数据库系统设计-MiniSQL

indexManager模块详细设计报告

黄昭阳 3130000696 - 11/6/15

indexManager模块总体设计

一、实验要求：

Index Manager负责B+树索引的实现，实现B+树的创建和删除（由索引的定义与删除引起）、等值查找、插入键值、删除键值等操作，并对外提供相应的接口。

B+树中节点大小应与缓冲区的块大小相同，B+树的叉数由节点大小与索引键大小计算得到。

二、实验完成情况：

完整实现实验要求的所有功能。

1. indexManager模块中总共有三个类，分别实现不同的逻辑功能：

indexManager：	负责B+树与API之间接口的命令衔接。
bPlusTree：	负责B+树所有节点的整体管理，以及B+树中对root的特殊处理。
bPlusTreeNode：	负责B+树节点所有的逻辑功能。

2. API为indexManager提供一个委托，用来向API传输通过与index有关的条件得到的节点，以便接下来对record的相关内容进行操作：

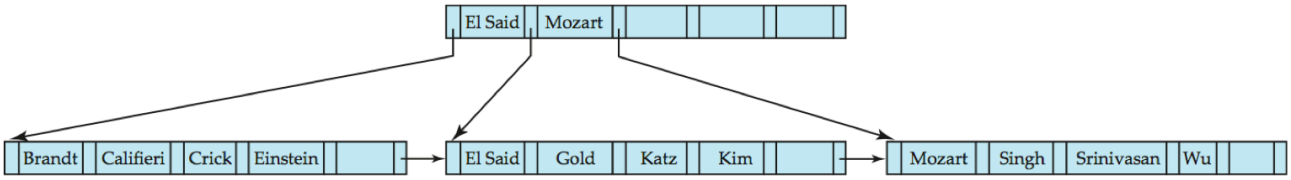
```
class basicDelegate {
public:
    virtual SQLTable* getTableInfo(string* tableName)=0;
    virtual SQLIndex* getIndexInfo(string* indexName)=0;
    virtual string* getRecordFileName(string* tableName)=0;
    virtual string* getIndexFileName(string* indexName)=0;
};  //基层委托，获取需要的相关信息

class indexDelegate:public basicDelegate {
public:
    virtual int selectedRecordAddressFromIndex(vector<int>&
recordAddress,vector<SQLCondition*>& condition)=0;
        //通过这个委托传输给API符合要求的用index筛选出来的record地址，同时传出剩下的其他条件，让record通过这些条件从中再次筛选出符合要求的记录来输出

    virtual int deleteRecordAddressFromIndex(vector<int>&
recordAddress,vector<SQLCondition*>& condition)=0;
        //通过这个委托传输给API符合要求的用index筛选出来的record地址，同时传出剩下的其他条件，让record通过这些条件从中再次筛选出符合要求的记录来删除
};
```

3. indexManager提供带条件的删除(以SQLCondition的形式传入)、插入、待条件的查找三种功能，完美完成实验要求。

4. B+树结构设计如下图所示，每一个节点包含N个值和N+1个指针，其中叶子节点的第N+1个指针用来指向它右侧的下一个叶子节点，这样进行不等值查询时将会很大程度的提升效率。



indexManager模块整体详细分析

一、indexManager整体结构分析和代码介绍：

```
class indexManager {
private:
    static indexDelegate* m_delegate;
//委托
public:
    static int deleteIndex(string* indexName,SQLData* data);
//等值删除
    static int insertIndex(string* indexName,SQLData* data,int
recordAddress);
//插入
    static int deleteIndexWithCondition(string* indexName,SQLCondition*
condition,vector<SQLCondition*>& otherConditions);

 /*————————————————————————————————————————————————*/
 /*    带条件的删除，其中condition为indexManager筛选所依靠的条件，而          */
 /*  otherConditions将通过委托deleteRecordAddressFromIndex()来传送给API */
 /*  一共record进行进一步筛选时作为筛选条件。                                 */
 /*————————————————————————————————————————————————*/

    static int selectIndexWithCondition(string* indexName,SQLCondition*
condition,vector<SQLCondition*>& otherConditions);

  //类似deleteIndexWithCondition

    static void setDelegate(indexDelegate* delegate)
    {
        m_delegate = delegate;
    }
};
```

二、indexManager的使用方法

API调用indexManager来select用户想要的记录的流程是这样的：

1. 通过interpreter获得select的所有条件，在所有条件中找到一条是可以使用index来进行查询的，将其单独抽离出来作为condition，其他的条件放到容器otherConditions内，用来提供给record进行进一步筛选。这样就可以调用

indexManager::selectIndexWithConditions(indexName,condition,otherConditions);

