南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

数据库系统实验报告

SimpleDB Lab3

王茂增

年级：2021级

专业：计算机科学与技术

指导教师：袁晓洁

2023 年 5 月 4 日

目录

一、 实验简介

本次实验的主要内容是接着上次Lab 1和Lab 2中实现的部分，完成BTreeFile的编写，并通过单元测试来验证代码的正确性。下面列出了每一个部分实现的主要功能，具体的思路以及代码实现将在实验详解中介绍。

- **Exercise 1：**实现findLeafPage方法
- **Exercise 2：**实现叶子结点和内部节点的分割方法
- **Exercise 3：**实现叶子结点和内部节点的偷取与合并方法
- **Consider Question：**编写BTreeReverseScan类和BTreeReverseScanTest类，对表进行反向遍历并测试正确性

二、 实验详解

（一） Exercise 1

1. 设计思路

在Exercise 1中，需要实现findLeafPage()方法，来根据键值查找页面。当键值不存在时，返回最左端叶节点；当键值存在且存在该键值对应页面时，返回含有该键值的页面中最左边的页面，否则返回索引小于键值的页面中键值最大的页面。

为实现该方法，我们需要指定节点开始逐层遍历，每次遍历子节点中键值大于等于查找键值的第一个页面，直到查找到叶节点。详细的实现步骤见重难点部分。

2. 重难点

1. **findLeafPage()方法**

   该方法首先判断是否已找到叶子结点，若找到则直接返回。否则则先得到该节点entry的迭代器，当查找键值为空时在最左端子节点继续查找；当不为空时遍历子节点，找到子节点中键值大于等于查找键值的第一个页面，并在该页面中继续查找。通过不断递归，找到目标页面。主要代码如下：

findLeafPage()方法

```
if (pid.pgcateg() == BTreePageId.LEAF) {
        //当为叶节点时直接返回
        return (BTreeLeafPage) this.getPage(tid, dirtypages, pid, perm);
}

//得到该节点词条的迭代器
BTreeInternalPage page = (BTreeInternalPage) this.getPage(tid, dirtypages,
    pid, perm);
Iterator<BTreeEntry> entries = page.iterator();

if (f == null) {
        //查找空值时返回最左边的叶节点
```

```
            return findLeafPage(tid, dirtypages, entries.next().getLeftChild(),
                perm, f);
        }

        while (true) {
            BTreeEntry entry = entries.next();
            if (entry.getKey().compare(Op.GREATER_THAN_OR_EQ, f)) {
                //当前节点键值大于等于查找键值时在左子节点继续查找
                return findLeafPage(tid, dirtypages, entry.getLeftChild(),
                    perm, f);
            }else if(!entries.hasNext()){
                //当左边的词条的键值都小于查找键值时在最后一个词条的右子节点继续查找
                return findLeafPage(tid, dirtypages, entry.getRightChild(),
                    perm, f);
            }
        }
```

（二） Exercise 2

1. 设计思路

在Exercise 2中，为了实现BTreeFile的insert操作，我们只需实现在叶页面或者内部页面满时的分割操作即可，即实现splitLeafPage方法和splitInternalPage方法。

上述方法的基本思路是创建一个空的页面，然后将一半的数据转移到新页面中，并进行可能的更新兄弟节点指针、在父节点中插入词条、在子节点中删除词条、创建一个新的根节点、分割父节点、将父节点上锁、更新父节点指针等操作，具体实现见重难点部分。

