南開大學
Nankai University

计算机学院

数据库 **SimpleDB** 实验报告

Lab3

姓名：唐明昊

学号：2113927

专业：计算机科学与技术

2023 年 4 月 30 日

目录

1 Search **2**

2 Insert **3**

2.1 splitLeafPage 3

2.2 splitInternalPage 4

3 Delete **6**

3.1 handleMinOccupancyPage 6

3.2 handleMinOccupancyLeafPage 6

3.2.1 stealFromLeafPage 6

3.2.2 mergeLeafPages 7

3.3 handleMinOccupancyInternalPage 7

3.3.1 stealFromInternalPage 7

3.3.2 mergeInternalPages 8

4 Extra Credit **8**

4.1 BTreeReverseScan 8

4.2 reverseIterator & reverseIndexIterator 9

4.3 BTreeReverseScanTest 9

5 Git Commit History **10**

1 Search

findLeafPage 函数对给定的 key 值，找到最合适的 left-most 的 page，特别的，key 值为 null 时，就返回最左边的叶节点。函数在内节点中递归的搜索，直到找到叶节点，在该过程中，会通过 Permission.READ_ONLY 锁定路径上的所有内节点。

findLeafPage

```
1  private BTreeLeafPage findLeafPage(TransactionId tid, HashMap<PageId, Page>
       dirtypages, BTreePageId pid, Permissions perm, Field f)
2      throws DbException, TransactionAbortedException {
3      // some code goes here
4      // base case:首先判断当前节点类型, 如果是叶节点则返回
5      int type=pid.pgcateg();
6      if(type==BTreePageId.LEAF)
7        // fetch the page from the buffer pool and return it
8        // calling the wrapper function
9        return (BTreeLeafPage) getPage(tid, dirtypages, pid, perm);
10
11     // locks all internal nodes along the path to the leaf node
12     BTreeInternalPage internalPage =(BTreeInternalPage) getPage(tid, dirtypages, pid,
           Permissions.READ_ONLY);
13
14     // iterate through the entries in the internal page and compare the entry value
           to the provided key value
15     Iterator<BTreeEntry> it=internalPage.iterator();
16     BTreeEntry temp = null;// 遍历出一个entry
17     while(it.hasNext()) {// 遍历当前节点所有的entry寻找f
18       temp=it.next();
19       // recurse on the left-most child every time in order to find the left-most
             leaf page
20       if(f==null)
21         return findLeafPage(tid,dirtypages,temp.getLeftChild(),perm,f);
22       if(temp.getKey().compare(Op.GREATER_THAN_OR_EQ,f))
23         // return the first (left) leaf page-> f<=currentKey
24         return findLeafPage(tid,dirtypages,temp.getLeftChild(),perm,f);
25     }
26     // 找到了最后一个entry了, 那只能是往右边走
27     return findLeafPage(tid,dirtypages,temp.getRightChild(),perm,f);
28   }
```

函数首先判断基本条件，如果当前节点为叶节点，则调用 getPage 获取该页并返回。如果不是叶节点，则遍历该 internalPage 的 entry，寻找当前 key 所在的子节点页：当 f 不为 null 时，找到第一个 entry>=key，向下递归遍历该 entry 的左孩子。当 while 循环执行完都没找到合适的子节点，则只能去最右孩子节点寻找了。

2 Insert

2.1 splitLeafPage

向满的叶节点插入 tuple 时需要将节点分裂，并递归地向上传递 entry，分裂内节点，为新的 tuple/entry 开出空间。叶节点向内节点传输时做的是复制操作，将右半节点的第一个 tuple 的 field 值作为 entry 的 key 值，并更新叶节点的兄弟指针和父指针。