来进行筛选。

2. indexManager接收到select命令，将通过condition来筛选得到所有符合要求的record记录的地址，将他们放到容器recordAddress内，然后通过委托

m_delegate->selectRecordAddressFromIndex(recordAddress,otherConditions);

传回给API。

3. API接收到来自indexManager的select委托后，将再次调用recordManager的相关接口来实现record内部的进一步筛选，然后进行输出。

API调用indexManager的deleteWithCondition时与select类似，indexManager进行初步筛选，通过API调用recordManager的相关接口进行进一步的精确筛选之后将再次调用indexManager的等值删除，即deleteIndex(indexName,data)进行删除，deleteWithCondition内将不会直接对index的值进行删除。

bPlusTree模块整体详细分析

一、bPlusTree整体结构分析和代码介绍：

```
class bPlusTree
{
public:
    bufferManager* buffer;
  //一颗B+树的所有节点必定都存在同一个文件里面，因此拥有一个buffer来进行文件管理

    bPlusTreeNode* root;
  //root作为B+树的根，bPlusTree将通过它来管理整棵B+树

    SQLDataType dataType;
  //记录这颗B+树的叶子节点的数据的数据类型，SQLDataType为自己设计的底层数据结构

    int dataSize;
  //用来记录B+树数据的大小

    int MaxSizePerNode;
  //根据B+树单个树节点的大小和一页4KB的大小，能够计算出每一个节点内最多有多少个数据

    bPlusTree(bufferManager* buffer,SQLDataType dataType, int dataSize);
  //构造函数

    ~bPlusTree();
    int insertData(SQLData* data,int recordAddress);
  //SQLData为自己设计的底层数据结构，包括SQLDataInt，SQLDataFloat，
SQLDataChar，使用多态的技术实现这些操作，详细内容将在总体设计报告中呈现，
recordAddress为伴随这个SQLData在record的地址。

    int deleteData(SQLData* data);
  //等值删除

    int findEqualTo(SQLData* data);
  //等值查找

    int findLargerThan(SQLData* data,vector<int>& recordAddress);
    int findLessThan(SQLData* data,vector<int>& recordAddress);
  //不等值查找，将查找到的符合要求的数据存在recordAddress内

    void printTree();
  //打印整棵树的相关信息
};
```

二、bPlusTree的使用方法：

bPlusTree的构造函数接受三个参数，其中：

bufferManager* buffer：　用来指定这颗B+树对应的文件

SQLDataType dataType：　用来指定这颗B+树的数据类型

int dataSize：　用来指定这颗B+树单个数据的数据大小

其他的函数功能都一一与函数名对应，这里提供一个使用样例进行说明：

```
            bufferManager* buffer = new bufferManager(indexFileName);
  //获取一个管理index文件的buffer，indexFileName为一个字符串
            bPlusTree* tree = new bPlusTree(buffer,(*itr)->attributeType,
(*itr)->attributeSize);
            vector<int> recordAddress;
            SQLData* data;
            switch (condition->data->type) { //condition为B+树使用的条件
                case INT:
                    data = new SQLDataInt(*condition->data);
                    break;
                case FLOAT:
                    data = new SQLDataFloat(*condition->data);
                    break;
                case CHAR:
                    data = new SQLDataChar(*condition->data);
                    break;
                default:
                    break;
            }
            if( *condition->op == string("==") )  //判断属于哪一种条件
            {
                cout << tree->findEqualTo(data);
                return 0;
            }
            if( *condition->op == string("<") )
            {
                tree->findLessThan(data,recordAddress);
                printAddress(recordAddress);
                return 0;
            }
            if( *condition->op == string(">") )
            {
                tree->findLargerThan(data,recordAddress);
                printAddress(recordAddress);
                return 0;
            }
```