2. 重难点

1. **splitLeafPage方法**

为了实现分割叶页面，我们首先要在溢出页面的右边新建一个页面，然后将溢出页面的后二分之一个元组移到右页面中，用类似双向链表插入的方式将右页面插入叶子结点的左右兄弟链表中。然后我们需要根据右页面的第一个元组的键生成词条并加入父节点，然后更新父节点中entry的父节点指针。该步骤可能需要创建一个新的根节点、分割父节点或者将父节点上锁并返回，需要通过内置的getParentWithEmptySlots方法实现实现上述功能；需要通过内置的updateParentPointers更新父节点的子节点中的父节点指针。最后返回插入元组所在页面。具体代码如下：

splitLeafPage方法

```
BTreeLeafPage rightPage = (BTreeLeafPage) getEmptyPage(tid, dirtypages,
    BTreePageId.LEAF);
Iterator<Tuple> tuples = page.reverseIterator();

//将溢出页面的后二分之一个元组移到右页面中
//不需要按照顺序插入，因为插入元组方法内部保证了元组的有序性
for(int i=page.getNumTuples()/2; i>0; i--)
{
```

```
            Tuple tuple = tuples.next();
            page.deleteTuple(tuple);
            rightPage.insertTuple(tuple);
    }

    //将新节点与左右节点连接起来
    if(page.getRightSiblingId() != null)
    {
            BTreePageId oldRightId = page.getRightSiblingId();
            BTreeLeafPage oldRightPage = (BTreeLeafPage) getPage(tid,
                dirtypages, oldRightId, Permissions.READ_WRITE);
            oldRightPage.setLeftSiblingId(rightPage.getId());
    }
    rightPage.setLeftSiblingId(page.getId());
    rightPage.setRightSiblingId(page.getRightSiblingId());
    page.setRightSiblingId(rightPage.getId());

    //根据右边页面的第一个元组的键生成词条并加入父节点
    Field index = rightPage.iterator().next().getField(keyField);
    BTreeEntry entry = new BTreeEntry(index, page.getId(), rightPage.getId());
    BTreeInternalPage parentPage = getParentWithEmptySlots(tid, dirtypages,
        page.getParentId(), index);
    parentPage.insertEntry(entry);
    updateParentPointers(tid, dirtypages, parentPage);//更新父指针

    return (field.compare(Op.GREATER_THAN_OR_EQ, index)? rightPage:page);
```

2. **splitInternalPage方法**

为了实现分割内部页面，我们采用的方法与分割叶页面时类似。不同之处在于并不需要更新兄弟节点指针；在将中间词条提升到父节点时需要删除原有词条及其右孩子；以及需要更新parentPage和rightPage两个页面子节点中的父节点指针。由于该方法的实现与splitLeafPage方法类似，故在本报告中不给出代码。

（三） Exercise 3

1. 设计思路

在Exercise 3中，为了实现BTreeFile的Delete操作，我们只需实现在叶页面或者内部页面数据过少时进行的Steal或Merge操作即可。

对于Steal操作，我们首先需要偷取目标页面的数据给下溢页面使两者的数据数目基本一致，然后更新父节点中对应entry的键值。具体实现方式见重难点。

对于Merge操作，我们首先将左页面的元组全部移动到右页面，然后删除右节点。另外，还可能进行将父节点某个entry加入子节点中、更新左右兄弟指针、删除右页面、删除父节点对应entry等操作，具体实现方式见重难点部分。

2. 重难点

1. stealFromLeafPage方法

该方法首先通过isRightSibling参数确定偷取左节点的数据还是右节点的数据，然后偷取数据使两者的元组数目基本一致，最后将新的键值更新到父节点的词条中。

stealFromLeafPage方法

```
Iterator<Tuple> tupleIterator = isRightSibling? sibling.iterator(): sibling
    .reverseIterator();
if(tupleIterator==null || !tupleIterator.hasNext())
throw new DbException("no sibling or sibling has no tuple");

//使偷取之后两者元组数相同
int numSteal = (sibling.getNumTuples() - page.getNumTuples())/2;
Tuple tempTuple = null;
for(int i=0; i<numSteal; ++i)//进行元组的偷取
{
        tempTuple = tupleIterator.next();
        sibling.deleteTuple(tempTuple);
        page.insertTuple(tempTuple);
}

//将新的键值更新到父节点对应的词条中
assert tempTuple != null;
entry.setKey(tempTuple.getField(keyField));
parent.updateEntry(entry);
```

