Index Manager 设计报告

一、 模块概述

MiniSQL 的整体设计要求对于表的主属性自动建立 B+树索引，对于声明为 unique 的属性可以通过 SQL 语句由用户指定建立/删除 B+树索引（因此，所有的 B+树索引都是单属性单值的）。

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、 主要功能

创建索引：若语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。
删除索引：若语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。
等值查找：根据所查找的索引和键值返回拥有该键值的记录在表中的位置（块号和块中偏移），若语句执行成功，则返回位置信息，若失败则返回 null 或抛出异常。
插入键值：在索引中插入新键值，若键值已存在，则更新相应位置信息。
删除键值：在索引中删除某个键值，若该键值不存在则什么也不做。

三、 对外提供的接口

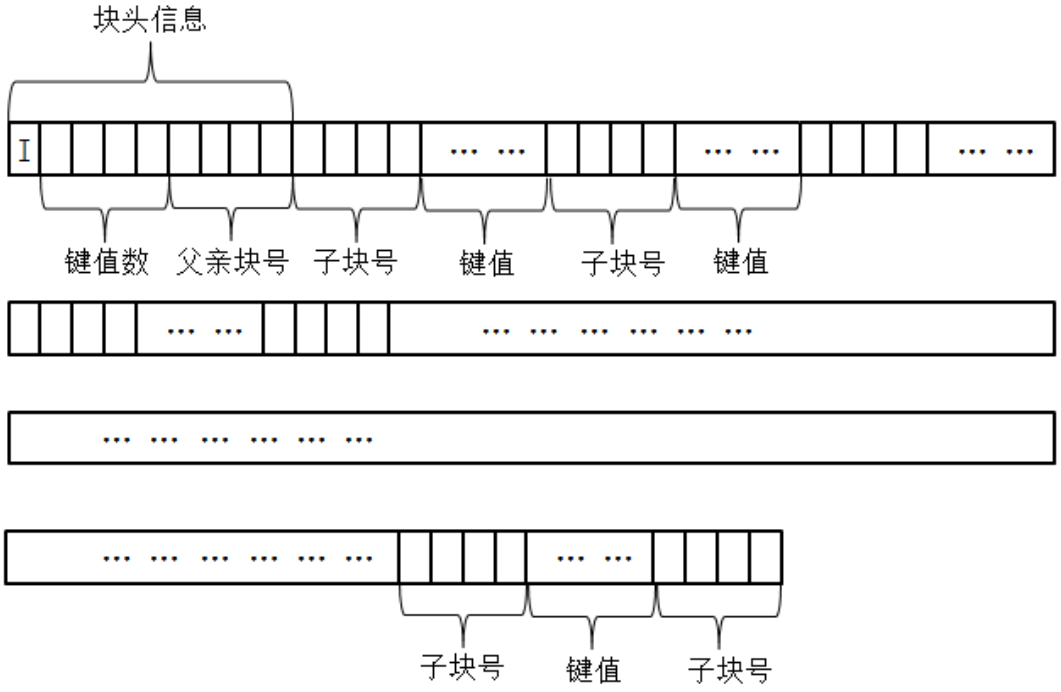
1. 创建索引
   public void createIndex(Table tableInfo,Index indexInfo) //需要 API 提供表和索引信息结构
2. 删除索引，即删除索引文件
   public void dropIndex(Index indexinfor)
3. 等值查找
   public offsetInfo searchEqual(Index indexInfo, String key)
4. 插入新索引值，已有索引则更新位置信息
   public void insertKey(Index indexInfo,String key,int blockOffset,int offset)
5. 删除索引值，没有该索引则什么也不做
   public void deleteKey(Index indexInfo,String deleteKey)