splitLeafPage

```
protected BTreeLeafPage splitLeafPage(TransactionId tid, HashMap<PageId, Page>
    dirtypages, BTreeLeafPage page, Field field)
    throws DbException, IOException, TransactionAbortedException {
    // some code goes here

        // 1.Split the leaf page by adding a new page on the right of the existing
    // page and moving half of the tuples to the new page.

    // call getEmptyPage() to get the new page
    BTreeLeafPage rightPage=(BTreeLeafPage)getEmptyPage(tid, dirtypages,
        BTreePageId.LEAF);
    // moving a subset of tuples/entries from a page to its right sibling
    int numberOfTuples=page.getNumTuples();
    Iterator<Tuple> it=page.reverseIterator();
    for(int i=0;i<numberOfTuples/2;i++) {
      Tuple temp=it.next();
      page.deleteTuple(temp);// 从原页面删除该tuple
      rightPage.insertTuple(temp);// 放到右兄弟去
    }

    // 2.Copy the middle key up into the parent page,
    // and recursively split the parent as needed to accommodate the new entry.
    // getParentWithEmtpySlots() will be useful here.

    // interact with leaf and internal pages using .iterator() to iterate through the
        tuples/entries in each page!
    Field midKey=rightPage.iterator().next().getField(keyField);
    // 将key包装成一个向上传递的entry
    BTreeEntry pushedUp=new BTreeEntry(midKey,page.getId(),rightPage.getId());
    BTreeInternalPage
        parent=getParentWithEmptySlots(tid,dirtypages,page.getParentId(),field);
    // 插入该entry
    parent.insertEntry(pushedUp);
    dirtypages.put(parent.getId(), parent);
    // update the parent pointers of all the children that were
        moved->updateParentPointers()
    updateParentPointers(tid,dirtypages,parent);


    // 3.Don't forget to update the sibling pointers of all the affected leaf pages.
```

```
36
37      // 原节点右节点指针更新
38      BTreePageId oldRightPageId=page.getRightSiblingId();
39      if(oldRightPageId!=null) {// 可能就已经是最右边了
40        // 更新左右指针
41        BTreeLeafPage oldRightPage=(BTreeLeafPage)getPage(tid,dirtypages,oldRightPageId,
    Permissions.READ_ONLY);
42        oldRightPage.setLeftSiblingId(rightPage.getId());
43        rightPage.setRightSiblingId(oldRightPageId);
44        // 放入缓存
45        dirtypages.put(oldRightPageId, oldRightPage);
46      }
47      // 分裂节点左右指针更新
48      page.setRightSiblingId(rightPage.getId());
49      rightPage.setLeftSiblingId(page.getId());
50      // 放入缓存
51      dirtypages.put(rightPage.getId(), rightPage);
52      dirtypages.put(page.getId(), page);
53
54      // 4.Return the page into which a tuple with the given key field should be
          inserted.
55        // 看field是插入到哪一半
56      if(field.compare(Op.LESS_THAN, midKey))// 比midKey小，则插入到原来那一页中
57        return page;
58      else
59        return rightPage;
60
61    }
```

函数首先调用 getEmptyPage 获得一个空白的叶节点，再将原节点中一半的 tuple 移动到右兄弟中去。经实验手册提醒，移动页中的内容使用迭代器，由于是将原页中右半部分移动，所以使用 reverseIterator 从后往前遍历。需要注意的是，在循环体中应该**先从原页中删除，再插入到新页中**。因为在调用 insertTuple 时会为其设置 RecordId，而删除时会通过 RecordId 定位，先插入再删除则会产生异常。

移动完 tuple 后，需要将中间值包装为 entry 复制到父节点，中间值即为 rightPage 第一个元素。在向父节点插入元素前，需要调用 **getParentWithEmptySlots**，该函数会调用 splitInternalPage 等函数为 entry 开辟空间。插入过后调用 **updateParentPointers** 调整指针，将子节点的父指针指向父亲。

