



# simpleDB-Lab3 实验报告

## B+ Tree Index

作者：2113619-曹珉浩

时间：April 26, 2023



# 目录

第 1 章	Exercise-1	1
第 2 章	Exercise-2	2
2.1	BTreeFile.splitLeafPage() . . . . .	2
2.2	BTreeFile.splitInternalPage() . . . . .	3
第 3 章	Exercise-3	4
3.1	BTreeFile.stealFromLeafPage() . . . . .	4
3.2	BTreeFile.stealFromLeftInternalPage() & stealFromRightInternalPage() . . . . .	4
3.3	BTreeFile.mergeLeafPages() . . . . .	5
3.4	BTreeFile.mergeInternalPages() . . . . .	6
第 4 章	BTreeReverseScan & BTreeReverseScanTest	7
第 5 章	Commit & Address	9

## 第 1 章 Exercise-1

本实验中只要求实现 `findLeafPage()` 方法，这个方法递归地搜索内部节点，查找并锁定 B+ 树中可能包含关键字段 `f` 的最左侧页面对应的叶子页面。如果对于传入的页面，已经是一个 leaf page，那么就直接从缓冲池中返回即可；因为树的节点有两种不同类型的页面，如果不是 leaf page，那么就是内部页，我们在内部页面中可以使用 `BTreeEntry.java` 中定义的接口，获取迭代器，遍历内部页面中的键值并访问每个键的左右子页面 ID。

对于传入的字段 `f`，如果 `f` 为空，根据要求需要返回最左边的 leaf page，递归的一路向左即可；若 `f` 不为空，就要进行比较：如果 entry 的 Key 值大于 `f`，那么就递归地向左走，反之就向右走，具体实现代码如下：

---

```
1 private BTreeLeafPage findLeafPage(TransactionId tid, HashMap<PageId, Page> dirtypages,
2   BTreePageId pid, Permissions perm, Field f)
3     throws DbException, TransactionAbortedException {
4     if(pid.pgcateg() == BTreePageId.LEAF) //如果是 leaf page, 从缓冲池中获取页面并将其返回
5         return (BTreeLeafPage) this.getPage(tid, dirtypages, pid, perm);
6     BTreeInternalPage page=(BTreeInternalPage)this.getPage(tid, dirtypages, pid, perm);
7     Iterator<BTreeEntry> entries=page.iterator();
8     if(f==null) //若 f 为空, 找到最左边的叶页
9         return findLeafPage(tid,dirtypages,entries.next().getLeftChild(),perm,f);
10    BTreeEntry entry=entries.next();
11    while(true) {
12        if(entry.getKey().compare(Op.GREATER_THAN_OR_EQ, f))
13            return findLeafPage(tid,dirtypages,entry.getLeftChild(),perm,f);
14        if(entries.hasNext())
15            entry=entries.next();
16        else break;
17    }
18    return findLeafPage(tid,dirtypages,entry.getRightChild(),perm,f);
19 }
```

---

## 第 2 章 Exercise-2

### 2.1 BTreeFile.splitLeafPage()

这个函数实现的功能是拆分叶页以为新元组腾出空间，并根据需要递归拆分父节点以容纳新条目，新条目应该有一个与右侧页面中第一个元组的键字段匹配的键（键被“复制”），以及指向拆分后产生的两个叶页的子指针。根据需要更新兄弟指针和父指针。最后还要返回应插入具有关键字段 field 的新元组的叶页，函数的实现代码如下：

---

```
1  protected BTreeLeafPage splitLeafPage(TransactionId tid, HashMap<PageId, Page> dirtypages,
2  BTreeLeafPage page, Field field)
3  throws DbException, IOException, TransactionAbortedException {
4      BTreeLeafPage newPage=(BTreeLeafPage) this.getEmptyPage(tid, dirtypages, BTreePageId.LEAF);
5      int num=page.getNumTuples()/2;
6      Iterator<Tuple> it=page.iterator();
7      for(int i=0;i<num;i++) {
8          Tuple t=it.next();
9          page.deleteTuple(t);
10         newPage.insertTuple(t);
11     }
12     //如果原来的页有左兄弟，那么要把较小页 newpage 作为它新的右兄弟
13     if(page.getLeftSiblingId()!=null) {
14         BTreeLeafPage sibling=(BTreeLeafPage) this.getPage(tid, dirtypages,page.getLeftSiblingId());
15         sibling.setRightSiblingId(newPage.getId());
16     }
17     newPage.setLeftSiblingId(page.getLeftSiblingId()); //newPage 的左兄弟就是原来页的左兄弟
18     newPage.setRightSiblingId(page.getId()); //小页 newPage 的右兄弟就是剩下的页
19     page.setLeftSiblingId(newPage.getId());
20     Field f=it.next().getField(keyField); //获得中间值，要上升到父节点
21     BTreeEntry entry=new BTreeEntry(f,newPage.getId(),page.getId());
22     BTreeInternalPage parent=this.getParentWithEmptySlots(tid, dirtypages, page.getParentId());
23     parent.insertEntry(entry);
24     this.updateParentPointers(tid, dirtypages, parent);
25     if(f.compare(Op.GREATER_THAN_OR_EQ, field))
26         return newPage;
27     return page;
28 }
```