2. stealFromLeftInternalPage方法

该方法首先将父节点的词条移动到溢出页面，并接上两个子节点。然后将左兄弟节点的词条取出，插入溢出页面，使得偷取之后两者词条数目大致相等。最后，将左兄弟的最后一个词条的键值作为在父节点中对应entry的键值。

stealFromLeftInternalPage方法

```
Iterator<BTreeEntry> entryIterator = leftSibling.reverseIterator();
if(entryIterator==null || !entryIterator.hasNext())
        throw new DbException("left sibling has no entry");

//将父节点的词条移动到溢出页面，并接上两个子节点
BTreeEntry moveEntry = entryIterator.next();
BTreeEntry center = new BTreeEntry(parentEntry.getKey(), moveEntry.
    getRightChild(), page.iterator().next().getLeftChild());
page.insertEntry(center);

//使偷取之后两者元组数相同
int numSteal = (leftSibling.getNumEntries() - page.getNumEntries())/2;
//将兄弟节点的词条取出，插入溢出页面
for(int i=0; i<numSteal-1; ++i)
{
```

```
            leftSibling.deleteKeyAndRightChild(moveEntry);
            page.insertEntry(moveEntry);
            moveEntry = entryIterator.next();
}

//将此时左兄弟的最后一个词条的键值作为父节点的键值
leftSibling.deleteKeyAndRightChild(moveEntry);
parentEntry.setKey(moveEntry.getKey());
parent.updateEntry(parentEntry);
updateParentPointers(tid, dirtypages, page);//更新下溢节点中词条的父节点指针
```

3. **stealFromRighttInternalPage方法**

该方法与stealFromLeftInternalPage方法类似，故不再介绍。

4. **mergeLeafPages方法**

将叶节点合并，首先将左页面的元组全部移动到右页面，然后将右节点的右兄弟与左节点相连，并删除右节点。之后，将右节点对应的页面设置为空，并在父节点中删除这一词条。具体代码如下所示：

mergeLeafPages方法

```
//将左页面的元组全部移动到右页面
Iterator<Tuple> tupleIterator = rightPage.iterator();
while(tupleIterator.hasNext())
{
            Tuple tuple = tupleIterator.next();
            rightPage.deleteTuple(tuple);
            leftPage.insertTuple(tuple);
}

//在删除右节点前，先将右节点的右兄弟与左节点相连
if(rightPage.getRightSiblingId() != null)
{
            BTreeLeafPage rightSibling = (BTreeLeafPage) getPage(tid,
                dirtypages, rightPage.getRightSiblingId(), Permissions.
                READ_WRITE);
            rightSibling.setLeftSiblingId(leftPage.getId());
}
leftPage.setRightSiblingId(rightPage.getRightSiblingId());

//将右节点对应的页面设置为空，并在父节点中删除这一词条
setEmptyPage(tid, dirtypages, rightPage.getId().getPageNumber());
deleteParentEntry(tid, dirtypages, leftPage, parent, parentEntry);
```

5. **mergeInternalPages方法**

将内部节点合并，首先要将父节点中对应的词条插入左页面中，然后将右页面的词条全部加入左页面，最后将右节点对应的页面设置为空，更新左节点词条中的父节点指针，并在父节点中删除这一词条。

stealFromLeftInternalPage方法

```
Iterator<BTreeEntry> tupleIterator = rightPage.iterator();

//先将父节点中对应的词条插入左页面中
BTreeEntry center = new BTreeEntry(parentEntry.getKey(), leftPage.
    reverseIterator().next().getRightChild(),
rightPage.iterator().next().getLeftChild());
leftPage.insertEntry(center);

//将右页面的词条全部加入左页面
while(tupleIterator.hasNext())
{
        BTreeEntry entry = tupleIterator.next();
        rightPage.deleteKeyAndLeftChild(entry);
        leftPage.insertEntry(entry);
}

//将右节点对应的页面设置为空，并在父节点中删除这一词条
setEmptyPage(tid, dirtypages, rightPage.getId().getPageNumber());
updateParentPointers(tid, dirtypages, leftPage);
deleteParentEntry(tid, dirtypages, leftPage, parent, parentEntry);
```

