# SimpleDB

一 实验任务

A basic database system

Heap files (Lab 1)

Buffer Pool (Labs 1-6)

Basic Operators (Labs 1 & 2)

Scan, Filter, JOIN, Aggregate

Query optimizer (Lab 3)

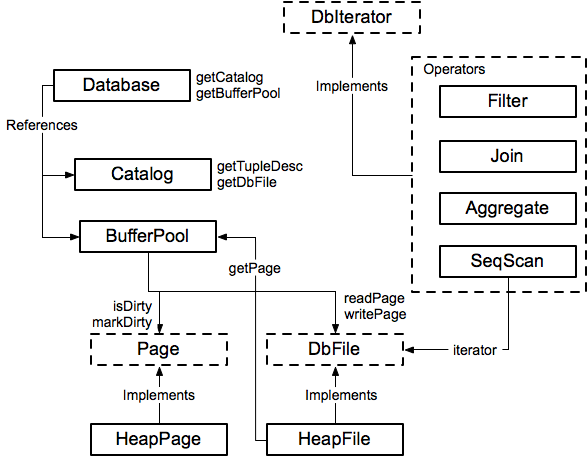
Transactions (Lab 4)

B-Tree Indexes (Lab 5)

Recovery (Lab 6)

Javadoc is your friend!

Module Diagram



Lab1

### **LAB1**

#### **Exercise 1**

Tuples in SimpleDB are quite basic. They consist of a collection of `Field` objects, one per field in the `Tuple`. `Field` is an interface that different data types (e.g., integer, string) implement. `Tuple` objects are created by the underlying access methods (e.g., heap files, or B-trees), as described in the next section. Tuples also have a type (or schema), called a \_tuple descriptor\_, represented by a `TupleDesc` object. This object consists of a collection of `Type` objects, one per field in the tuple, each of which describes the type of the corresponding field.

****Implement the skeleton methods in:****

* src/java/simpledb/storage/TupleDesc.java
* src/java/simpledb/storage/Tuple.java

At this point, your code should pass the unit tests TupleTest and TupleDescTest. At this point, modifyRecordId() should fail because you havn't implemented it yet.

要求：完成Tuple和TupleDesc类的代码补充，成功通过TupleTest以及TupleDescTest的测试

-------（完成）

理解：

Tuple就是元组，包含了一条记录的所有域，并且包含了一个表示其模式（schema)的TupleDesc。

*//元组包含的数据****private*** ArrayList<Field> **fields** ;  
*//元组描述符 形容数据的类型及名称****private*** TupleDesc **tuple\_descriptor**;  
*//元组id号，用于disk上的定位****private*** RecordId **recordId**;

TupleDesc是元组的模式，包含了所表示的元组的模式，用集合保存其域类型以及域名称。

*//元组描述符的条目集合****private*** ArrayList<TDItem> **TDItems**;

感悟：只要理解了Tuple以及TupleDesc的含义，就可以迅速根据注释的提示完善代码，相对来说比较容易。而今天的主要时间花在了对项目环境的调试，我查了好久资料才搞清楚为什么图标都是j而不是c，通过修改project structure的project和module配置，并且添加编译的输出路径，最后项目终于可以正常写了。

#### **Exercise 2**

The catalog (class Catalog in SimpleDB) consists of a list of the tables and schemas of the tables that are currently in the database. You will need to support the ability to add a new table, as well as getting information about a particular table. Associated with each table is a TupleDesc object that allows operators to determine the types and number of fields in a table.

The global catalog is a single instance of Catalog that is allocated for the entire SimpleDB process. The global catalog can be retrieved via the method Database.getCatalog(), and the same goes for the global buffer pool ( using Database.getBufferPool()).

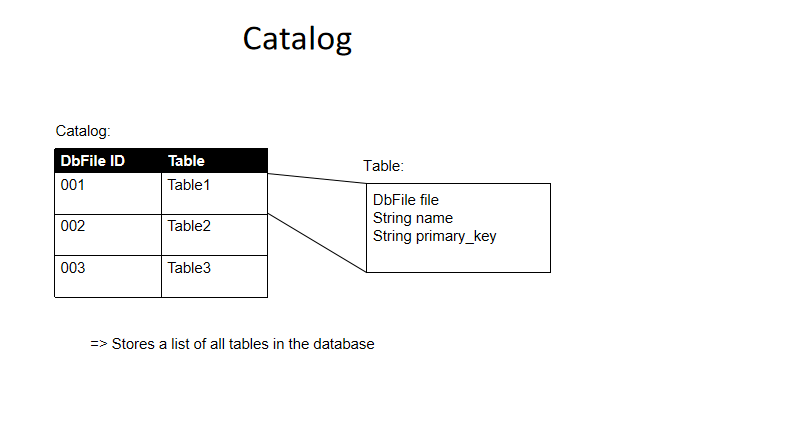
****Implement the skeleton methods in:****

* src/java/simpledb/common/Catalog.java

At this point, your code should pass the unit tests in CatalogTest.

要求：完成Catalog类的代码补充，通过CatalogTest测试。---（完成）

理解：Catalog是数据库下的一个目录，其中包含了许多表，所以要在该类内创建一个Table类型的表，表包含了表名，主键，表文件。而Catalog内需要用一个集合结构保存表及其编号，这里选择了hashmap<Integer,Table>来保存。



*//Catalog内的众多表集合*HashMap<Integer,Table> **tables**;

*//table类代表一个目录下的表****public class*** Table  
 *//表名* String **tableName**;  
 *//表文件 其子实现类中包含该表的元组描述符* DbFile **dbFile**;  
 *//主键值* String **pkeyfield**;

感悟：一开始没有弄明白Catalog所表示的含义，以为直接创建一个tuple类的集合就可以了。所以，之后在编写代码前先仔细浏览该类中的其他函数，从这些函数推理出所需要补充的内容。

2022年3月10时22分2秒

#### **Exercise 3**

##### **BufferPool**

The buffer pool (class `BufferPool` in SimpleDB) is responsible for caching pages in memory that have been recently read from disk. All operators read and write pages from various files on disk through the buffer pool. It consists of a fixed number of pages, defined by the `numPages` parameter to the `BufferPool` constructor. In later labs, you will implement an eviction policy. For this lab, you only need to implement the constructor and the `BufferPool.getPage()` method used by the SeqScan operator. The BufferPool should store up to `numPages` pages. For this lab, if more than `numPages` requests are made for different pages, then instead of implementing an eviction policy, you may throw a DbException. In future labs you will be required to implement an eviction policy.

