
南開大學
Nankai University

# 南　　开　　大　　学

## 计　算　机　学　院

**数据库实验报告**

---

**lab3**

---

姓名：林帆

学号：2113839

2023 年 5 月 4 日

# 目录

# 一、 实验概述

本次实验需要对数据库中 B+ 树的部分功能进行实现，实现方式是完成 BTreeFile 类（可以理解为一种特殊的 HeapFile 类型）中的节点查找、分裂、分配和合并等基础函数，同时涉及了下列 B ＋树的相关类:

| 类名 | 类的功能 |
|---|---|
| BTreePage | B+ 树中的 page |
| BTreePageId | BTreePage 类的 PageId |
| BTreeEntry | Bnode 节点，含有一个 Key 和指向左右儿子的指针 |
| BTreeLeafPage | 叶节点，B+ 树中存放 tuple 数据的页 |
| BTreeInternalPage | 内部节点，索引其他节点 |

表 1: B+ 树的相关类（部分）

# 二、 exercise1

## (一) findLeafPage()

在 exercise1 中，需要完成函数 findLeafPage()，查找含有给定关键字的 leafPage。需要注意的是，若关键字 key 为 null，返回最左端的节点；若有两个相邻节点都含有 key，则返回两节点中靠左的那个。

由于 B+ 树是二叉树状结构，需要不断向下查找，最终找到满足要求的叶节点，因而函数实现的主要思想是递归，递归返回的条件是查找到了叶节点。而若找到的是内部节点的 pageId 时，在调用 getPage 查找到节点后，利用迭代器从左到右依次遍历这一内部层的 Entry，并比较当前 Entry 关键字与 key，若 Entry>=key，说明 key 位于当前 Entry 的左 child 分支中，递归调用 findLeafPage() 即可。若没有找到 >=key 的 Entry，调用 findLeafPage() 查找右 child 即可。

findLeafPage() 函数的实现

```
1  //先判断page类型
2  //若为叶节点直接返回
3  if(pid.pgcateg()==BTreePageId.LEAF)
4      return (BTreeLeafPage)getPage(tid,dirtypages,pid,perm);
5  //为内部节点，先定位并上锁（只读）
6  BTreeInternalPage page=(BTreeInternalPage) getPage(tid,dirtypages,pid,
       Permissions.READ_ONLY);
7  //创建迭代器
8  Iterator<BTreeEntry> it= page.iterator();
9  BTreeEntry entry=null;
10 //迭代寻找
11 while(it.hasNext()){
12   entry=it.next();
13   //若f为空或者entry>=f时，查找最左节点
14   if(f==null||entry.getKey().compare(Op.GREATER_THAN_OR_EQ,f))
15       return findLeafPage(tid,dirtypages,entry.getLeftChild(),perm,f);
16 }
```

```
17          //若始终没有找到，返回最右节点
18      return findLeafPage(tid,dirtypages,entry.getRightChild(),perm,f);
```

# 三、 exercise2

B+ 树中的节点元素个数不能超过 m 个，当对一个节点的插入操作过多，使得节点元素满后，需要对节点进行分裂操作，分裂过程中有 Entry 的上升步骤。需要注意的是，LeafPage 分裂过程中节点的上升需要新建 Entry，InternalPage 分裂过程中需要实现 Entry 的移动。

## (一) splitLeafPage()

在叶节点的 split 函数中，需要实现分裂、连接和节点上升操作。

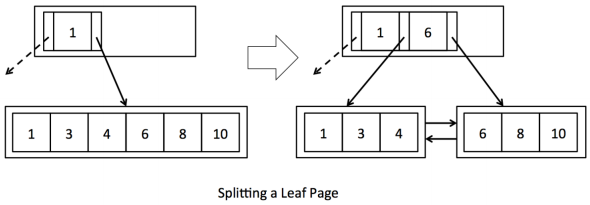


图 1: 叶节点分裂

**step1: 分裂.** 创建新的 leafPage 作为右页面，使用待分裂页面 page 的反向迭代器 reverseIterator，循环遍历 page 中的后一半 tuple，先调用 page 中的 delete，再调用右页面的 insert，实现 tuple 的移动。需要注意**先删除后插入**，否则插入同一个引用声明的 tuple 会导致 recordId 的覆盖。

splitLeafPage() 的分裂操作

```
1       //创建新页面作为分裂后的右页面
2       BTreeLeafPage newRightPage=(BTreeLeafPage)getEmptyPage(tid,dirtypages
            ,BTreePageId.LEAF);
3       //将旧页面的后一半删除，并保存到新建页面中
4       //使用反向迭代器读取后一半的tuple
5       Iterator<Tuple>it= page.reverseIterator();
6       int tupleNum = page.getNumTuples();
7       for(int i=0;i<tupleNum/2;i++){
8               Tuple tuple=it.next();
9               page.deleteTuple(tuple);
10              newRightPage.insertTuple(tuple);
11      }
```