接着调整兄弟指针：将新页放入链表中，调整左右兄弟指针。

最后返回 tuple 插入的那一页：将 field 与中间值做判断，即可得到 tuple 放在左右哪一页中。

2.2 splitInternalPage

内节点分裂与叶节点分裂不同的点是，**中间值不是复制到父节点，内节点是直接将中间值 push 到父节点**。

splitInternalPage

```
protected BTreeInternalPage splitInternalPage(TransactionId tid, HashMap<PageId,
    Page> dirtypages, BTreeInternalPage page, Field field)
        throws DbException, IOException, TransactionAbortedException {
    // some code goes here

        // 1.Split the internal page by adding a new page on the right of the
            existing page

    BTreeInternalPage rightPage = (BTreeInternalPage) getEmptyPage(tid, dirtypages,
        BTreePageId.INTERNAL);
    // moving half of the entries to the new page.
    int numberOfEntries=page.getNumEntries();
    Iterator<BTreeEntry> it=page.reverseIterator();
    for(int i=0;i<numberOfEntries/2;i++) {
      BTreeEntry temp=it.next();
      // deletes only a key and a single child pointer
      // 报错->tried to delete entry on invalid page or table
      page.deleteKeyAndRightChild(temp);// 从后往前删
      rightPage.insertEntry(temp);// 得先删除再插入! 插入时会设置他的recordId!

    }

    // 2.Push the middle key up into the parent page,
    // and recursively split the parent as needed to accommodate the new entry.
    // getParentWithEmtpySlots() will be useful here.

    BTreeEntry midEntry=it.next();// 再拿出一个entry
    page.deleteKeyAndRightChild(midEntry);
    // push up, 设置mid的左右孩子
    midEntry.setLeftChild(page.getId());
    midEntry.setRightChild(rightPage.getId());
    // 插入父节点
    BTreeInternalPage
        parent=getParentWithEmptySlots(tid,dirtypages,page.getParentId(),
  midEntry.getKey());
    parent.insertEntry(midEntry);
    // 标记为脏
    dirtypages.put(parent.getId(), parent);
    dirtypages.put(page.getId(), page);
    dirtypages.put(rightPage.getId(), rightPage);

    // 3.Don't forget to update the parent pointers of all the children moving to the
        new page.
    // updateParentPointers() will be useful here.

    // 不需要像叶子节点一样调整兄弟间的指针
    updateParentPointers(tid,dirtypages,parent);
    updateParentPointers(tid,dirtypages,page);// 作为内部节点, 同样有child
```

```
44      updateParentPointers(tid,dirtypages,rightPage);
45
46      // 4.Return the page into which an entry with the given key field should be
            inserted.
47
48      if(field.compare(Op.LESS_THAN, midEntry.getKey()))
49        return page;
50      else
51        return rightPage;
52
53    }
```

代码整体逻辑与叶节点分裂是一致的：生成新的一页，移动一半的 entry 过去；将中间 entry 移动到父节点，但由于这里是 push，所以当前页需要调用 deleteKeyAndRightChild 移除当前 entry；调用 updateParentPointers 调整各节点父指针；最后返回对应页。

3 Delete

3.1 handleMinOccupancyPage

在调用删除函数时,会判断删除后当前页是否会少于半满,进而进行处理。**handleMinOccupancyPage** 即是最上层的处理函数，它再根据页的类型调用 leaf 或者 internal 的处理函数。

3.2 handleMinOccupancyLeafPage

handleMinOccupancyLeafPage **优先从当前页的左兄弟借 key**，若其有多余 key 则调用 stealFromLeafPage 从中拿取 tuple 到当前页；若兄弟也在最少值，则将兄弟与当前页合并到一起。

3.2.1 stealFromLeafPage

函数根据所借兄弟的左右情况获取相应的迭代器；而后读取一定量的 tuple 到当前页中，保持两边量均等；最后最重要的，需要**调整父节点对应 entry 的 key 值**，与前面保持一致，选择右兄弟的第一个值做为 key 值，调用 updateEntry。

stealFromLeafPage

```
protected void stealFromLeafPage(BTreeLeafPage page, BTreeLeafPage sibling,
    BTreeInternalPage parent, BTreeEntry entry, boolean isRightSibling) throws
        DbException {
    // some code goes here
        // Move some of the tuples from the sibling to the page so
    // that the tuples are evenly distributed.
    Iterator<Tuple> it;
        // 首先判断是借左兄弟还是右兄弟
    if(isRightSibling)
      it=sibling.iterator();// 拿右兄弟第一个key
    else
      it=sibling.reverseIterator();// 拿左兄弟最后一个key
```

```
    // 两边均分，需要拿出numberOfSteal个tuple
    int numberOfSteal=(sibling.getNumTuples()+page.getNumTuples())/2
        -page.getNumTuples();
    Tuple tupleToSteal=null;
    for(int i=0;i<numberOfSteal;i++) {
      tupleToSteal=it.next();
      // 先删除再插入
      sibling.deleteTuple(tupleToSteal);
      page.insertTuple(tupleToSteal);
    }
    // Be sure to update the corresponding parent entry.
    // split的时候都是把右节点第一个key放上去
    if(isRightSibling)tupleToSteal=it.next();//
        如果是右兄弟借需要再走一个拿右兄弟的第一个key
    entry.setKey(tupleToSteal.getField(keyField));
    parent.updateEntry(entry);
  }
```

3.2.2 mergeLeafPages

将处于最低限度的两页合并到一起，操作逻辑就是将右边页中所有的 tuple 移动到左边页来。除此之外几个需要注意的点是：

- 调整兄弟指针：右边页中所有的元素删除后，该页为空，则需要将右兄弟的右兄弟连到左兄弟去。
- 将右兄弟设空，使得可以再次使用：调用 setEmptyPage 函数将右兄弟置为空页。
- 将父节点对应左右兄弟间的 entry 删除：由于两页合并为一页，则两者之间的 entry 需要调用 deleteEntry 函数进行删除。

3.3 handleMinOccupancyInternalPage

内节点的处理与叶节点的处理思路上大体一致，但在向兄弟节点借元素时，**需要将父节点的 entry pull down**，为了方便处理，分别为向左借向右借做函数。