The Database class provides a static method, Database.getBufferPool(), that returns a reference to the single BufferPool instance for the entire SimpleDB process.

****Implement the**getPage()**method in:****

* src/java/simpledb/storage/BufferPool.java

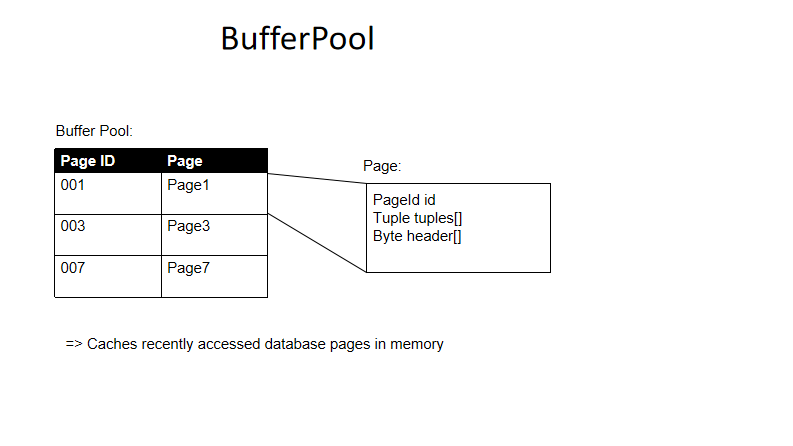
We have not provided unit tests for BufferPool. The functionality you implemented will be tested in the implementation of HeapFile below. You should use the DbFile.readPage method to access pages of a DbFile.

任务：补充BufferPool的构造函数以及getPage（） ---完成

理解：

BufferPool是缓冲池，用于保存从硬盘读取的文件的页，所以使用一个HashMap<PageId,Page>结构保存其页号以及页。在完成getPage（）函数时，要想向缓冲池中读取一个页，目前现需要判断读取的页是否在缓冲池中，在则读出，若不在，则判断缓冲池页数量是否还有剩余，有则通过数据库类Database获取pid的文件，在通过文件获取目标页，添加到缓冲池中，再返回目标页。

（因为未涉及到锁与事务的章节，暂时忽略条件中的其他参数）



*//缓冲池内的页*HashMap<PageId,Page> **pages**;

***public*** Page getPage(TransactionId tid, PageId pid, Permissions perm)  
 ***throws*** TransactionAbortedException, DbException {  
 *// some code goes here  
 //查询pageid* Page page = **pages**.get(pid);  
 *//若该页不在缓冲池中* ***if***(page==***null***){  
 *//若缓冲池页数大小到达限定大小 采取驱逐策略 这里先抛出异常* ***if***(**pages**.size()>***DEFAULT\_PAGES***){  
 ***throw new*** DbException(**"缓冲池页数超出限制"**);  
 }  
 *//从数据库目录中获取指定pid的文件* DbFile dbFile = Database.*getCatalog*().getDatabaseFile(pid.getTableId());  
 *//读出该文件的pid号页 加入到缓冲池中* page = dbFile.readPage(pid);  
 **pages**.put(pid,page);  
  
 }  
 ***return*** page;  
}

感悟：....

2022年3月3日11时15分

#### **Exercise 4**

A HeapFile object is arranged into a set of pages, each of which consists of a fixed number of bytes for storing tuples, (defined by the constant BufferPool.DEFAULT\_PAGE\_SIZE), including a header. In SimpleDB, there is one HeapFile object for each table in the database. Each page in a HeapFile is arranged as a set of slots, each of which can hold one tuple (tuples for a given table in SimpleDB are all of the same size). In addition to these slots, each page has a header that consists of a bitmap with one bit per tuple slot. If the bit corresponding to a particular tuple is 1, it indicates that the tuple is valid; if it is 0, the tuple is invalid (e.g., has been deleted or was never initialized.) Pages of HeapFile objects are of type HeapPage which implements the Page interface. Pages are stored in the buffer pool but are read and written by the HeapFile class.

SimpleDB stores heap files on disk in more or less the same format they are stored in memory. Each file consists of page data arranged consecutively on disk. Each page consists of one or more bytes representing the header, followed by the \_ page size\_ bytes of actual page content. Each tuple requires tuple size \* 8 bits for its content and 1 bit for the header. Thus, the number of tuples that can fit in a single page is:

\_tuples per page\_ = floor((\_page size\_ \* 8) / (\_tuple size\_ \* 8 + 1))

Where tuple size is the size of a tuple in the page in bytes. The idea here is that each tuple requires one additional bit of storage in the header. We compute the number of bits in a page (by mulitplying page size by 8), and divide this quantity by the number of bits in a tuple (including this extra header bit) to get the number of tuples per page. The floor operation rounds down to the nearest integer number of tuples (we don't want to store partial tuples on a page!)

Once we know the number of tuples per page, the number of bytes required to store the header is simply:

headerBytes = ceiling(tupsPerPage/8)

The ceiling operation rounds up to the nearest integer number of bytes (we never store less than a full byte of header information.)

The low (least significant) bits of each byte represents the status of the slots that are earlier in the file. Hence, the lowest bit of the first byte represents whether or not the first slot in the page is in use. The second lowest bit of the first byte represents whether or not the second slot in the page is in use, and so on. Also, note that the high-order bits of the last byte may not correspond to a slot that is actually in the file, since the number of slots may not be a multiple of 8. Also note that all Java virtual machines are [big-endian](http://en.wikipedia.org/wiki/Endianness).