---

首先，构造了一个新的 leaf page，并接收 num/2 个元组，使得元组在这两个页上均匀分布，同时

可以注意到，newPage 接受的是较小的  $\text{num}/2$  个元组，因此它是靠左边的页。

然后就是 page 的左页，page，page 的右页和 newPage 四个页之间的位置关系：因为 page 保留较大的  $\text{num}/2$  个元组，它依然与原来的右页连接，因此在这个函数中不需要考虑其右页的问题。对于剩下的三个页，它们的排序过程就类似于在双向链表中插入一个新节点，首先为了不失去左页的连接，应该先把新页的左页设置为原来页的左页，以及原来页左页的右页应设置为 newPage(这两步顺序是可以颠倒的)，然后将 newPage 的右页设为 page，page 的左页设为 newPage，按照这个策略更新，就不会失去任何页的访问能力，并成功的插入了一个新的 leaf page。

构造完两个新的 leaf page 之后，还要更新它们的父节点，值得注意的是，两个 leaf page 的父节点不再是 leaf page 而是内部页。因为刚才的迭代器已经删除了  $\text{num}/2$  个元组，再调用这个迭代器的 next() 函数就获得了中间值，要把这个中间值复制到父节点，并做更新操作。

最后，要求返回插入关键字段 field 应该在的页面，使用 compare() 方法即可，如果中间值比 field 大的话，就返回左页 newPage，否则就返回右页 page。

## 2.2 BTreeFile.splitInternalPage()

分裂内部页的函数实现和分裂 leaf page 的函数实现大致相同，值得注意的一点是，leaf page 中中间值是不删除的，因为对于每个 key，它都会在内部页和 leaf page 中各出现一次；而在内部页分裂的过程中，中间值是要删除的：

---

```
1 page.deleteKeyAndLeftChild(entry);
```

---

其余代码基本类似，不在此赘述。

## 第3章 Exercise-3

### 3.1 BTreeFile.stealFromLeafPage()

这个函数从兄弟 leaf page 那里窃取元组并将它们复制到给定页面，以便两个页面至少是半满的，并且要更新父条目，使键与右侧页面中第一个元组的键字段相匹配，代码实现如下：

---

```
1  protected void stealFromLeafPage(BTreeLeafPage page, BTreeLeafPage sibling,
2      BTreeInternalPage parent, BTreeEntry entry, boolean isRightSibling) throws DbException {
3      int num=(sibling.getNumTuples()-page.getNumTuples())/2; //要偷的元组数，实现均匀分布
4      Iterator<Tuple> it=(isRightSibling)?sibling.iterator():sibling.reverseIterator();
5      Tuple t=null;
6      for(int i=0;i<num;i++) {
7          t=it.next();
8          sibling.deleteTuple(t);
9          page.insertTuple(t);
10     }
11     if(isRightSibling)
12         t=it.next();
13     entry.setKey(t.getField(keyField));
14     parent.updateEntry(entry);
15 }
```

---

实现思路为：先计算要偷取的元组数，然后获得迭代器，如果是右兄弟的话就偷前几个小的，为正向迭代器；否则就偷左兄弟几个大的，即反向迭代器。然后进行删除和插入操作。最后更新父条目，如果是窃取左兄弟的元组，因为左兄弟的元组都比该页小，最右边的元组是最大的，移动之后父节点的右指针依然满足大于等于的关系，迭代器不需要进行移动，但如果窃取的是右兄弟的元组，迭代器就要下移一位，以满足 B+ 树的要求。

### 3.2 BTreeFile.stealFromLeftInternalPage() & stealFromRightInternalPage()

这两个函数的实现基本相同，在此只阐述 stealFromLeftInternalPage() 方法的实现。

这个方法从左兄弟窃取条目并将它们复制到给定页面，以便两个页面至少是半满的，键可以被认为是在父条目中旋转，因此父条目中的原始键被“下拉”到右侧页面，而左侧页面中的最后一个键被“推上”到父条目。根据需要更新父指针。实现代码如下：

---

```
1  protected void stealFromLeftInternalPage(TransactionId tid, HashMap<PageId, Page> dirtypages,
2      BTreeInternalPage page, BTreeInternalPage leftSibling, BTreeInternalPage parent,
```

