Index Manager设计报告

计算机科学与技术学院 计科1403班 陈则衔 3140102231

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，可以完成B+树的创建、删除、维护、等值查找、区间查找、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，均为4K，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，返回用户失败的原因。

删除索引：若语句执行成功，则输出执行成功信息；若失败，返回用户失败的原因。

查找操作：根据所查找的索引、键值、键值所对应的操作(大小比较），返回满足条件的所有记录条（即一张临时表），若查找不到，则返回一个空表，在查找过程中出现错误信息则输出异常信息。

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

|  |
| --- |
| void createIndex(const Table& tableinfor, Index& indexinfor); //需要API提供表和索引信息结构 |

1. 删除索引，即删除索引文件

|  |
| --- |
| void dropIndex(Index& indexinfor); |

1. 查找操作

|  |
| --- |
| Data select(const Table& tableinfor, const Index& indexinfor, vector<int> num, vector<Condition> op, int blockOffset = 0);  //需要API提供表和索引信息结构，同时提供比较操作信息(包括除!=以外的所有操作，也可以是多条）  //num表示建立index索引的属性在op中的编号,blockOfset缺省值为0,因为根节点一定在编号为0的block中(文件) |

1. 插入新索引值，已有索引则更新位置信息

|  |
| --- |
| IndexBranch insertValue(Index& indexinfor, IndexLeaf node, int blockOffset = 0); |

1. 删除索引值，没有该索引则什么也不做

|  |
| --- |
| void deleteValue()  //删除操作时，记录先发生变化(标记位变化)，B+树下次会自动忽略这条记录 |

四、 设计思路

首先，B+树种所有结点大小统一为4K.我们需要知道的是，B+树索引确实是真实存在的，但是被存放在磁盘当中，所有的操作都与Buffer Manager进行交互，因此，我们可以看成整棵B+树都在内存当中，当需要访问某一块时，通过调用Buffer Manager，暂时申请一篇内存存放，当修改完B+树某一个block后，析构函数自动会将这片内存中的信息再存放到BufferManger中的block中。

那么怎么通过某一块结点来访问它的子结点呢？这跟每个结点的结构有关。在每个结点中，我们将一个key值及key值对应的指针看成一条记录，那么在每个结点中会存在N条记录(N由记录长度决定)，显然，叶结点和非叶结点的记录内容是不同的。除此之外，还存在其他一些标志位，如该节点是否为根结点，该节点是否为子结点，N为多少等等。

那么，我们对先记录内容进行区分。

叶结点中的记录：

|  |
| --- |
| class IndexLeaf  {  public:  string key; //索引关键词  int type; //关键词类型  int File\_Offset; //该关键词所对应的一整条记录(表)在文件中的位移，单位block  int Block\_Offset;//该条记录在block中的位移  IndexLeaf();  ...  }; |

非叶结点中的记录

|  |
| --- |
| class IndexBranch  {  public:  string key; //索引关键词  int type; //关键词类型  int ptrChild; //指针，指向子结点，值代表子结点在文件中的block偏移量  IndexBranch();  ...  }; |

这样，我们可以想象，叶结点中包含了N个向量IndexLeaf、非叶节点中包含了N个向量IndexBranch.除了这两者之外，叶结点和非叶结点是有很多公共属性的。我们可以将其抽象出来，构造一个BPlusTree类作为基类，而Leaf类和Branch类作为继承类。

那么，BPlusTree类作为一个基类，其应该包含以下这些公共属性：

|  |
| --- |
| class BPlusTree  {  public:  bool isRoot; //该结点是否为父结点  int bufferNum;  //该结点在文件中的block编号  int ptrFather; //该结点的父结点，若为父节点此值无用  int recordNum; //该结点包含多少条记录，即前文所说的N值  int columnLength; //关键词属性长度  BPlusTree(){}  BPlusTree(int NbufNum): bufferNum(NbufNum), recordNum(0){}  int GetPtr(int pos); //获得某个结点的父节点  int GetRecordNum(); //获得N值  }; |

这样之后，叶结点类和非叶结点类继承之后另加各自加上IndexLeaf和IndexBranch(以向量类vector的方式），除此之外，还有一些函数操作，包括插入，末端弹出以及首端入栈(在B+树分类时，将一个饱和的结点分一半给予空结点时使用).