3.3.1 stealFromInternalPage

从兄弟节点迭代获取元素时，不能直接将其移动到相应页中，而是**需要以父节点为桥**，一个个的移动过去。具体来说，每次从左兄弟获取到一个 entry 时，将其右孩子指针赋给父节点的 midEntry，再将 midEntry pull down，插入到 page 中，而后 midEntry 的值再设为从左兄弟获取的 entry。该过程就实现了从左兄弟移动元素到右兄弟，同时保证父亲节点的 entry 不产生问题。

stealFromLeftInternalPage

```
protected void stealFromLeftInternalPage(TransactionId tid, HashMap<PageId, Page>
    dirtypages,
      BTreeInternalPage page, BTreeInternalPage leftSibling, BTreeInternalPage parent,
      BTreeEntry parentEntry)
  throws DbException, IOException, TransactionAbortedException {
```

```
        // Move some of the entries from the left sibling to the page so
    // that the entries are evenly distributed.

    Iterator<BTreeEntry>it=leftSibling.reverseIterator();// 从左兄弟借
    int numberOfSteal=(leftSibling.getNumEntries()-page.getNumEntries())/2;
    // the original key in the parent is "pulled down" to the right-hand page
    BTreeEntry entryInSibling=null;
    // 以parent的key为桥，一个个放到右边的page中
        //（准备放到parent，但只有最后一个才会真的放上去）
    BTreeEntry midEntry=new
        BTreeEntry(parentEntry.getKey(),null,page.iterator().next().getLeftChild());

    for(int i=0;i<numberOfSteal;i++) {
      entryInSibling=it.next();// 左兄弟拿出一个entry
      midEntry.setLeftChild(entryInSibling.getRightChild());// 设置对应的子指针
      page.insertEntry(midEntry);// 从parent上面拿下来
      // 准备拿到parent上面去再放入右边兄弟
      // 如果真的放过去了，会在插入前会再拿出左孩子指针给它。
            // 右孩子已经被上一个当作左孩子放过去了!
      midEntry=new
          BTreeEntry(entryInSibling.getKey(),null,entryInSibling.getRightChild());
      leftSibling.deleteKeyAndRightChild(entryInSibling);//
          拿entry，相应的孩子也得拿过去
    }

    // Be sure to update the corresponding parent entry.
    //the last key in the left-hand page is "pushed up" to the parent.
    parentEntry.setKey(midEntry.getKey());
    parent.updateEntry(parentEntry);// parentEntry的child指针并没有变

    // Be sure to update the parent pointers of all children in the entries that were
        moved.
    updateParentPointers(tid,dirtypages,page);
    updateParentPointers(tid,dirtypages,leftSibling);
  }
```

3.3.2 mergeInternalPages

合并内节点与合并叶节点不同的地方同样是**需要先将父节点的 entry pull down**。而后操作与叶节点基本一致，将右兄弟中所有元素全部移动到左兄弟中。还有一点不同的是，每个内节点都有孩子节点，所以最后还需要调用 updateParentPointers **调整 page 的孩子指针**。

4 Extra Credit

4.1 BTreeReverseScan

BTreeReverseScan 参照 BTreeScan 进行设计，观察整个类，需要修改的地方仅在 reset 函数。

reset

```
// TODO: implement reverseIterator & reverseIndexIterator
    if(ipred == null) {
        this.it = ((BTreeFile)
            Database.getCatalog().getDatabaseFile(tableid)).reverseIterator(tid);
    }
    else {
        this.it = ((BTreeFile)
            Database.getCatalog().getDatabaseFile(tableid)).reverseIndexIterator(tid,
            ipred);
    }
```

4.2 reverseIterator & reverseIndexIterator

在 BTreeFile 中参照 iterator 和 indexIterator 设计 reverseIterator 和 reverseIndexIterator，用于 BTreeReverseScan 调用。

均采用内部类，因为 BTreeInternalPage 和 BTreeLeafPage 均实现有 reverseIterator，所以只需将原来 curp.iterator 换成 curp.reverseIterator 即可。

reverseIndexIterator 参照 indexIterator，需要使用谓词来筛选 tuple，**对偶的的将原来的判断条件进行互换即可**，如大于变为小于。

另一方面，indexIterator 需要调用 findLeafPage 去获取第一页，所以对应的需要实现 reverseFindLeafPage 去获取最后一页。实现上同 findLeafPage 一致，对偶的修改判断条件，递归地查找符合条件的 right-most 的节点。