---

```

3 BTreeEntry parentEntry) throws DbException, IOException, TransactionAbortedException {
4     int num=(leftSibling.getNumEntries()-page.getNumEntries())/2;
5     Iterator<BTreeEntry> entries=leftSibling.reverseIterator();
6     //center 要从父节点旋转下来到右节点
7     BTreeEntry center=new BTreeEntry(parentEntry.getKey(),null,
8         page.iterator().next().getLeftChild());
9     for(int i=0;i<num;i++) {
10         BTreeEntry left=entries.next();
11         center.setLeftChild(left.getRightChild());
12         page.insertEntry(center);
13         center=new BTreeEntry(left.getKey(),null,left.getRightChild());
14         leftSibling.deleteKeyAndRightChild(left);
15     }
16     parentEntry.setKey(center.getKey());
17     parent.updateEntry(parentEntry);
18     this.updateParentPointers(tid, dirtypages, page);
19 }

```

这个函数就是一个：父页面最左边的 entry 降落到右页面，然后左页面最右边的 entry 升到父节点这样的旋转过程，首先依旧计算出要移动的元组数量，然后获得一个反向迭代器，用于获得左页面中较大的 num 个 entry。

之后新定义一个 entry(要降落下来的父节点)，因为这个 entry 比删除的 entry 的 key 要小，因此父节点降落下来后，原来 entry 的左孩子要成为父节点的右孩子，而新降落下来的父节点的左孩子就为空。然后不断循环这个旋转过程。

### 3.3 BTreeFile.mergeLeafPages()

这个方法通过将所有元组从右页移动到左页来合并两个叶页。代码实现如下：

```

1 protected void mergeLeafPages(TransactionId tid, HashMap<PageId, Page> dirtypages,
2     BTreeLeafPage leftPage, BTreeLeafPage rightPage,
3     BTreeInternalPage parent, BTreeEntry parentEntry)
4     throws DbException, IOException, TransactionAbortedException {
5     Iterator<Tuple> it=rightPage.iterator();
6     while(it.hasNext()) {
7         Tuple t=it.next();
8         rightPage.deleteTuple(t);
9         leftPage.insertTuple(t);
10    }
11    if(rightPage.getRightSiblingId()!=null) {

```



```

12     BTreeLeafPage sibling=(BTreeLeafPage) this.getPage(tid, dirtypages,
13         rightPage.getRightSiblingId(), Permissions.READ_ONLY);
14     sibling.setLeftSiblingId(leftPage.getId());
15 }
16 leftPage.setRightSiblingId(rightPage.getRightSiblingId());
17 this.setEmptyPage(tid, dirtypages, rightPage.getId().getPageNumber());
18 this.deleteParentEntry(tid, dirtypages, leftPage, parent, parentEntry);
19 }

```

首先获得右侧页面元组的迭代器，然后在循环中将右侧页面的元组全都插入到左侧页面中，循环结束时右侧页面为一个空页，但要注意如果这个右侧页面的还有右兄弟，就要把这个页面的左兄弟和它的右兄弟相连，完成这个操作后，把右页设为空页，最后删除父级 entry。

### 3.4 BTreeFile.mergeInternalPages()

这个方法通过将所有条目从右页移动到左页并从父条目“下拉”相应的键来合并两个内部页面，并根据需要更新 parent 指针，使右页可重用。具体实现和合并右 leaf page 很相似，代码实现如下：

```

1  protected void mergeInternalPages(TransactionId tid, HashMap<PageId, Page> dirtypages,
2      BTreeInternalPage leftPage, BTreeInternalPage rightPage,
3      BTreeInternalPage parent, BTreeEntry parentEntry)
4      throws DbException, IOException, TransactionAbortedException {
5      //要先把父 entries 下拉到左页面：
6      leftPage.insertEntry(new BTreeEntry(parentEntry.getKey(),
7          leftPage.reverseIterator().next().getRightChild(),
8          rightPage.iterator().next().getLeftChild()));
9      Iterator<BTreeEntry> entries=rightPage.iterator();
10     while(entries.hasNext()) {
11         BTreeEntry entry=entries.next();
12         rightPage.deleteKeyAndLeftChild(entry);
13         leftPage.insertEntry(entry);
14     }
15     this.setEmptyPage(tid, dirtypages, rightPage.getId().getPageNumber());
16     this.updateParentPointers(tid, dirtypages, leftPage);
17     this.deleteParentEntry(tid, dirtypages, leftPage, parent, parentEntry);
18 }

```

首先要注意，在合并之前要把父级 entry 插入进左页面，以保证顺序，然后依然是获得迭代器，从右侧移动到左侧。这些操作都和合并 leaf page 很类似，最后再更新一下左页的 parent 指针即可。