****Implement the skeleton methods in:****

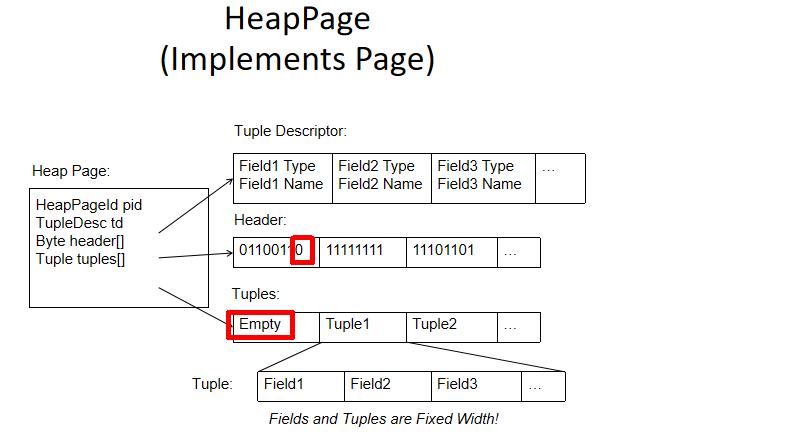
* src/java/simpledb/storage/HeapPageId.java
* src/java/simpledb/storage/RecordId.java
* src/java/simpledb/storage/HeapPage.java

Although you will not use them directly in Lab 1, we ask you to implement getNumEmptySlots() and isSlotUsed() in HeapPage. These require pushing around bits in the page header. You may find it helpful to look at the other methods that have been provided in HeapPage or in src/simpledb/HeapFileEncoder.java to understand the layout of pages.

You will also need to implement an Iterator over the tuples in the page, which may involve an auxiliary class or data structure.

At this point, your code should pass the unit tests in HeapPageIdTest, RecordIDTest, and HeapPageReadTest.

After you have implemented HeapPage, you will write methods for HeapFile in this lab to calculate the number of pages in a file and to read a page from the file. You will then be able to fetch tuples from a file stored on disk.



任务：

1\*完善RecordId以及HeapPageId类的代码。

2\*补充HeapPage类内的getNumEmptySlots（），getNumTuples（），getHeaderSize（）以及isSlotUsed（）函数，并且添加一个辅组类TupleIterator用来遍历该页表内的所有元组。

理解：

第一个任务较为简单只要根据初始化函数传入的参数分别创建其对于的数据参数即可

RecordId：表示单条元组的信息，包含了所在页面的信息，以及当前记录所在表中的编号（用于访问此记录）

***public class*** RecordId ***implements*** Serializable {  
  
 *//表示当前记录条属于的页面* ***private*** PageId **pageId**;  
 *//表示当前的记录条的编号* ***private int* tupleNo**;

HeapPageId：HeapPage的唯一标识符，表示当前页属于哪一个表，以及当前页所在文件的page编号

*//当前的HeapPage属于哪一个表****private int* tableId**;  
*//page编号****private int* pgNo**;

第二个任务较难

首先HeapPage表示一个HeapFile文件保存的页面，包含了许多重要信息。

*//当前页面的id号****final*** HeapPageId **pid**;  
*//当前页面的模式****final*** TupleDesc **td**;  
*//头部--所有元组的位图 对应位为1时说明该元组存在，为0则不存在****final byte***[] **header**;  
*//存放元组条信息****final*** Tuple[] **tuples**;  
*//当前元组的总数****final int* numSlots**;

其次，教义中的公式用来完善getHeaderSize函数以及getNumTuples函数

计算一页能容纳的元组tuple以及表示该元组的一位的数量

\_tuples per page\_ = floor((\_page size\_ \* 8) / (\_tuple size\_ \* 8 + 1))

计算位图（头部）的大小，即上述公式求得的数量除以8，就是位图的大小

headerBytes = ceiling(tupsPerPage/8)

接着，难点就是辅助类TupleIterator的编写，重点在于元组数组中可能存在分布不均匀的空元组，为了避免输出空元组，要在迭代器中对要输出的元组进行判断。同时，当非空元组都输出完成后，应该及时退出，所以要设置一个int类型的usedNum用来表示已经输出的非空元组，当全部输出完后，不再迭代。

*// tuple迭代器****private class*** TupleIterator ***implements*** Iterator<Tuple>{  
 *//当前遍历到达tuple数组坐标* ***private int* index** = 0;  
 *//当前遍历到的已经使用过的tuple数量 避免当tuple数组中非空元组访问完后继续访问* ***private int* usedNum**=0;  
 *//当前页面的所有非空元组数量* ***private int* tupleUsedNum**=getNumTuples()-getNumEmptySlots();  
  
 @Override  
 ***public boolean*** hasNext() {  
 ***return* index**<getNumTuples()&&**usedNum**<**tupleUsedNum**;  
 }  
  
 @Override  
 ***public*** Tuple next() {  
 ***if***(!hasNext()){  
 ***throw new*** NoSuchElementException(**"不存在该元组"**);  
 }  
 *//找到非空的元组* ***while***(!isSlotUsed(**index**)){  
 **index**++;  
 }  
  
 *//返回一个非空元组 都要usednum++* **usedNum**++;  
  