Leaf：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| R | L | RecordNum | FatherPtr | LeftPtr | RightPtr | Key | FileOff | BlockOff | ... |

Branch:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| R | L | RecordNum | FatherPtr | LeftPtr | RightPtr | Key | ChildPtr | ... |

当数据结构建立完成之后，我们就可以开始做我们主要的工作，IndexManager.此时的IndexManger只需要负责管理，即只提供各个函数而没有实体(即前文所见的实现的各个函数功能)。因为B+树是存在磁盘中，所以没有必要有实体，需要时从BufferManager中拿取即可。

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，从BufferManager中拿取已存在B+树的结点(若有需要)，也可以临时构建一个B+树结点，更新完成之后，BufferManger中的Block已经改变，即完成了既定目标。索引信息传入时是引用，因此可以做到实时管理修改。

在BufferManger的管理下，一颗B+树虽然存放在磁盘中，却可以随时读取，整棵树存放在内存中一样，通过IndexManger类中的函数可以达到对B+的所有操作。这一切基于B+树的数据结构，基类在BPlusTree，以及继承类Leaf和Branch，而Leaf和Branch又是由IndexLeaf和IndexBranch构成的，正式这样的结构支撑起了整棵B+树.具体如图所示：

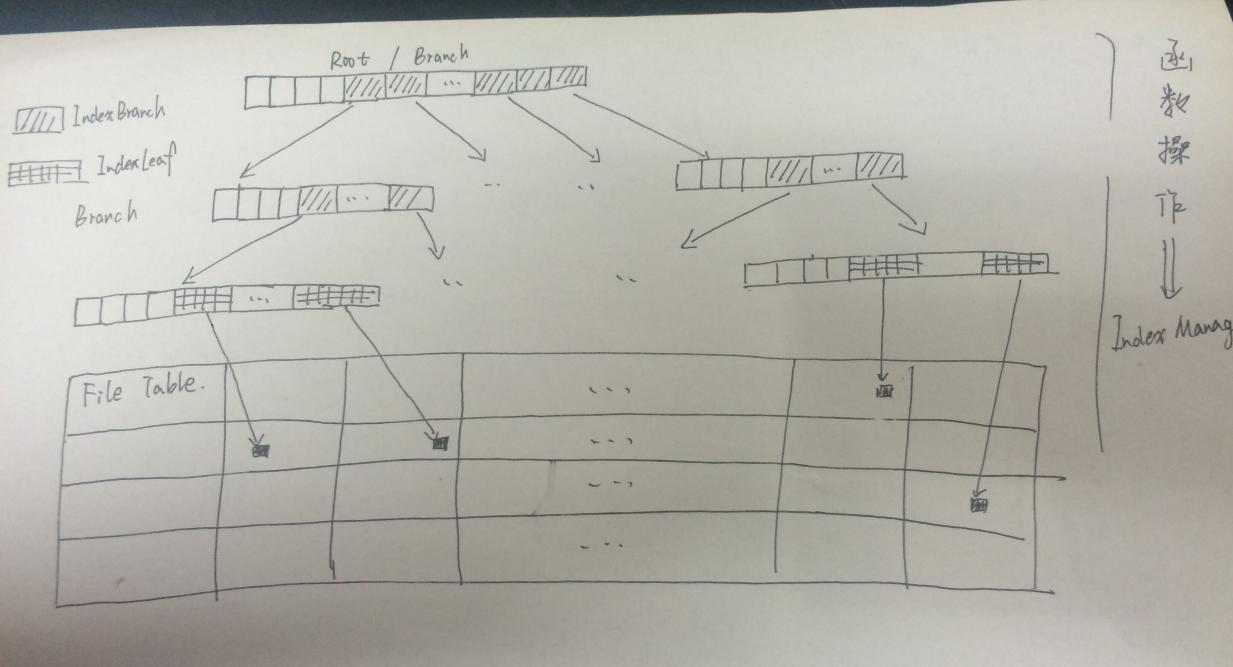


图 5.1 B+树结构

其中需要特别说明的是，我们在IndexManger中操作B+树时，一定会在内存中构造出临时的B+树模块，这些模块的构造就是从Buffer中拿取，我们将值写回Buffer中，是在各个类的析构函数中完成的。

1. 关键函数和代码（伪代码）

IndexManager：CreateIndex(建立B+树）

|  |
| --- |
| void IndexManager::createIndex(const Table & tableinfor, Index & indexinfor)  {    //建立一个新的索引文件  fstream f.new();  //为树根申请一个block并且初始化  int blockNum = buf.getEmptyBlock();  buf.bufferBlock.initial();  //找到表文件，读取表中数据建立B+树  filename = tableinfor.name + ".table";  int length = tableinfor.totalLength + 1;  const int recordNum = BLOCKSIZE / length;  //从表中读取一条条记录，并且把记录插入B+树中  for ( blockOffset = 0 to tableinfor.blockNum )  {  //找到文件中的block在Buffer中的位置  int bufferNum = buf.getBlockNum(filename, blockOffset);  for (offset = 0 to recordNum)  {  //获得记录中的索引关键词  key = getColumnValue();  //形成一个叶记录  IndexLeaf node(key, indexinfor.type, blockOffset, offset);  //插入该叶记录  insertValue(indexinfor, node);  }  }  } |