## 第 4 章 BTreeReverseScan & BTreeReverseScanTest

实现了一个名为 BTreeReverseScan 的类，它反向扫描 BTreeFile，在参考了 BTreeScan 类之后，这个反向类的实现思路如下：

- 新增反向迭代函数 ReversefindLeafPage()，与正向函数 findLeafPage() 正好相反：当传入字段 f 为空时，返回最右面的 leaf page；若字段 f 不为空，在下面的递归实现中，先向右递归再向左递归。

---

```
1 private BTreeLeafPage ReversefindLeafPage(TransactionId tid,
2     HashMap<PageId, Page> dirtypages, BTreePageId pid, Permissions perm, Field f)
3     throws DbException, TransactionAbortedException {
4     if (pid.pgcateg() == BTreePageId.LEAF)
5         return (BTreeLeafPage) getPage(tid, dirtypages, pid, perm);
6     BTreeInternalPage page = (BTreeInternalPage) getPage(tid, dirtypages,
7         pid, Permissions.READ_ONLY);
8     Iterator<BTreeEntry> entries = page.reverseIterator(); //获取该页面的反向迭代器
9     if (f == null) //若 f 为空，找到最右面的 leaf page
10         return findLeafPage(tid, dirtypages, entries.next().getRightChild(), perm, f);
11     BTreeEntry entry = null;
12     while(entries.hasNext()){
13         entry = entries.next();
14         if(entry.getKey().compare(Op.LESS_THAN_OR_EQ, f))
15             return findLeafPage(tid, dirtypages, entry.getRightChild(), perm, f);
16     }
17     return findLeafPage(tid, dirtypages, entry.getLeftChild(), perm, f);
18 }
```

---

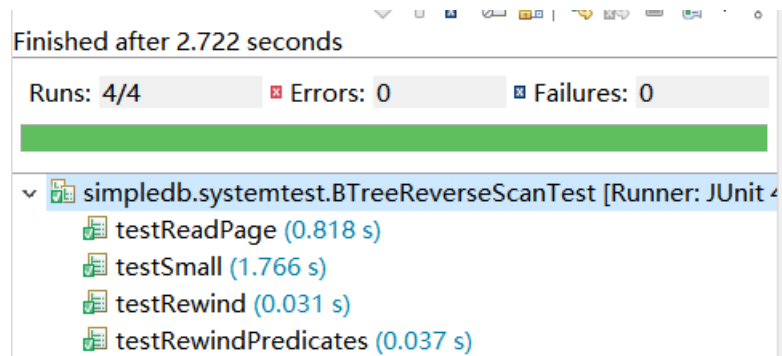
- 参考辅助类 BTreeSearchIterator 创建辅助类 BTreeReverseSearchIterator，注意把正向迭代器修改为反向迭代器，然后把大于改为小于，并使用刚才实现的 ReversefindLeafPage() 方法代替原来的 findLeafPage() 方法即可，代码不在此赘述。
- 新建 BTreeReverseScan 类，主要修改 reset() 方法：使用上两步中实现的辅助类与方法，将正向迭代器更改为反向迭代器，其他地方不需修改：

---

```
1 if(ipred == null) //将这里改为反向迭代器
2     this.it = ((BTreeFile) Database.getCatalog().
3         getDatabaseFile(tableid)).reverseIterator(tid);
4 else //将这里改为反向迭代器
5     this.it = ((BTreeFile) Database.getCatalog().
6         getDatabaseFile(tableid)).indexReverseIterator(tid, ipred);
```

---

- 最后参考 BTreeScanTest 实现反向测试类 BTreeReverseScanTest，把原来正向的相关类和方法均修改为反向即可，代码不在此赘述。然后进行测试，得到的实验结果如下：






















## 第 5 章 Commit & Address

Gitlab 仓库地址：<https://gitlab.com/cmh1447283266/simpliedb-2113619>

新增类地址：[BTreeReverseScan.java](#) & [BTreeReverseScanTest.java](#)

commit 记录：

26 Apr, 2023 2 commits		
 新增的两个类文件 cmh authored 2 minutes ago	fc41e671	 
 完成了Lab3 新增了两个类文件 cmh authored 5 minutes ago	92a1c0cd	 
25 Apr, 2023 1 commit		
 Lab-3 commit5(finish exercise3) cmh authored 1 day ago	e77beb7a	 
24 Apr, 2023 1 commit		
 Lab-3 commit4 cmh authored 2 days ago	ebf8d406	 
23 Apr, 2023 1 commit		
 Lab-3 commit3(finish exercise-2) cmh authored 3 days ago	32f69f1c	 
22 Apr, 2023 1 commit		
 Lab-3 commit2 cmh authored 3 days ago	3e81ceea	 
21 Apr, 2023 1 commit		
 Lab-3 commit1 cmh authored 5 days ago	24fe162f	