 *//index需要自增，避免一直访问自己* ***return* tuples**[**index**++];  
 }  
}

最后一个难点，isSlotUsed()函数的编写，虽然原理不难，但是不仔细就出错了，具体看注释。

***public boolean*** isSlotUsed(***int*** i) {  
 *// some code goes here  
 //判断第i个插槽是否以及被使用 根据header表示的位图进行判断  
 /\*每个字节的低位（最低有效位）代表文件中较早的插槽的状态。  
 因此，第一个字节的最低位表示页面中的第一个插槽是否正在使用。  
 第一个字节的第二低位表示页面中的第二个插槽是否正在使用，依此类推  
 。另外，请注意，最后一个字节的高位可能与文件中实际存在的插槽不对应，  
 因为插槽的数量可能不是8的倍数。还要注意，所有Java虚拟机都是big-endian。\*/* ***int*** indexInbyet = i/8;*//判断位于header的第几个字节* ***int*** location = i%8;*//位于字节从右往左的第几位  
 //将indexInbyte位置的byte与掩码1左移location位后的数字相与  
 // 若该数为0说明改为不为1 为false  
 //代码错误 无法检测出id编号的slot是否使用  
// int i1 = header[indexInbyet] & (1 << location);  
// if(i1==0) return false;  
// return true;* ***byte*** target = **header**[indexInbyet];  
 ***return*** (***byte***)(target<<(7-location))<0;

感悟：

该练习花费的主要时间都在调试上，函数设计中存在一些小错误，我通过对每个测试案例（主要是HeapPageTest)的挨个调试，发现了代码中存在的问题，对该类的理解更深入了。

2022年3月4日23时34分

#### **Exercise 5**

****Implement the skeleton methods in:****

* src/java/simpledb/storage/HeapFile.java

To read a page from disk, you will first need to calculate the correct offset in the file. Hint: you will need random access to the file in order to read and write pages at arbitrary offsets. You should not call BufferPool methods when reading a page from disk.

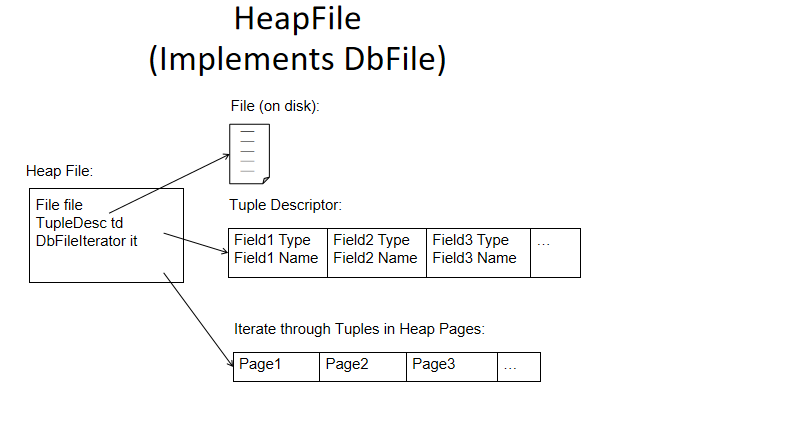
You will also need to implement the `HeapFile.iterator()` method, which should iterate through through the tuples of each page in the HeapFile. The iterator must use the `BufferPool.getPage()` method to access pages in the `HeapFile`. This method loads the page into the buffer pool and will eventually be used (in a later lab) to implement locking-based concurrency control and recovery. Do not load the entire table into memory on the open() call -- this will cause an out of memory error for very large tables.

At this point, your code should pass the unit tests in HeapFileReadTest.

任务：

完善HeapFile内的代码。

理解：



HeapFile代表存储的文件。所以需要保存文件，文件内的页数，以及该文件的描述符。一个文件就是一张表。

*//HeapFile保存的文件****private*** File **file**;  
*//该文件的表的描述符****private*** TupleDesc **tupleDesc**;  
*//代表文件中保存的页数****private int* pageNum**;

难点在于readPage（）函数和HeapFileIterator迭代器的编写。

ReadPage（）题目要求读取页面上需要能够随机读取。所以使用RandomAcessFile来获取文件类的指针，同时，通过要读取的文件内的页号乘以页大小就可以得到要读入的位置的指针，接着掉用read函数读取一页大小的字节进入题前准备好的data数组内即可。

***public*** Page readPage(PageId pid) {  
 *// some code goes here* Page page = ***null***;  
 ***byte*** data[] =***new byte***[BufferPool.*getPageSize*()];  
  
 *//随机访问file文件并读出* ***try*** (RandomAccessFile raf = ***new*** RandomAccessFile(getFile(), **"r"**)) {  
 *//根据要读去的页面号 获取在文件中的位置* ***int*** pos = pid.getPageNumber()\*BufferPool.*getPageSize*();  
 *//设置指针位置* raf.seek(pos);  
 raf.read(data,pos,BufferPool.*getPageSize*());  
 page = ***new*** HeapPage((HeapPageId) pid,data);  
 } ***catch*** (FileNotFoundException e) {  
 e.printStackTrace();  
 } ***catch*** (IOException e) {  
 e.printStackTrace();  
 }  
 ***return*** page;  
}

HeapFileIterator的编写就更难了。我阅读了别人的代码，才理解大致的想法。

首先要想迭代HeapFile文件内的元组就要判断要读取的页的编号，因为一个文件有多个页，同时还需要TranscationId，再添加一个iterator参数来保存每次要读的页的迭代器，所以设置如下变量。

***private class*** HeapFileIterator ***implements*** DbFileIterator{  
  
 *//file文件内每页的iterator* ***private*** Iterator<Tuple> **iterator**;  
 *//当前迭代到的页面* ***private int* pageNo**;  
 *//事务id* TransactionId **tid**;

同时设置一个getTupleIterator函数，传入需要的PageId信息，通过已有条件从缓冲池获取要读入的具体页的迭代器

***private*** Iterator<Tuple> getIterator(HeapPageId hpid) ***throws*** TransactionAbortedException, DbException {  
 *// 不能直接使用HeapFile的readPage方法，而是通过BufferPool来获得page，理由见readPage()方法的Javadoc* HeapPage page =(HeapPage) Database.*getBufferPool*().getPage(**tid**, hpid, Permissions.***READ\_ONLY***);  
 ***return*** page.iterator();  
}

剩下的next（），hasnext()只要对iterator的情况进行分类再合理返回即可，具体见源码。

感悟：

这次实验的难度上来一行档次，编写readPage函数时对随机输入流不熟悉，不知道该如何下手。还有HeapFileIterator，完全是一头雾水，不知道如何下手，究其原因还是对目前为止出现的类的性质与功能不太熟悉，所以后续完成练习6后要梳理一下具体的类及其联系。

2022年3月4日14时33分43秒

##### **2.6. Operators**

Operators are responsible for the actual execution of the query plan. They implement the operations of the relational algebra. In SimpleDB, operators are iterator based; each operator implements the DbIterator interface.