**step2: 连接.** 由于叶节点的相邻节点间有前后指针相连，因而需要对分裂后的连接关系进行更新。利用 getRightSiblingId() 函数获取原页面的右邻居页面，设置右邻居为新建页面，将新建页面的左右邻居分别设为原页面和原页面的右邻居页面。邻居指针的更新使用函数 setLeftSiblingId() 和 setRightSiblingId()。最后修改后，需要将已修改完且后续不再改动的页面放入脏页缓存。

**step3: 节点上升.** 叶节点分裂后，对于原节点和新增节点，需要在上层添加索引（即内部节点），保证关键字满足关系：原叶节点 < 索引节点 <= 新增叶节点。因而需要在父节点中新建并添加以两个叶节点中心作为关键字的 Entry，并添加对应父子关系，更新相应指针。

其中，中心关键字段 mid 的获取利用了新建右页面的迭代器，在 BTreeEntry 的构造函数中传入两个叶节点的 pageId 后即可新建 newEntry，利用父节点（内部节点）的 insertEntry() 函数直接插入 newEntry。完成上述步骤后将父节点也放入脏页缓存，并更新指针。

splitLeafPage() 的节点上升操作

```
// 将分裂中心tuple的字段构成entry直接copy至parent
//确定parent（内部节点）
BTreeInternalPage parent=getParentWithEmptySlots(tid,dirtypages,page.
    getParentId(),field);
//确定中心field(右边页面的第一个tuple)
Field mid=newRightPage.iterator().next().getField(keyField);
//构造entry
BTreeEntry newEntry=new BTreeEntry(mid,page.getId(),newRightPage.getId())
//添加至parent中
parent.insertEntry(newEntry);
//操作后放入脏页缓存
dirtypages.put(parent.getId(), parent);
//更新父子指针
updateParentPointers(tid,dirtypages,parent);
```

而针对父节点增添元素带来的上层迭代分裂问题，在获取父节点时使用的 getParentWithEmptySlots() 函数中已经解决。

最后通过比较上升节点的关键字 mid 和插入的关键字 field 的大小，判断需要返回的 field 当前所在页面（被插入）。

## (二) splitInternalPage()

在内部节点的 split 函数中，主要实现的是分裂和节点上升操作，由于同层的相邻内部节点之间没有指针连接，因而在新建节点后只需要实现上下层索引的改变即可。

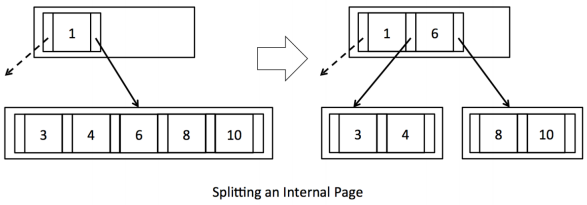


图 2: 内部节点分裂

**step1: 分裂.** 该步骤与叶节点的分裂相同，需要将原页面的前一半数据通过反向迭代器转移至新建右页面，移动语句的顺序依然是先删除后插入。但与 leafPage 的差别在于，leafPage 直接存储 Tuple 类型的数据，因而需要转移的数据类型和对应迭代器的指向类型都是 Tuple; 而 InternalPage 存储页面索引，因而需要转移的数据是 Entry，迭代器类型为 Iterator<BTreeEntry>。

**step2: 节点上升.** 从图 2 可以看到，内部节点分裂后，中心节点需要上升，发生实际的移动，因而无须创建新 Entry，只需再次移动迭代器一步（移动一半 Entry 后迭代器指向中心位置），即

可获得需要上移的节点。然后调用原页面（左侧）的 deleteKeyAndRightChild() 函数删除节点，并将分裂后的两个页面设为 Entry 的左右 child，调用 insert() 函数插入父节点即可。

splitInternalPage() 的节点上升操作

```
        BTreeEntry mid=it.next();
        //左侧删除mid
        page.deleteKeyAndRightChild(mid);
        //设置左右儿子
        mid.setLeftChild(page.getId());
        mid.setRightChild(newRightPage.getId());
        //上升至父节点
        BTreeInternalPage parent=getParentWithEmptySlots(tid,dirtypages,page.
            getParentId(),field);
        parent.insertEntry(mid);
```

随后将三个 InternalPage 放入脏页缓存，并更新指针，返回插入 field 的页面。

# 四、 exercise3

B+ 树中节点的元素个数须不小于 m/2，因而当 delete 元素操作的次数过多后，需要对节点进行调整。当须调整节点的邻居节点元素“富裕”时，可以实现“借”元素并重新分配的操作，即 exercise3 中的前三个 steal 函数；当邻居也不够“富裕”时，需要对两个节点进行合并，即后两个 merge 函数。