四、 设计思路

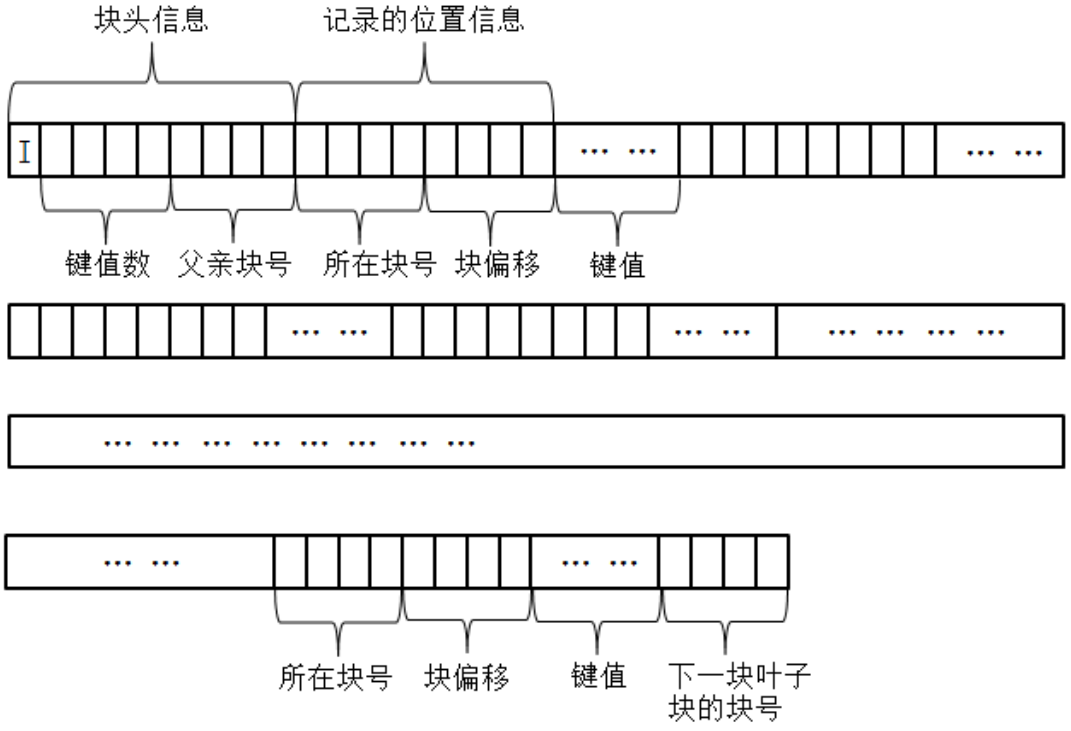
将 4K 块作为 B+树的节点，Index Manager 需要设计一棵不是实体存在的 B+树，因为真实的数据内容是由 Buffer Manager 进行管理的，Index Manager 只对每一个块提供方法的包装。