Operators are connected together into a plan by passing lower-level operators into the constructors of higher-level operators, i.e., by 'chaining them together.' Special access method operators at the leaves of the plan are responsible for reading data from the disk (and hence do not have any operators below them).

At the top of the plan, the program interacting with SimpleDB simply calls getNext on the root operator; this operator then calls getNext on its children, and so on, until these leaf operators are called. They fetch tuples from disk and pass them up the tree (as return arguments to getNext); tuples propagate up the plan in this way until they are output at the root or combined or rejected by another operator in the plan.

For this lab, you will only need to implement one SimpleDB operator.

#### **Exercise 6.**

****Implement the skeleton methods in:****

* src/java/simpledb/execution/SeqScan.java

This operator sequentially scans all of the tuples from the pages of the table specified by the tableid in the constructor. This operator should access tuples through the DbFile.iterator() method.

At this point, you should be able to complete the ScanTest system test. Good work!

You will fill in other operators in subsequent labs.

任务：

完善操作类SeqScan的代码

理解：

Sescan是扫描操作，用来遍历文件中的所有元组（非空）。所以需要以下参数

*//进行扫描的事务操作的id****private*** TransactionId **tid**;  
*//扫描的表id****private int* tableId**;  
*//扫描的表的别名****private*** String **tableAlias**;  
*//表的迭代器****private*** DbFileIterator **tableIterator**;

为了顺利遍历文件中的元组，只需要封装在HeapFile内完成的HeapFileIterator类即可，补充好相关代码就ok了。具体见源码

稍微难一点的地方在对文件表名起别名的地方（其实也不是很难），只需要获取文件的描述符，通过描述获取域名以及类型，按照要求和域名加上别名即可，再和类型重新一起构造一个描述符然后返回就好了。

*\** ***@return*** *the TupleDesc with field names from the underlying HeapFile,  
 \* prefixed with the tableAlias string from the constructor.  
 \* 返回一个描述符，该描述符所描述的表若需要将其域名重新修改，添加一个在  
 \* 此次扫描时加入的表别名，改为"别名.域名"的形式，当别名不存在时，改成  
 \* "null.域名"  
 \*/****public*** TupleDesc getTupleDesc() {  
 *// some code goes here  
 //获取需要修改的描述符* TupleDesc tupleDesc = Database.*getCatalog*().getTupleDesc(**tableId**);  
 ***int*** nums = tupleDesc.numFields();  
 *//重新补充创建一个新的描述所需要的数组* Type[] typeAr = ***new*** Type[nums];  
 String[] fieldAr = ***new*** String[nums];  
 ***for***(***int*** i=0;i<nums;i++){  
 typeAr[i] = tupleDesc.getFieldType(i);  
 String oldName = tupleDesc.getFieldName(i);  
 fieldAr[i]=(getAlias()==***null***?**"null."**:getAlias()+**"."**)+oldName;  
 }  
 ***return new*** TupleDesc(typeAr,fieldAr);  
  
}

感悟：

这个实验需要补充的内容难度不是很大，我碰到的主要问题是测试案例中的testSmall（）和testCache（）都通不过。我首先从testSmall（）测试进行调试，发现我在编写HeapFile内的Iterator的hasnext（）函数时，

忘了在以下代码中对pageNo进行++操作，导致了迭代器一直在读取文件中的同一页内容，而出现了错误。

*//iterator是整个文件的迭代器 所以当前上述条件都不满足时 需要将该迭代器更新 让他去迭代下一个页面的元组  
//如果当前迭代器迭代到的页面小于总页数****if***(**pageNo**<numPages()-1) {  
 **pageNo**++;  
 HeapPageId hpid = ***new*** HeapPageId(getId(), **pageNo**);  
 **iterator** = getIterator(hpid);  
 *//这里不能直接返回true 因为不知道下一个页面中是否含有有效元组 所以还需再次调用迭代器的hasnext()* ***return* iterator**.hasNext();

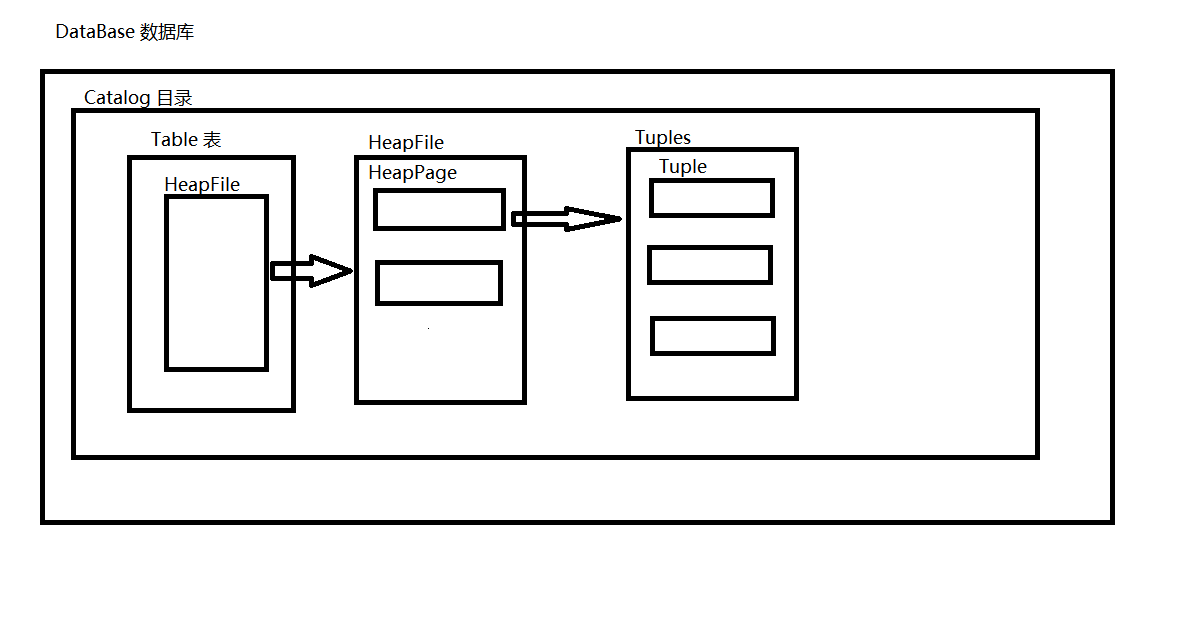
在解决了上述问题之后，我再次测试了testSmall(),结果在readPage()内外的读取文件内容时，把raf.read(data,0,data.**length**)写成了raf.read(data,pos,BufferPool.getPageSize());导致程序抛出边界异常，主要原因是我误解了该函数的意义。该函数是 public int read(byte b[], int off, int len) 打开位置（通过raf.seek()将指针放好），off为与指针的偏移量，len表示读取的内容长度大小。