4.3 BTreeReverseScanTest

BTreeReverseScanTest 参照 BTreeScanTest 设计，将 BTreeScan 全部换成 BTreeReverseScan 进行测试。

需要特别主要的是，在测试时，**会调用 assertEquals，一个个去判断使用 BTreeReverseScan 生成的元素是否与插入的元素相等**。插入的元素保存在一个 collection 里，在 BTreeScanTest 时，将这个集合从小到大排序，然后每次取出一个与 BTreeReverseScan 迭代出来的元素比较，顺序一致即通过。

在 BTreeReverseScanTest 这，需要将原来的顺序倒过来，即从大到小排序。很容易想到修改 compare 函数，令小于时返回 1，大于时返回-1。

但在**最终测试时是无法通过**的，原因是调用的 **sort 函数是稳定排序**，当 t1==t2 时位置不进行改变，先进入的元素排在前面，这在 BTreeScanTest 里是正确的，但在此处则得完全反向过来。于是修改判断条件，当 t1==t2 时返回-1。

compare

```
public int compare(ArrayList<Integer> t1, ArrayList<Integer> t2) {
    // TODO: reverse
    int cmp = 0;
    // 调用的是collection的sort函数，原本的是：小->-1，放前面；大->1，放后面；
    // 相等的不变，稳定排序！
```

```
6       // 现在需要reverse：小的->1，放后面...但有个问题：稳定排序
7       // 用户请求才用：归并排序&插入排序，判断>0则swap
8       // 默认使用的是TimSort&binarySort，判断条件是<0
9       // 相等的元素，先进来的reverse之后要排到后面->等于的情况也需要进行交换位置!
10      if(t1.get(keyField) < t2.get(keyField)) {
11          cmp = 1;
12      }
13      else if(t1.get(keyField) >= t2.get(keyField)) {
14          cmp = -1;
15      }
16      return cmp;
17  }
```

不设置 <= 条件是因为通过查看 sort 底层得知，使用的是 **TimSort**，当判断条件小于 0 时进行交换，所以设置 >= 时返回-1。

除此之外，还有一种方法是不修改排序函数，让 sort 函数从小到大排序，排完以后手动将其全部反向一遍。

5 Git Commit History

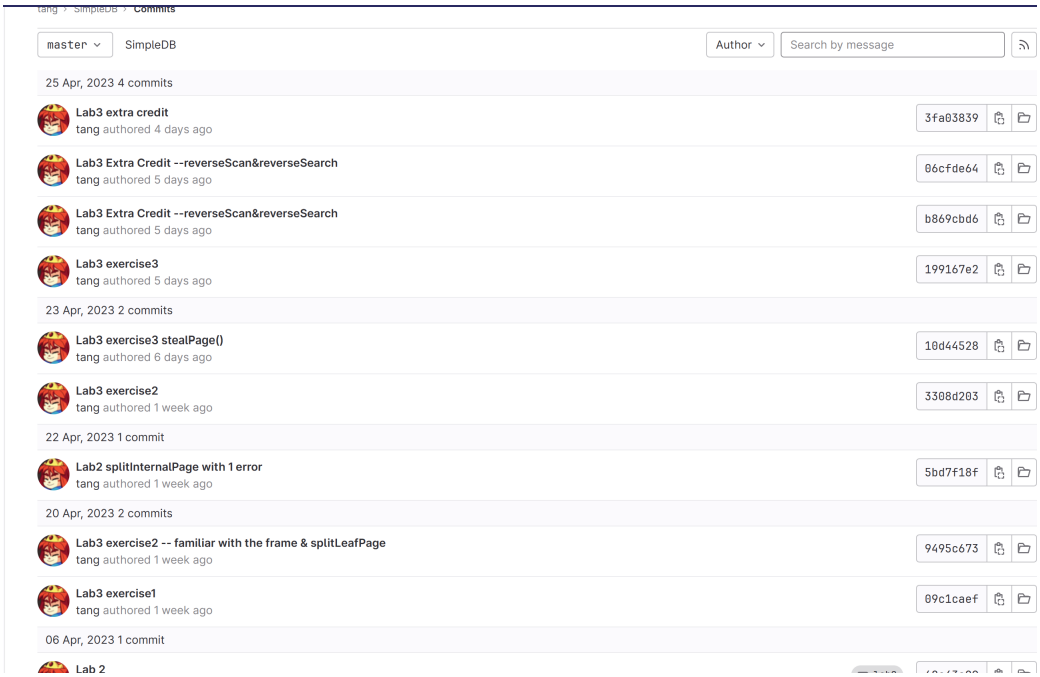

图 5.1: Git Commit History