（四） Consider Question：

1. 设计思路

在本次思考题中，需要实现反向扫描类并编写测试类。为了实现这一目的，我们首先需要实现findReverseLeafPage方法。然后我们需要以BTreeFile中已经实现的正向迭代器为蓝本，实现BTreeSearchIterator和BTreeSearchReverseIterator迭代器，最后实现BTreeReverseScan类。在成功实现BTreeReverseScan类之后，我们还需要编写BTreeReverseScanTest类对反向扫描类进行测试。

findReverseLeafPage方法与findLeafPage方法类似。不过会在查找键值为空时返回最后边的叶节点；在不为空时遍历子节点，找到子节点中键值小于等于查找键值的第一个页面，并在该页面中继续查找。

BTreeSearchIterator迭代器会反向遍历存储的元组，而BTreeSearchReverseIterator迭代器会根据给定谓词进行反向遍历。上述方法通过每个页面中的反向迭代器进行遍历，在页面遍历完成时会对左兄弟页面继续进行遍历，直到遍历完成。

BTreeReverseScan类通过上述两个迭代器实现遍历操作，并根据构造函数中传入的谓词是否为null判断使用BTreeSearchIterator迭代器还是BTreeSearchReverseIterator迭代器。

BTreeReverseScanTest类会分别对遍历、重置、谓词+重置、读取页面功能进行测试。

2. 重难点

1. **BTreeFileReverseIterator迭代器**

   对于该迭代器的open方法，我们首先找到最右边的叶子页面，然后将it迭代器赋值为该页面的反向迭代器，完成open方法。

BTreeFileReverseIterator.open()方法

```
BTreeRootPtrPage rootPtr = (BTreeRootPtrPage) Database.getBufferPool().
    getPage(
tid, BTreeRootPtrPage.getId(f.getId()), Permissions.READ_ONLY);
BTreePageId root = rootPtr.getRootId();
curp = f.findReverseLeafPage(tid, root, Permissions.READ_ONLY, null);
it = curp.reverseIterator();
```

对于该迭代器的readNext方法，我们首先检测it迭代器，当it存在但是已遍历完该页面时将迭代器置空。当it为空时，我们需要不断遍历找到当前页面的左兄弟页面，直到找到含有数据的页面或者到达最左边页面。经过上述两步后，若it为空说明已经遍历完所有叶子结点，返回空；否则返回下一个元组。

BTreeFileReverseIterator.readNext()方法

```
//当迭代器存在但是已遍历完该页面时将迭代器置空
if (it != null && !it.hasNext())
        it = null;

//不断遍历，直到找到可用的页面的迭代器或遍历完所有页面
while (it == null && curp != null) {
        BTreePageId nextp = curp.getLeftSiblingId();
        if (nextp == null) {
                curp = null;//当页面没有左兄弟时说明遍历结束
        } else {
                curp = (BTreeLeafPage) Database.getBufferPool().getPage(
                        tid,nextp,Permissions.READ_ONLY);//若含左兄弟则读取
                it = curp.reverseIterator();//得到该页面的反向迭代器
                if (!it.hasNext())
                it = null;//若该页面不含元素，则继续遍历下一页面
        }
}

//此时迭代器为空说明已经遍历完所有叶子结点，返回空
if (it == null)
return null;

//此时迭代器已经含有下一元素，返回下一个元组
return it.next();
```

2. **BTreeSearchReverseIterator迭代器**

该迭代器在BTreeFileReverseIterator迭代器的基础上加入了谓词功能，会得到一个满足谓词的所有元素的迭代器。由于与BTreeFileReverseIterator迭代器类似，故在本报告中不再给出实现代码。