*//根据要读去的页面号 获取在文件中的位置****int*** pos = pid.getPageNumber()\*BufferPool.*getPageSize*();  
*//设置指针位置*raf.seek(pos);  
raf.read(data,0,data.**length**);  
page = ***new*** HeapPage((HeapPageId) pid,data);

解决上述问题后，ScanTest所有测试案例都通过了。今天换了好长时间在调试上，难受啊！以后要细心点了。

2022年3月4日星期五21时20分19秒

总结：

主要类的大致包含关系（有点丑，大致理解就好）

### LAB2

Recall that SimpleDB OpIterator classes implement the operations of the relational algebra. You will now implement two operators that will enable you to perform queries that are slightly more interesting than a table scan.

Filter: This operator only returns tuples that satisfy a Predicate that is specified as part of its constructor. Hence, it filters out any tuples that do not match the predicate.

Join: This operator joins tuples from its two children according to a JoinPredicate that is passed in as part of its constructor. We only require a simple nested loops join, but you may explore more interesting join implementations. Describe your implementation in your lab writeup.

****Exercise 1.****

Implement the skeleton methods in:

* src/java/simpledb/execution/Predicate.java
* src/java/simpledb/execution/JoinPredicate.java
* src/java/simpledb/execution/Filter.java
* src/java/simpledb/execution/Join.java

At this point, your code should pass the unit tests in PredicateTest, JoinPredicateTest, FilterTest, and JoinTest. Furthermore, you should be able to pass the system tests FilterTest and JoinTest.

任务：完善上述类代码

理解：

一 下述两个类比较简单，只需要顺着函数填写代码即可。

Predicate为谓词 内部定义了一个比较枚举类，用来区分不同的比较符。

该类为了对选定的域进行比较，应当保存一个指定域索引的值，一个比较操作符，以及对比的操作数。

*//要比较的元组的域的索引****private int* fieldnum** ;  
*//操作符 定义比较的含义****private*** Op **op**;  
*//对比的操作数****private*** Field **operand**;

JoinPredicate与Predicate类基本相同 但是Join是要比较两个表的域，所以应包含两个表的比较域的索引，以及一个比较操作符。

*//连接的两个域的索引****private int* field1**;  
***private int* field2**;  
*//操作符****private*** Predicate.Op **op**;

二 下述两个操作相对难点，需要借鉴源码中以及写好的Project以及OrderBy，理解前面两个类后就可以很好的知道操作类编写的大致思路。

①Filter过滤器 依照断言筛选出合格的元组集合 并保存其输出的迭代器（仿照OrderBy）

*//断言****private*** Predicate **predicate**;  
*//要过滤的表的迭代器****private*** OpIterator **child**;  
*//保存过滤后符合断言的元组****private*** List<Tuple> **childTuples** = ***new*** ArrayList<>();  
*//元组描述符****private*** TupleDesc **tupleDesc**;  
*//符合断言的元组集合的迭代器****private*** Iterator<Tuple> **iterator**;

核心函数:

***public void*** open() ***throws*** DbException, NoSuchElementException,  
 TransactionAbortedException {  
 *// some code goes here  
 //打开访问的表的迭代器 才能进行迭代* **child**.open();  
 ***while***(**child**.hasNext()){  
 *//获取元组* Tuple newTuple =**child**.next();  
 *//对下一个元组进行断言 为真则加入要输出的元组* ***if***(**predicate**.filter(newTuple)){  
 **childTuples**.add(newTuple);  
 }  
 }  
 *//将符合条件的所有元组集合的迭代器设置为类内的迭代器* ***this***.**iterator** = **childTuples**.iterator();  
 ***super***.open();  
}

②Join

相连操作 保存连接的两个关系的迭代器，以及连接谓语 ，还有两者合并后的元组描述符，用来返回连接后新的元组。与filter类中相似，同样使用元组集合保存符合谓语的所有新元组，并保持其迭代器。

*//相连俩关系的迭代器****private*** OpIterator **child1**;  
***private*** OpIterator **child2**;  
*//连接谓语****private*** JoinPredicate **joinPredicate**;  
*//元组描述符****private*** TupleDesc **tupleDesc**;  
*//元组集合迭代器****private*** Iterator<Tuple> **iterator**;  
*//元组集合****private*** List<Tuple> **tuples**;