bPlusTreeNode模块整体详细分析

一、bPlusTreeNode整体结构分析和代码介绍：

```
class bPlusTreeNode
{
public:
    SQLDataType dataType;
    bufferManager* buffer;
    void* block;
    int MaxSizePerNode;
    int validLength;
    int isLeaf;
    int dataSize;
    vector<int> pointers;
    vector<SQLData*> data;
  //以上为bPlusTreeNode内部的相关数据

    int chooseIndex(SQLData* data);
  //当当前树节点不是叶子节点时，此函数根据data的值选择出下一次应当往哪一个分叉走

    SQLData* getMinimalData();
  //得到当前分支下的最小的节点值，当一个节点需要分裂时会使用此函数

    static bPlusTreeNode* constructNodeFromBuffer(void*
buffer,SQLDataType dataType,int dataSize,bufferManager* buffermanager);
  //从一块4K大小的缓存区中，根据给定的相关元素来构造出一个bPlusTreeNode

    static int writeNodeIntoBuffer(bPlusTreeNode* node,int pageNumber);
  //将一个bPlusTreeNode根据一定的数据格式写到当前bPlusTree对应文件的第pageNumber
页内

    bPlusTreeNode(void* block,int isLeaf,int MaxSizePerNode,SQLDataType
dataType,int dataSize,bufferManager*
buffer):block(block),MaxSizePerNode(MaxSizePerNode),isLeaf(isLeaf),validL
ength(0),dataType(dataType),dataSize(dataSize),buffer(buffer)
  //构造函数，根据给的相关信息构造一个新的bPlusTreeNode节点

    ~bPlusTreeNode()
  //析构函数，将相关的指针全部释放掉

    int appendData(SQLData* data,int pointer);
  //将一组数据放到当前树节点的数据区的尾部

   static bPlusTreeNode* startUseTreeNode(int pageNumber,SQLDataType
dataType,int dataSize,bufferManager* buffer);
  //在index文件中一个4K的页对应一个树节点，此函数接受一个bufferManager* buffer来了
解是哪一个文件，通过pageNumber知道是此文件的第几页，然后根据相关参数将此页的内容构造成
一个bPlusTreeNode传出
```

```
    static void finishUseTreeNode(bPlusTreeNode* node,int pageNumber);
  //类似startUseTreeNode函数，将node的数据根据某种格式写到其对应文件的第pageNumber页，然后析构掉node节点，结束对该节点的使用

    int getValidLength();
  //获得当前节点分叉数的个数

     void setValidLength(int len);
  //将当前节点的分叉数置为len个，len之后的分叉将被删除，在分割操作时将调用此函数

     int mergeIndex(int index);
  //当前节点的第index分叉的值已经不满足B+树的性质了（因为之前在这一分叉下面曾经删除过一个叶子节点），所以需要将index分叉与index+1分叉或index-1分叉进行合并

     int splitIndex(int index);
  //当前节点的第index分叉的值已经不满足B+树的性质了（因为之前在这一分叉下面曾经插入过一个叶子节点，或者曾经在该分叉上进行了合并），所以需要将第index分叉进行平均分割

     int insertIntoNode(SQLData* data,int recordAddress);
  //将一组数据插入到B+树的叶子中，并对受到影响的B+树节点进行调整

     int deleteFromNode(SQLData* data);
  //将一个数据从B+树叶子节点中删除，与之对应的recordAddress也会被删除，并对受到应先的B+树节点进行调整

     int findEqualTo(SQLData* data);
  //等值查找

     int findLargerThan(SQLData* data,vector<int>& recordAddress);
  //由于B+树是有序树，此函数的做法是找到最左端符合要求的值找到并放到recordAddress中，然后利用B+树底层叶子节点之间的指针将从该节点起，知道最右端的叶子节点的值全部取出

     int findLessThan(SQLData* data,vector<int>& recordAddress);
  //由于B+树势有序数，此函数的做法是从B+树的最左端一个叶子开始向右扫描，直到遇到一个不符合条件的值，将此期间所有的值全部放入recordAddress中

     int isApproriate();
  //判断当前节点的分叉数量是否满足B+树的性质

     void printTree(int tabs);
  //将整棵B+树的信息全部输出到屏幕

};
```

二、bPlusTreeNode的使用方法

这里介绍插入数值的方法，删除与此类似。

```
insertIntoNode(SQLData* data,int recordAddress) //接收需要插入的一组数据
{
    int index;
    for(index = 0;index<validLength;index++)
        if(*(this->data[index])>=*(data)) break;
       //找到需要插入哪一个分叉
    if(!isLeaf)  //不是叶子节点则进入此分叉进行下一次插入
    {
        bPlusTreeNode* node =
startUseTreeNode(pointers[index],dataType,dataSize,buffer);
        //得到该指针对应位置的B+树节点

        if (node->insertIntoNode(data,recordAddress)==0) { //插入失败则返回0
            finishUseTreeNode(node, pointers[index]);
            return 0;
        }
        if (node->getValidLength()+1>MaxSizePerNode) {
               //插入使得分叉增加超过上限而不再满足B+树的性质了
            finishUseTreeNode(node, pointers[index]);
               //注销掉这个节点的使用
            return splitIndex(index); //分裂这个节点的第index叉
        }
        finishUseTreeNode(node, pointers[index]);
        //注销掉这个使用的节点
    }
    else {
        //否则直接将这组数据插入这个叶子节点
        this->data.insert(this->data.begin()+index,data);
        this->pointers.insert(this->pointers.begin()+index,
recordAddress);
        validLength++;
    }
    return 1;  //插入成功返回1
}
```

