了快速查到要找的记录。

**四、IndexManager的主要功能：**

1.创建索引 ： 首先调用CreateBPlusTree（）函数创建一个B+树，然后调用InsertValue函数往B+树中一个一个插值。

2.删除索引值： 调用DeleteValue函数。

3.插入索引值： 调用InsertValue函数。

4.查找某一记录的位置：当要查找某一记录的记录偏移量时，调用FindValue函数实现。

5.查找某一批记录的位置（<,<=,>,>=）

由于时间原因，这个功能我们还没有实现。