核心函数：

***public void*** open() ***throws*** DbException, NoSuchElementException,  
 TransactionAbortedException {  
 *// some code goes here* **child1**.open();  
 **child2**.open();  
 Tuple tuple=***null***;  
 *//使用内置循环找出符合条件的两个元组，将两个元组合并如元组集合* ***while***(**child1**.hasNext()){  
 Tuple next1 = **child1**.next();  
 ***while***(**child2**.hasNext()){  
 Tuple next2 = **child2**.next();  
 System.***out***.println(next2);  
 ***if***(**joinPredicate**.filter(next1,next2)){  
 tuple = ***new*** Tuple(**tupleDesc**);  
 ***for***(***int*** i=0;i<**tupleDesc**.numFields();i++){  
 ***if***(i<next1.getTupleDesc().numFields()){  
 tuple.setField(i, next1.getField(i));  
 }***else***{  
 tuple.setField(i, next2.getField(i-next1.getTupleDesc().numFields()));  
 }  
 }  
 **tuples**.add(tuple);  
 }  
 }  
 *//这段代码十分重要 如果不加入使child2重置 会导致无法对两个关系的所有元组进行join比较并连接  
 //因为child2已经到达了最后一个元素 所以不论外围循环还有多少次内层循环都直接终止！！！* **child2**.rewind();  
 System.***out***.println(**child2**);  
 }  
 **iterator**=**tuples**.iterator();  
 ***super***.open();  
}

感悟：

主要的难点在于理解OrderBy和Project的思路，搞懂了之后就比较顺畅。

写join时遇到了一个小问题，在下述代码中的注释里描述了问题。

*//使用内置循环找出符合条件的两个元组，将两个元组合并如元组集合****while***(**child1**.hasNext()){  
 Tuple next1 = **child1**.next();  
 ***while***(**child2**.hasNext()){

......  
 }  
 **tuples**.add(tuple);  
 }  
 }  
 *//这段代码十分重要 如果不加入使child2重置 会导致无法对两个关系的所有元组进行join比较并连接  
 //因为child2已经到达了最后一个元素 所以不论外围循环还有多少次内层循环都直接终止！！！* **child2**.rewind();  
 System.***out***.println(**child2**);

2022年3月8日16时56分31秒

**Exercise 2**

##### **2.2. Aggregates**

An additional SimpleDB operator implements basic SQL aggregates with a GROUP BY clause. You should implement the five SQL aggregates (COUNT, SUM, AVG, MIN, MAX) and support grouping. You only need to support aggregates over a single field, and grouping by a single field.

In order to calculate aggregates, we use an Aggregator interface which merges a new tuple into the existing calculation of an aggregate. The Aggregator is told during construction what operation it should use for aggregation. Subsequently, the client code should call Aggregator.mergeTupleIntoGroup() for every tuple in the child iterator. After all tuples have been merged, the client can retrieve a OpIterator of aggregation results. Each tuple in the result is a pair of the form (groupValue, aggregateValue), unless the value of the group by field was Aggregator.NO\_GROUPING, in which case the result is a single tuple of the form (aggregateValue).

Note that this implementation requires space linear in the number of distinct groups. For the purposes of this lab, you do not need to worry about the situation where the number of groups exceeds available memory.

Implement the skeleton methods in:

* src/java/simpledb/execution/IntegerAggregator.java
* src/java/simpledb/execution/StringAggregator.java
* src/java/simpledb/execution/Aggregate.java

At this point, your code should pass the unit tests IntegerAggregatorTest, StringAggregatorTest, and AggregateTest. Furthermore, you should be able to pass the AggregateTest system test.

任务：

完善上述类代码。

理解：

IntgerAggregator和StringAggregator类是聚合器，用于完成对Int和String类型的五种聚合操作（String类型只支持count），主要的难点在于IntgerAggregator类的合并操作。

IntegerAggregator:

*//按该索引所在的域进行分组****private int* gbField**;  
*//索引所在域的分组类型****private*** Type **gbfieldType**;  
*//聚集域的索引****private int* aField**;  
*//聚集操作****private*** Aggregator.Op **op**;  
*//聚合函数是否有分组****private boolean* isGroup** ;  
*//记录以分组域为键，聚合值为value的map结构****private*** HashMap<Field,Integer> **aggregate**;  
*//记录以分组域为键，总数为value的map结构----用于计算avg****private*** HashMap<Field,Integer> **count**;  
***private*** String **groupfieldName**=**""**;  
***private*** String **fieldName**=**""**;

该聚合器使用两个HashMap类的接管分别保存以聚合域作为key以及其要求的聚合函数的值为value的映射关系，而count映射用于求解avg（平均值）的数值，因为没有办法在合并过程中计算出avg。所以通过count计算出分组域中的相同域的出现次数，最后再根据op是否为avg来修改aggregate中value的值。其他的聚合函数只需要一个aggregate就足够了，具体代码如下：

***public void*** mergeTupleIntoGroup(Tuple tup) {  
 *// some code goes here  
 //首先根据当前的tup以及isGroup来创建key与value；* Field key;  
 Integer value;  
 *//分别记录当前情况的聚合值以及总数* ***int*** currentAggreateValue;  
 ***int*** currentCount;  
 **fieldName** = tup.getTupleDesc().getFieldName(**aField**);  
 *//创建key* ***if***(**isGroup**){  
 key = tup.getField(**gbField**);  
 **groupfieldName** = tup.getTupleDesc().getFieldName(**gbField**);  
 }***else***{  
 key = ***new*** IntField(Aggregator.***NO\_GROUPING***);  
  
 }  
  
 *//强转获取value值* value = ((IntField)tup.getField(**aField**)).getValue();  
  
 *//根据对key是否存在于aggreate HashMap结构中判断是进行初始化  
 //若不存在* ***if***(!**aggregate**.containsKey(key)){  
 *//若是Max和Min则需要将该键值的初始value设置为最小或最大值* ***if***(**op**==Op.***MAX***){  
 **aggregate**.put(key, Integer.***MIN\_VALUE***);  
 }  
  
 ***if***(**op**==Op.***MIN***){  
 **aggregate**.put(key, Integer.***MAX\_VALUE***);  
 }  
  
 *//剩下三种操作只需要将key对应的value初始化为0* ***if***(**op**==Op.***SUM***||**op**==Op.***AVG***||**op**==Op.***COUNT***){  
 **aggregate**.put(key, 0);  
 }  
  