首先需要设计每一个块的内部结构以记录树结构信息。因为中间块与叶子块的结构不同，所以需要一个位进行标记，还需要知道这个块里有多少个键值以及它的父亲块号。

中间块的结构设计如下：



叶子块的结构设计如下：

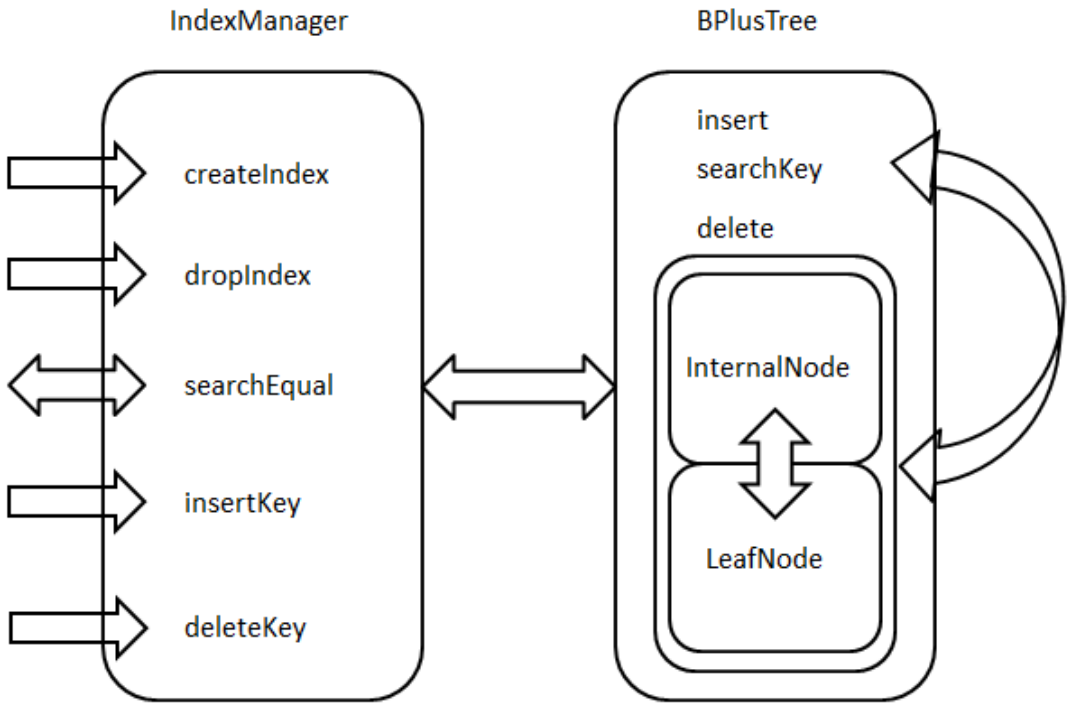


另一方面需要建立一棵只提供方法的 BPLusTree，它由中间节点和叶子节点构成，就是通过它们对中间块和叶子块进行方法的包装。这两类节点极其相似，可以把它们抽象为 Node 抽象类。BPlusTree 除了节点外，还应包含必要的信息比如这棵树的根节点信息。Index Manager 的实现与 BPlusTree 是息息相关的，由 Index Manager 提出操作需求，BPlusTree 进行具体实施。

五、 整体架构

设计类 IndexManager，由其提供对外接口。在每一个函数的实现过程中，临时构建一棵 B+树，该任务由类 BPlusTree 承担。更新成功后，将这棵树的根节点信息存回 Catalog Manager 进行管理。

在 BPlusTree 内，设计一个抽象类 Node，内部成员为 Buffer Manager 提供的 Buffer Block 块，将这些实体存在的缓冲块引用进来，通过一定的函数方法操作，新置、修改块内信息，构建出供 BPlusTree 整体操作的节点。BPlusTree 中主要有构造、插入、等值查找和删除函数，而 Node 类由 InternalNode 和 LeafNode 类继承，承担插入、查找、删除的具体操作。



六、 关键函数和代码（伪代码）

IndexManager 的插入函数（插入新索引值，已有索引则更新位置信息）：

```
public void insertKey(Index indexInfo,String key,int blockOffset,int offset) throws Exception{
    try{
        BPlusTree thisTree=new BPlusTree(indexInfo,buf,indexInfo.rootNum);//创建树访问结构（但不是新树）
        thisTree.insert(key, blockOffset, offset);//插入
        CatalogManager.setIndexRoot(indexInfo.indexName,
        thisTree.myRootBlock.blockOffset);//重置根信息
    }catch(NullPointerException e){
        System.err.println();
    }
}
```

查找和删除的函数与其类似，只不过调用的是 BPlusTree 的对应的查找和删除函数。

IndexManager 的 createIndex 函数相当于此函数的扩展，只不过调用了 BPlusTree 的另一个构造函数（创建索引文件，设置根块号为 0），再将表文件中的每一条记录进行 Insert 插入。