模块关键函数详细介绍与分析

一、indexManager类的关键函数详细介绍

1. deleteIndexWithCondition(string* indexName,SQLCondition* condition,vector<SQLCondition*>& otherConditions);

函数分析：

其中condition为indexManager筛选所依靠的条件，而otherConditions将通过委托deleteRecordAddressFromIndex()来传送给API一共record进行进一步筛选时作为筛选条件，record将所有符合用户要求的记录筛选出来后再传回API哪些是真实需要被删除的记录，API根据传回来的值调用等值删除deleteIndex来确切删除对应的值。通过这样的筛选->再筛选->真实删除的结构来完成整个删除的业务逻辑。

2. static int selectIndexWithCondition(string* indexName,SQLCondition* condition,vector<SQLCondition*>& otherConditions);

函数分析：

具体过程类似deleteIndexWithCondition函数，只是最后一步删除修改为输出到屏幕

二、bPlusTree类的关键函数详细分析

1. int insertData(SQLData* data,int recordAddress);

函数分析：

bPlusTree的insertData函数需要对root节点作为特殊处理，其中root的情况又分为：

a. root为NULL，即树内没有任何节点，需要新建一个叶子节点作为root。

b. root为叶子节点，此时插入的数值将直接被插入root中，同时root插入节点后存在分裂的可能，分裂时root需要被设置为非叶子节点，整棵树将增加一层。

c. root为非叶子节点，此时插入的数值将进入root下的分叉之中，同时root插入后也存在分裂的可能

2. int deleteData(SQLData* data);

函数分析：

bPlusTree的deleteData函数需要对root节点作为特殊处理，其中root的情况又分为：

a. root为NULL，即树内没有任何节点，删除失败。

b. root为叶子节点，此时删除可能导致整棵树内无任何有效值，需要删除root

c. root为非叶子节点，此时删除操作需要进入root分叉的下一层中，同时删除操作可能导致root的某些叉需要合并

3、 int findEqualTo(SQLData* data);
int findLargerThan(SQLData* data,vector<int>& recordAddress);
int findLessThan(SQLData* data,vector<int>& recordAddress);

函数分析：

直接调用bPlusTreeNode内部的相关函数即可

三、bPlusTreeNode类的关键函数详细分析

1. int insertIntoNode(SQLData* data,int recordAddress);

函数分析：

节点的插入分为叶子节点插入和非叶子节点插入两种，当插入导致节点内分叉数膨胀超过上限时，需要被分成两个部分，这个分割操作split将被其父节点来操作。同时非叶子节点的插入可能会导致节点内关键字需要更新。

2、 int deleteFromNode(SQLData* data);

函数分析：

节点的删除分为叶子节点删除和非叶子节点删除两种，当删除导致节点内分叉数过少超过下线时，需要将它与其相邻的节点进行合并，同时合并可能导致合并后的节点又超过节点分叉数过多超过上限，需要再执行一次分离操作，这些merge和split操作都将被其父节点来操作。同时这些删除、合并、分离的操作会导致节点内关键字需要更新。

3、 int mergeIndex(int index);
int splitIndex(int index);

函数分析：

这两个操作内部实现机制十分复杂，同时细节繁多，两种操作的方式都需要严格分为叶子节点和非叶节点两种，这里举一个例子。

```
int bPlusTreeNode::mergeIndex(int index)
{
    if(index == validLength) index--;  //当index时随后一叉，则往前挪一个
    bPlusTreeNode* node1 =
startUseTreeNode(pointers[index],dataType,dataSize,buffer);
    bPlusTreeNode* node2 = startUseTreeNode(pointers[index
+1],dataType,dataSize,buffer);
//将node2合并进入node1中，最终node2将被舍弃
    if(node1->isLeaf == 0)
        node1->appendData(node2->getMinimalData(),node2->pointers[0]);
    else node1->pointers.erase(node1->pointers.begin()+node1-
>validLength);
//分是否是叶子节点两种情况将一组值并入node1的末尾
    int increment = 1 - node1->isLeaf; //用以区分是否是叶子节点两种情况处理
    for(int i=0;i<node2->getValidLength();i++)
    {
        switch (dataType) {
           . . .
          //node1->appendData(new SQLDataInt(*(node2->data[i])),node2-
>pointers[i+increment]);
        }
    }
    if(node1->isLeaf) node1->pointers.push_back(node2->pointers[node2-
>validLength]); //当node1是叶子时，尾指针应当指向node2的下一个叶子节点
    finishUseTreeNode(node2, pointers[index+1]);
    delete data[index];
    data.erase(data.begin()+index);
    //删除node2节点对应的值，此时这个值已经没有意义了

    pointers.erase(pointers.begin()+index+1);
    //删除当前节点分叉到node2节点处的那个指针，此时这个值也已经没有意义了
    validLength--;
    //合并后当前节点长度将会减一

    if (node1->getValidLength()+1 <= MaxSizePerNode) {
        finishUseTreeNode(node1, pointers[index]);
        return 1;
    } else {
        finishUseTreeNode(node1, pointers[index]);
        return splitIndex(index);
    //如果合并后node1节点的长度超过上限，则需要再做一次分离
    }
}
```