## (一) stealFromLeafPage()

对两个相邻叶节点的重新分配，需要实现迭代器设置、tuple 移动和父节点更新。

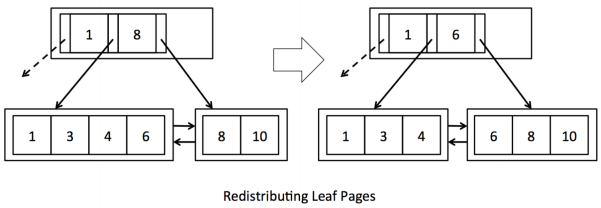


图 3: 叶节点的重新分配

**迭代器设置.** 为保证 tuple 的顺序，需要先判断须调整节点 page 和邻居节点 sibling 的关系：若邻居为右节点，则将 tuple 的迭代器设为邻居的 iterator；若邻居为左节点，则将迭代器设为邻居的反向迭代器 reverseIterator。

stealFromLeafPage() 的迭代器设置操作

```
    Iterator<Tuple>it;
    if(isRightSibling)
        it= sibling.iterator();
    else
        it= sibling.reverseIterator();
```

**tuple 移动.** 以均分为原则, 调用两个叶节点的 getNumTuples() 函数, 计算需要移动的 tuple 数量，设为循环的次数。在循环中，使用迭代器依次读取 tuple，先从邻居中删除，后向自己插入，实现元组的移动。

stealFromLeafPage() 的 tuple 移动操作

```
//计算移动元组数(均分)
int num=(page.getNumTuples()-sibling.getNumTuples())/2;
//开始删除和插入
for(int i=0;i<num;i++){
        //读取迭代器中的tuple
        Tuple tp=it.next();
        sibling.deleteTuple(tp);
        page.insertTuple(tp);
}
```

**父节点更新.** 由于 tuple 的移动，两个叶节点的中心发生了改变，因而需要更新父节点中对应 Entry 的关键字。当前迭代器再移动一步即可找到中心节点 mid，调用 Entry 的 setkey() 函数，修改关键字。需要注意的是，BTreeEntry 类是一个接口，只存储索引的指向关系，实现父节点实际的数据更新，需要调用 InternalPage 的 updateEntry() 函数。

stealFromLeafPage() 的父节点更新操作

```
//确定中心节点
Tuple mid=it.next();
entry.setKey(mid.getField(keyField));
//entry是接口，实际的数据改变需要调用函数
parent.updateEntry(entry);
```

## (二) stealFromLeftInternalPage()

内部节点的重新分配是针对 Entry 的操作，该函数给规定 spare 的节点为左节点。重点需要实现 Entry 的转移和父节点的更新。

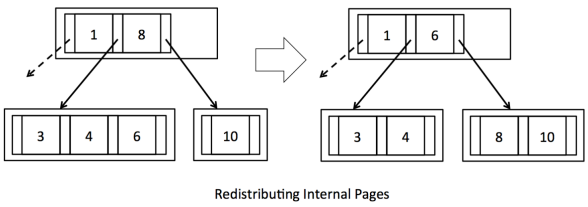


图 4: 内部节点的重新分配

**Entry 转移.** 与叶节点的重新分配相同，首先需要确定邻居节点的迭代器，并计算需要转移的 Entry 的个数。然后以父节点中的 parentEntry 为桥梁，实现 Entry 从左到右的移动。在每次移动的过程中，无须将 Entry 从左侧上移再下移至右侧，只需新建 mid（Entry 类型），设置关键字和左右指针，插入右侧即可；而左侧 Entry 使用迭代器依次拿出，每次循环都将关键字保存 tempKey 中，新建 mid 的关键字代入上一轮保存的 tempKey。

mid 的指针设置需要重点关注，左指针需要指向左页迭代器所指向 Left（Entry 类型）的右指针，右指针指向 mid 插入前右页的第一个 Entry 的左指针。

stealFromLeftInternalPage() 的 Entry 转移

```
BTreeEntry left=null;
BTreeEntry mid=null;
Field tempKey= parentEntry.getKey();
for(int i=0;i<num;i++){
    left=it.next();
    mid=new BTreeEntry(tempKey,left.getRightChild(),page.iterator().next
        ().getLeftChild());
    page.insertEntry(mid);
    //删除子节点
    leftSibling.deleteKeyAndRightChild(left);
    tempKey=left.getKey();
}
```

**父节点更新.** 循环结束后，父节点的 parentEntry 需要更新，而当前 tempKey 即为两个子节点的中心，直接调用 setKey() 修改即可，关键字修改后同样需要使用 updateEntry() 函数完成父节点的实际数据修改。

stealFromLeftInternalPage() 的父节点更新操作

```
//将当前left上移父节点中
parentEntry.setKey(tempKey);//实际没删
parent.updateEntry(parentEntry);
```