BplusTree 的构造函数其一：

```
BPlusTree(Index indexInfo){

    //新建索引文件
    ……

    //根据索引键大小计算分叉数
    //4k 大小的块要一个字节分辨是叶子节点还是中间节点，四个字节记录索引键数目，3 个字节说明父节点的块号和一个字节（最后）标记是不是根节点
    //还有一个 POINTERLENGTH 长度的指针指向下一个兄弟节点
    //而每个索引键包括 8 个字节的指针（前四个字节表示记录在表文件的那一块，后四个字节表示在该块的偏移量）和 myIndexInfo.columnLength 长度的键值
    ……

    myIndexInfo=indexInfo;
    CatalogManager.setIndexRoot(indexInfo.indexName, 0);
    CatalogManager.addIndexBlockNum(indexInfo.indexName);//更新 Catalog 的信息

    new LeafNode(myRootBlock=BufferManager.createBlock(filename,0)); //创建该索引文件的第一块，并用 LeafNode 类包装它
}
```

BPlusTree 的插入函数为：

```
public void insert(String key,int blockOffset, int offset){
    if (key == null)       throw new NullPointerException();

    //根据块的信息，以不同的类型包装块
    rootNode 设为 InternalNode orLeafNode;

    BufferBlock newBlock=rootNode.insert(key, blockOffset, offset); //节点的插入操作

    if(newBlock!=null){ //假如有返回，说明根块被更新了
        myRootBlock=newBlock;
    }

    CatalogManager.setIndexRoot(myIndexInfo.indexName, myRootBlock.blockOffset);
}
```

BPlusTree 的查找、删除函数与此类似。

在插入的具体过程中，InternalNode 有两个相关函数。

InternalNode 开始执行插入的函数：

```
BufferBlock insert(String insertKey,int blockOffset, int offset){

        //找到了分支位置，获取分支子块的标号
        ……
        BufferBlock nextBlock=BufferManager.readBlock(filename, nextBlockNum); //将这个子块读进来

        //将这个子块进行节点包装
        Node nextNode= new InternalNode(nextBlock) or new LeafNode(nextBlock);

        return nextNode.insert(insertKey, blockOffset, offset); //进入中间节点的递归查找
    }
```

InternalNode 在插入时进行分支的函数：

```
// branchKey 为要新插入的路标，leftChild 为新路标的左子节点（也就是已经存在的节点），rightChild 为新路标的右子节点
    BufferBlock branchInsert(String branchKey,Node leftChild,Node rightChild){
        int keyNum = block.getInt(1, 4);//获取路标数

        if(keyNum==0){ //全新节点
            设置路标和左右子节点的指针
            return this.block; //将该节点块返回作为新根块
        }

        if(++keyNum>MAX_CHILDREN_FOR_INTERNAL){   //分裂节点
            //创建一个新块并包装
            ……
            //找到了路标插入的位置
            ……
            //把需要转移的记录 copy 到新 block
            ……
            //给新插入条留出位置
            ……
            //设置新插入信息
            ……
            //找出原块与新块之间的路标，提供给父节点做分裂用
            ……
            //更新新块的子块的父亲
            for(int j=0;j<=newBlock.getInt(1, 4);j++){
                ……
            }
```

```
                    if(旧块没有父块){   //创建父块
                        //创建新块作为父块
                        ……
                        将新旧块的父块信息置为该块号;
                    }
                    else{
                        新块的父块号=旧块的父块号;
                    }

                    //读出父亲块并为其包装，然后再递归分裂
                    return   ((InternalNode)createNode(ParentBlock)).branchInsert(spiltKey, this,
newNode);
                }

                else{   //不需要分裂节点时
                    //找到插入的位置
                    ……
                    //给新插入条留出位置
                    ……
                    //设置新插入信息
                    ……
                    return null;
                }
            }
```