IndexManager:insertValue(插入一个叶记录)

|  |
| --- |
| //blockOffset表示当前block,缺省值为0(根节点)  IndexBranch IndexManager::insertValue(Index & indexinfor, IndexLeaf node, int blockOffset)  {  //返回值，初始缺省构造为空  IndexBranch reBranch;  //得到当前block  int bufferNum = buf.getBlockNum(filename, blockOffset);  if (isLeaf) //Leaf  {  Leaf leaf(bufferNum, indexinfor);  leaf.insert(node);//插入叶结点中  //分裂  if (leaf.recordNum > MaxrecordNum)  {  if (leaf.isRoot)//根分裂  {  int rbufferNum = leaf.bufferNum;//新根  leaf.bufferNum = buf.addBlockInFile();//旧叶结点  int sbufferNum = buf.addBlockInFile();//新叶结点  //新根/叶结点初始化  Branch/Leaf initial()  //旧叶结点分一半给予新结点 |\*\*\*\*\*\*| + | | --> |\*\*\*| + |\*\*\*\*|  while (leafadd.nodelist.size() < leaf.nodelist.size())  {  IndexLeaf tnode = leaf.pop();  leafadd.insert(tnode);  }  //插入两个标记结点到根结点  IndexBranch tmptNode;  tmptNode = leafadd;  branchRoot.insert(tmptNode);  tmptNode.key = leaf;  branchRoot.insert(tmptNode);  return reBranch;  }  else//叶结点分裂  {  //新叶结点并且初始化  int bufferNum = buf.addBlockInFile(indexinfor);  Leaf leafadd(bufferNum);  //旧结点给予新结点记录  for(Leaf.push.indexLeaf to LeafAdd)  until Leaf.size()<LeafAdd.size();  //父结点增加一条记录  reBranch.insert;  return reBranch;  }//if (leaf.isRoot)  }//if (leaf.recordNum > MaxrecordNum)  else {  return reBranch;  }  }//if (isLeaf)  else //非叶结点  {  Branch branch(bufferNum, indexinfor);  list<IndexBranch>::iterator i = branch.nodelist.begin();  i = MinNodelist(branch,node);//定位到记录  IndexBranch bnode;  bnode = insertValue(indexinfor, node, (\*i).ptrChild);//向下搜索  if (bnode.key == "") //正常插入  {  return reBranch;  }  else //底层分裂，插入新纪录  {  branch.insert(bnode);//插入非叶结点  if (branch.recordNum > MaxrecordNum)//分裂  {  if (branch.isRoot) //根分裂  {  ..  //与叶结点分裂一致  }//if (branch.isRoot)  else //非叶结点分裂  {  ..  //与叶结点分裂一致  }  }//if (branch.recordNum > MaxrecordNum)  else  {  return reBranch;  }  }//if (bnode.key == "")  }////if (isLeaf)  return reBranch;  } |

显然，插入的过程可以分为如下：

Step1.插入时进行判断，如果是叶结点，则直接插入，插入后判断是否需要分裂

Step2.若不是叶结点，则向下寻找，直到叶结点，执行Step1

Step3.当前结点分裂，则其父节点需要加一条记录

在IndexManager：insertValue中，我们使用了两个重要的函数，即IndexLeaf插入Leaf以及IndexBranch插入Branch，以Branch为例，具体如下：

Branch：insert

|  |
| --- |
| void Branch::insert(IndexBranch node)  {  recordNum++;//记录数量+1  list<IndexBranch>::iterator i = nodelist.begin();  if (nodelist.size() == 0)//空  nodelist.insert(i, node);  else  {  //找到合适的位置  for (i = nodelist.begin() to nodelist.end())  if ((\*i).key > node.key)  break;  //插入  nodelist.insert(i, node);  }  } |

析构函数中，我们将要将临时的B+树模块重新写入Buffer中，同样以Branch为例，具体如下：

~Branch

|  |
| --- |
| Branch::~Branch()  {  //是否为根结点  if (isRoot)  buf.bufferBlock[bufferNum].Values[0] = 'R';  else  buf.bufferBlock[bufferNum].Values[0] = '\_';  //一定不是叶结点  buf.bufferBlock[bufferNum].Values[1] = '\_';  //写入recordNum  string strRecordNum = IntToString(recordNum, 4);  strncpy(buf.bufferBlock[bufferNum], strRecordNum, 4);  //写入FaterPtr  int position = 6;  // 0-->R/\_ 1-->L/\_ (2,5)-->recordNum (6,9)-->FP  strncpy(bufferBlock[bufferNum], ptrFather, POINTERLENGTH);  position += POINTERLENGTH;  //该结点不含任何记录  if (nodelist.size() == 0) {  cout << "NodeList.size = 0. Excuse me?" << endl;  exit(0);  }  //将记录写入Buffer  for (i = nodelist.begin() to nodelist.end())  {  string key = (\*i).key;  //定长关键词  while (key.length() <columnLength)  key += NIL;  //写入关键词  strncpy(bufferBlock[bufferNum], key, columnLength);  position += columnLength;  //写入指针  string ptrChild = IntToString((\*i).ptrChild,POINTERLENGTH);  strncpy(bufferBlock[bufferNum], ptrChild, POINTERLENGTH);  position += POINTERLENGTH;  }  } |

1. 附录

相关源文件详见IndexManager.h与IndexManager.cpp