3. **BTreeReverseScan类**

该类通过上述两个迭代器实现遍历操作，并根据构造函数中传入的谓词是否为null判断使用BTreeSearchIterator迭代器还是BTreeSearchReverseIterator迭代器。由于实现代码

比较简单，故在本报告中不再给出。

4. **TupleComparator.compare方**

对于该方法，在输入的两个数组的key field相等时返回0，小于时返回1，大于时返回-1，以实现逆序排序。

BTreeFileReverseIterator.open()方法

```
int cmp = 0;
if (t1.get(keyField) < t2.get(keyField)) {
        cmp = 1;
} else if (t1.get(keyField) > t2.get(keyField)) {
        cmp = -1;
}
return cmp;
```

5. **testRewindPredicates方法**

对于该方法，我们首先随机选择键值，然后随机生成有三个属性，1000条记录的表和对应的数组，并对数据进行逆序排序。然后，我们在数组中遍历得出等于谓词条件下的答案数组tuplesFiltered，并通过BTreeReverseScan类进行遍历，判断反向扫描类是否能正确处理等于谓词。然后进行重置并重复上述操作，以判断重置后反向扫描类是否能正确处理等于谓词。除此之外，本方法还测试了大于谓词和小于谓词，由于实现方法类似，故不在本报告中给出代码。

BTreeFileReverseIterator.open()方法

```
ArrayList<ArrayList<Integer>> tuples = new ArrayList<ArrayList<Integer>>();
int keyField = r.nextInt(3);
BTreeFile f = BTreeUtility.createRandomBTreeFile(3, 1000, null, tuples,
    keyField);
Collections.sort(tuples, new BTreeReverseScanTest.TupleComparator(keyField)
    );//此时数组根据键属性逆序存
    储

//测试等于谓词
TransactionId tid = new TransactionId();
ArrayList<ArrayList<Integer>> tuplesFiltered = new ArrayList<ArrayList<
    Integer>>();
IndexPredicate ipred = new IndexPredicate(Predicate.Op.EQUALS, new IntField
    (r.nextInt(BTreeUtility.MAX_RAND_VALUE)));
Iterator<ArrayList<Integer>> it = tuples.iterator();
while (it.hasNext()) {//首先将满足等于谓词的数组存储在数组中
        ArrayList<Integer> tup = it.next();
        if (tup.get(keyField) == ((IntField) ipred.getField()).getValue())
            {
                tuplesFiltered.add(tup);
        }
}

```

```
18 BTreeReverseScan reverseScan = new BTreeReverseScan(tid, f.getId(), "table"
       , ipred);
19 reverseScan.open();
20
21 //遍历，判断反向扫描类是否能正确处理等于谓词
22 for (int i = 0; i < tuplesFiltered.size(); ++i) {
23         assertTrue(reverseScan.hasNext());
24         Tuple t = reverseScan.next();
25         assertEquals(tuplesFiltered.get(i).get(keyField), SystemTestUtil.
               tupleToList(t).get(keyField));
26 }
27
28 reverseScan.rewind();//重置迭代器
29
30 //遍历，判断重置后反向扫描类是否能正确处理等于谓词
31 for (int i = 0; i < tuplesFiltered.size(); ++i) {
32         assertTrue(reverseScan.hasNext());
33         Tuple t = reverseScan.next();
34         assertEquals(tuplesFiltered.get(i).get(keyField), SystemTestUtil.
               tupleToList(t).get(keyField));
35 }
36 reverseScan.close();
```

6. **testReadPage方法**

对于该方法，我们首先随机生成有2个属性，30*502条记录的表和对应的数组，并对数据进行逆序排序,给表添加记录读取页面数目的功能。然后，我们在数组中遍历得出等于谓词条件下的答案数组tuplesFiltered，并通过BTreeReverseScan类对表进行遍历，将遍历结果与tuplesFiltered比较，判断反向扫描类是否能正确处理等于谓词。并且，我们还需要检测读取页面的数目是否为3或4，包括一个根指针页面、根页面以及一个或两个叶子页面。除此之外，本方法还测试了大于谓词和小于谓词，由于实现方法类似，故不在本报告中给出代码。