LeafNode 执行插入的函数与 InternalNode 进行分支的函数类似，当然参数不同。

InternalNode 的查找函数是一个递归过程，具体查找返回值在 LeafNode 中。

LeafNode 的查找函数：

```
    offsetInfo searchKey(String key){
            int keyNum=block.getInt(1, 4);
            if(keyNum==0) return null; //空块则返回 null

            //二分查找
            int start=0;
            int end=keyNum;
            int middle = (start + end) / 2;
            ……

            int pos=8+middle*(myIndexInfo.columnLength+8);
            String middleKey = block.getString(8+pos, myIndexInfo.columnLength);

            //将找到的位置上的表文件偏移信息存入 off 中，返回给上级调用
            offsetInfo off=new offsetInfo();
```

```
            off.offsetInfile=block.getInt(pos, 4);
            off.offsetInBlock=block.getInt(pos+4, 4);

            return middleKey.compareTo(key) == 0 ? off : null;     //再次确认有没有找到这个索引键值，没有则返回 null
        }
```

InternalNode 与删除有关的函数有 6 个，其中主要的函数有两个：开始执行删除的函数和删除一个子块信息（以及他前面的那条路标）的函数。开始执行删除的函数与开始执行插入的函数类似，是个递归调用。

InternalNode 删除一个子块信息的函数：

```
    BufferBlock     delete(BufferBlock blk){
            int keyNum = block.getInt(1, 4);

            for(int i=0;i<=keyNum;i++){
                if(找到了子块标号){
                    //把这条标号和前面的路标都移除
                    ……
                    keyNum--;

                    if(keyNum >=MIN_CHILDREN_FOR_INTERNAL) return null; //移除后直接结束

                    if(没有父节点){
                        if(keyNum==0){     //没有路标，只有一个子块标号时，把它的子块作为根块，把 this 块删除
                            return     BufferManager.readBlock(filename,     block.getInt(8, POINTERLENGTH));
                        }
                        return null;
                    }

                    //找到父亲块
                    ……
                    //查找后续兄弟块
                    for(;j<parentKeyNum;j++){
                        if(读到后续兄弟块){
                            读到该块和后续兄弟块之间的键值 unionKey;

                            //能够合并
                            if((siblingBlock.getInt(1, 4)+keyNum)<=MAX_CHILDREN_FOR_INTERNAL){
                                //调用合并函数将两个块合并
                                return this.union(unionKey,siblingBlock);
```

```
                                }

                                //两个节点原来都是 MIN_FOR_LEAF，delete 一个以后的情况没考虑，这个时候会涉及三个节点
                                //这种情况没有处理，要处理的话将涉及到 3 个兄弟块，因为 this 块有 MIN-1 个路标，兄弟块有 MIN 个路标，向兄弟块挪一个路标显然是白费力气，
                                //但是它们却也不一定能够合并，所以很麻烦
                                //此处没有处理
                                if(siblingBlock.getInt(1,  4)==MIN_CHILDREN_FOR_INTERNAL) return null;

                                //不能合并，通过从兄弟块转移一个路标来满足 B+树路标最小限制，还需注意调整父块中这两个兄弟之间的键值修改
                                (new InternalNode(parentBlock)).exchange(rearrangeAfter(siblingBlock,unionKey),block.blockOffset);
                                return null;
                            }
                        }

                        //找不后续块
                        找到前续兄弟块;

                        //能够合并
                        if((siblingBlock.getInt(1, 4)+keyNum)<=MAX_CHILDREN_FOR_INTERNAL){
                            return (new InternalNode(siblingBlock)).union(unionKey,block);
                        }

                        //上述未想到处理办法的情况
                        if(siblingBlock.getInt(1, 4)==MIN_CHILDREN_FOR_INTERNAL) return null;

                        //重排的情况
                        (new InternalNode(parentBlock)).exchange(rearrangeBefore(siblingBlock,unionKey),sibling);
                        return null;
                    }
                }
                //未找到子块标号
                return null;
            }
```

LeafNode 删除函数与此函数类似，只不过参数是要删除的键值。LeafNode 查找后续兄弟节点时也可以直接使用块末尾的下一块叶子块的块号这个信息，但如果找不到后续兄弟块的话还是需要先找父块，再从父块中获取前一块兄弟叶子块的块号。