最后将脏页缓存并更新节点的指针。

## (三) stealFromRightInternalPage()

该函数规定 spare 的节点为右节点，实现方式与前一个函数几乎相同，过程需要注意新建 Entry 的指针指向，和删除 Entry 时调用的 delete 函数所删 child 的方向。

## (四) mergeLeafPages()

当须调整叶节点的左右节点都没有多余元素时，需要实现两个节点的合并操作，将右节点的全部 tuple 移动至左节点，并更新邻居间的指针连接关系。

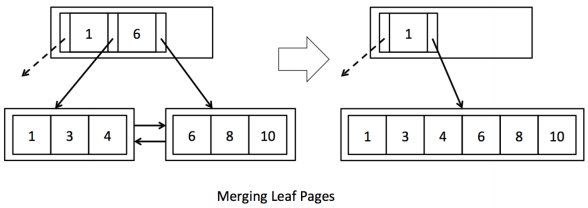


图 5: 叶节点的合并

设置右节点的迭代器依次读取 tuple，先删除后插入 tuple，实现 tuple 从右向左的移动。随后将左节点的右指针指向右节点的右邻居；若右邻居不为 null，则将其左邻居设为左节点。最后，由于右节点将被抛弃，因而父节点中指向左右节点的 parentEntry 也需要删除，而删除前需要调用 setEmptyPage() 函数将删除的右页面清空，防止脏数据存入磁盘的同时，以便后续重新利用该页面。

mergeLeafPages() 函数的实现

```
        //tuple从右->左移动
        Iterator<Tuple> it= rightPage.iterator();
        while(it.hasNext()){
                Tuple tp=it.next();
                rightPage.deleteTuple(tp);
                leftPage.insertTuple(tp);
        }
        //叶节点的连接关系更新
        //确定右节点的右节点
        BTreePageId rightNeighborId=rightPage.getRightSiblingId();
                leftPage.setRightSiblingId(rightNeighborId);
        //右节点的右节点需要设置左邻居（若存在）
        if(rightNeighborId!=null){
                BTreeLeafPage rightNeighbor=(BTreeLeafPage) getPage(tid,
                    dirtypages,rightNeighborId,Permissions.READ_WRITE);
                rightNeighbor.setLeftSiblingId(leftPage.getId());
        }
        //把删除的页面重置，以便后续使用
        setEmptyPage(tid,dirtypages,rightPage.getId().getPageNumber());
        //子节点的更新
        deleteParentEntry(tid,dirtypages,leftPage,parent,parentEntry);
```

### （五） mergeInternalPages()

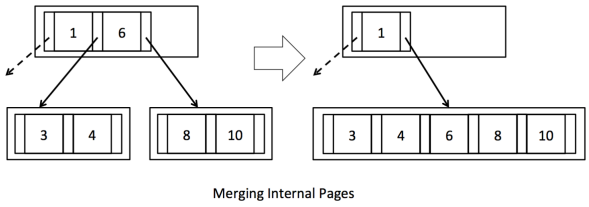


图 6: 内部节点的合并

内部节点的合并需要将左右节点的所有 Entry 合并，并将父节点的对应 Entry 下移合并。下移的实现需要新建 Entry，设置为父节点的关键字，左指针指向左节点最右 Entry 的右指针，右指针指向右节点最左 Entry 的左指针，并插入左节点中。然后将右节点中的 Entry 全部移入左节点中，使用 updateParentPointers() 更新左节点的指针，使用 setEmptyPage() 清除右页面，再将左节点和父节点放入脏页缓存即可。

## 五、 exercise4

exercise4 需要实现对 B+ 树的反向扫描和测试，过程如下所示。

**step1:** 在 BTreeFile 中构建 ReverseFindLeafPage() 函数，只需将原有 findLeafPage() 函数修改为查询右节点，并修改 entry 与 f 的比较方式为 LESSTHANOREQ。

**step2:** 在 BTreeFile 中构建 indexReverseIterator 和 ReverseIterator，及其相应内部类。将调用的 iterator 修改为 reverseIterator，findLeafPage() 调用改为 ReverseFindLeafPage() 函数，并反向改变比较方式。

**Step3:** 新建 BTreeReverseScan 类，将 Reset() 函数中调用的迭代器修改为反向迭代器 ReverseIterator 和 indexReverseIterator。

**step4:** 新建 BTreeReverseScanTest 类，反向修改 BTreeScanTest 类即可。

# 六、 代码提交记录


图 7: gitlab 提交记录

# 七、 实验总结

通过本次实验，学习了 B+ 树的节点查找、分裂、重新分配和合并操作，对 B+ 树的结构及其在数据库中的应用有了更深的理解。