4. int findEqualTo(SQLData* data);
int findLargerThan(SQLData* data,vector<int>& recordAddress);
int findLessThan(SQLData* data,vector<int>& recordAddress);

函数分析：

由于B+树是有序树。

findEqualTo函数的将根据标准的B+树查询办法进行查询。

findLargerThan函数的将找到最左端符合要求的值找到并放到recordAddress中，然后利用B+树底层叶子节点之间的指针将从该节点起，直到最右端的叶子节点的值全部取出。

findLessThan函数的做法是从B+树的最左端一个叶子开始向右扫描，直到遇到一个不符合条件的值，将此期间所有的值全部放入recordAddress中。

这里举一个例子进行说明

```
int bPlusTreeNode::findLargerThan(SQLData* data, vector<int>&
recordAddress)
{
    if(isLeaf) //当前节点已经是叶子节点，将直接判断是否需要取值
    {
        for(int i=0; i<validLength ; i++) //将满足要求的值全部放入容器中
            if(*this->data[i] > *data )
recordAddress.push_back(pointers[i]);
        if(pointers[validLength] == -1)
     //为-1说明这已经是最右方的叶子节点，直接退出即可
        {
            delete data; //删除data，防止内存泄露
            return 0;
        }
        bPlusTreeNode* p = startUseTreeNode(pointers[validLength],
dataType, dataSize, buffer);  //否则将要进入当且叶子节点右侧的叶子节点
        p->findLargerThan(data, recordAddress);
        finishUseTreeNode(p, pointers[validLength]);
              //访问完成后退出
        return 0;
    }
  //否则不是叶子节点的话讲进行选择进入当前节点的哪一叉
    int index = chooseIndex(data);
    bPlusTreeNode* p = startUseTreeNode(pointers[index], dataType,
dataSize, buffer);
    p->findLargerThan(data, recordAddress);
  //递归查找
    finishUseTreeNode(p, pointers[index]);
    return 0;
}
```

5. static bPlusTreeNode* constructNodeFromBuffer(void* buffer,SQLDataType dataType,int dataSize,bufferManager* buffermanager);

static int writeNodeIntoBuffer(bPlusTreeNode* node,int pageNumber);

函数分析：

这两个函数的作用是对bPlusTreeNode和一个4K的buffer进行互相转换，分别是通过一个buffer构造出一个bPlusTreeNode和将一个bPlusTreeNode写到指定的buffer中内部实现原理与数据库设计无关，这里不进行赘述。

indexManager模块总结

1. 对B+树内部的具体实现有了一个详细的了解，实现了B+树支持的所有操作，如：删除、插入、查询。其中查询包括等值和不等值查询两类。

2. 在B+树与bufferManager进行衔接的过程中，发现了一个与数据库系统设计中类似的”脏数据“问题，B+树节点内容从一个buffer来构建，如果此时对节点内容进行修改而又从该buffer建立了一个节点，将会产生数据不同步的问题，为了解决这个问题，我每次在需要再次构建一个新的树节点时，都会将原来的树节点内容与buffer进行同步并释放掉原来的树节点。

3. B+树的删除、插入两个操作将会导致两个附加操作：合并、分割，这两个函数的实现十分的繁琐，且细节部分十分之多，稍不小心就会产生不清楚的bug，但是实现成功这两个函数并通过了大量的测试之后，对B+树的整体了解又上升了一个层次。

4. 将indexManager模块分成三个类：indexManager，bPlusTree，bPlusTreeNode三个类来实现，使得层次结构更加分明，在后来将各个模块合成整体的时候的调试也变得更为直观和方便。

5. 对于bPlusTreeNode类刚开始时只是完成了它要求的基础操作，但是在后来与bufferManager模块合并的过程中各种转换边的极为繁琐，因此增加了许许多多的附加函数，增加了代码重用性，代码可阅读性也大幅度提升，在bPlusTree类处理有关bPlusTreeNode的相关事务时也变得便捷很多。