BTreeFileReverseIterator.open()方法

```
1  final int LEAF_PAGES = 30;
2
3  ArrayList<ArrayList<Integer>> tuples = new ArrayList<ArrayList<Integer>>();
4  int keyField = 0;
5  BTreeFile f = BTreeUtility.createBTreeFile(2, LEAF_PAGES * 502, null,
       tuples, keyField);
6  Collections.sort(tuples, new BTreeReverseScanTest.TupleComparator(keyField)
       );
7  TupleDesc td = Utility.getTupleDesc(2);
8  BTreeReverseScanTest.InstrumentedBTreeFile table = new BTreeReverseScanTest
       .InstrumentedBTreeFile(f.getFile(), keyField, td);
9  Database.getCatalog().addTable(table, SystemTestUtil.getUUID());
10
11 //测试等于谓词
```

```
12  TransactionId tid = new TransactionId();
13  ArrayList<ArrayList<Integer>> tuplesFiltered = new ArrayList<ArrayList<
        Integer>>();
14  IndexPredicate ipred = new IndexPredicate(Predicate.Op.EQUALS, new IntField
        (r.nextInt(LEAF_PAGES * 502)));
15  Iterator<ArrayList<Integer>> it = tuples.iterator();
16  while (it.hasNext()) {//先将满足等于谓词的数组存储到数组中
17          ArrayList<Integer> tup = it.next();
18          if (tup.get(keyField) == ((IntField) ipred.getField()).getValue())
                {
19                  tuplesFiltered.add(tup);
20          }
21  }
22
23  Database.resetBufferPool(BufferPool.DEFAULT_PAGES);
24  table.readCount = 0;
25  BTreeReverseScan reverseScan = new BTreeReverseScan(tid, f.getId(), "table"
        , ipred);
26  SystemTestUtil.matchTuples(reverseScan, tuplesFiltered);
27  //读取的页面数应该为三或四，包括一个根指针页面、根页面以及一个或两个叶子页面
28  assertTrue(table.readCount == 3 || table.readCount == 4);
```

三、 总结

本次实验中，我完成了SimpleDB的Lab 3部分。本次实验在Lab 1和Lab 2的基础上，根据BTreeFile类中已经完成的一系列基础操作，实现了findLeafPage方法、分割方法、偷取与合并方法，并编写了BTreeReverseScan类和BTreeReverseScanTest类，对表进行反向遍历并测试正确性。

通过本次实验，我实现了B+树的基本功能，直观了解了B+树的查找、插入、删除算法的精髓。同时，我也自己编写了BTreeReverseScanTest类，深入理解了测试是如何进行的。

四、 提交记录

```
commit 9ebbf050ad867371ff7d04c59155fb4eb17bae26 (HEAD -> master, origin/master, origin/HEAD, github/master)
Author: mzwangg <mzwangg@gmail.com>
Date:   Wed May 3 23:04:55 2023 +0800

    Lab_3 Consider_Question

commit c9ead74be30a96c0d7a38e2674996e126152e608
Author: mzwangg <mzwangg@gmail.com>
Date:   Mon May 1 23:31:34 2023 +0800

    Lab_3 Exercise3

commit feb29b033ac6311315fc09ad8ed7854a60c243d0
Author: mzwangg <mzwangg@gmail.com>
Date:   Thu Apr 27 07:15:09 2023 +0800

    Lab_3 Exercise2

commit 15be560a4e22b1032bbb736b09947dd498a92be6
Author: mzwangg <mzwangg@gmail.com>
Date:   Wed Apr 26 23:05:36 2023 +0800

    Lab_3 Exercise1
```

图 1: 提交记录

五、 源代码

https://gitlab.com/mzwangg/SimpleDB