 *//因为该键key是第一次出现说明其出现次数为0 在count中也为0 所以要对在count中的数量进行初始化* **count**.put(key, 0);  
 }  
  
 *//无论是否存在 上述都已经初始化完成 应该保存当前值* currentAggreateValue = **aggregate**.get(key);  
 currentCount = **count**.get(key);  
  
 *//开始根据操作符进行操作* ***if***(**op**==Op.***MAX***){  
 ***if***(currentAggreateValue<value){  
 *//小于则更新为新合并的元组聚合域* **aggregate**.put(key, value);  
 }  
 }  
 ***if***(**op**==Op.***MIN***){  
 ***if***(currentAggreateValue>value){  
 *//大于则更新为新合并的元组聚合域* **aggregate**.put(key, value);  
 }  
 }  
  
 ***if***(**op**==Op.***SUM***){  
 **aggregate**.put(key, currentAggreateValue+value);  
 }  
  
 ***if***(**op**==Op.***COUNT***){  
 *//先自增* **aggregate**.put(key,++currentAggreateValue);  
 }  
  
 ***if***(**op**==Op.***AVG***){  
  
 **aggregate**.put(key,currentAggreateValue+value);  
 *//先自增* **count**.put(key,++currentCount);  
 }  
  
  
}

最后为了返回该聚合操作下的结果，需要根据op以及保存的map结构重新构造一个集合List<Tuple>，返回该集合的迭代器就好了。

***public*** OpIterator iterator() {  
 *// some code goes here* ArrayList<Tuple> tuples = ***new*** ArrayList<>();  
 TupleDesc td =getTupleDesc();  
 *//分组与否来进行设置* ***if***(**isGroup**){  
 *//分组情况下 遍历键集，若求解的是平均值则对value值重新处理* ***for***(Field key : **aggregate**.keySet()){  
 Integer value = **aggregate**.get(key);  
 ***if***(***this***.**op**== Op.***AVG***){  
 value /= **count**.get(key);  
 }  
 Tuple tuple = ***new*** Tuple(td);  
 tuple.setField(0, key);  
 tuple.setField(1, ***new*** IntField(value));  
 tuples.add(tuple);  
 }  
 }***else***{  
 *//未分组情况下，aggreate以及count结构包含所有元组的聚合元素结果* Integer value = **aggregate**.get(***new*** IntField(Aggregator.***NO\_GROUPING***));  
 ***if***(**op**==Op.***AVG***){  
 value /= **count**.get(***new*** IntField(Aggregator.***NO\_GROUPING***));  
 }  
 Tuple tuple = ***new*** Tuple(td);  
 tuple.setField(0, ***new*** IntField(value));  
 tuples.add(tuple);  
 }  
  
 ***return new*** TupleIterator(td, tuples);  
}

StringAggregator的编写只需要在IntgerAggregator类的基础进行删减就好了。

Aggregate类是聚合操作，与Join和Filter相似，基本上完成IntgerAggregator与StringAggregator类的编写后在按照先前的Join和Filter的编写思路即可。具体见代码。

感悟：

主要花费的时间在IntgerAggregator类的编写上，我一开始打算建立一个tuple的集合，用来保存所有分组后的聚合值。但是卡了半天还是无从下手，我不知道是先对分组与否的情况进行区分。同时，在对聚合值计算的步骤上，我原本打算挨个扫描集合类的元组进行判断，但这样的效率很低，需要不断遍历集合，并且对不存在于集合中的元组还需要初始化。于是我找到网上的代码，理解并借鉴了一波，就像上述的理解中一样，只需要使用映射结构就可以有效减少扫描所带来的时间开销，同时映射的关系很好的解决了尚未出现在map中的域的情况，只需要简单的添加一两个if条件判断即可，大大简化了代码。

2022年3月8日16时56分14秒

****Exercise 3.****

**2.3. HeapFile Mutability**

Now, we will begin to implement methods to support modifying tables. We begin at the level of individual pages and files. There are two main sets of operations: adding tuples and removing tuples.

****Removing tuples:**** To remove a tuple, you will need to implement deleteTuple. Tuples contain RecordIDs which allow you to find the page they reside on, so this should be as simple as locating the page a tuple belongs to and modifying the headers of the page appropriately.

****Adding tuples:**** The insertTuple method in HeapFile.java is responsible for adding a tuple to a heap file. To add a new tuple to a HeapFile, you will have to find a page with an empty slot. If no such pages exist in the HeapFile, you need to create a new page and append it to the physical file on disk. You will need to ensure that the RecordID in the tuple is updated correctly.

Implement the remaining skeleton methods in:

* src/java/simpledb/storage/HeapPage.java
* src/java/simpledb/storage/HeapFile.java  
  (Note that you do not necessarily need to implement writePage at this point).

To implement HeapPage, you will need to modify the header bitmap for methods such as insertTuple() and deleteTuple(). You may find that the getNumEmptySlots() and isSlotUsed() methods we asked you to implement in Lab 1 serve as useful abstractions. Note that there is a markSlotUsed method provided as an abstraction to modify the filled or cleared status of a tuple in the page header.

Note that it is important that the HeapFile.insertTuple() and HeapFile.deleteTuple() methods access pages using the BufferPool.getPage() method; otherwise, your implementation of transactions in the next lab will not work properly.

Implement the following skeleton methods in src/simpledb/BufferPool.java:

* insertTuple()
* deleteTuple()

These methods should call the appropriate methods in the HeapFile that belong to the table being modified (this extra level of indirection is needed to support other types of files — like indices — in the future).

At this point, your code should pass the unit tests in HeapPageWriteTest and HeapFileWriteTest, as well as BufferPoolWriteTest.

任务：

完善代码。  
理解：

HeapPage,HeapFile,BufferPool三个类中的InsertTuple和deleteTuple层层调用。

HeapPage

对新插入的元组进行判断，若符合各项规定就找到页面中的第一个空闲slot，将其加入。删除元组也类似，只要元组在表中，就找到元组进行删除。上述两个操作完成后都需要对header中的位信息进行修改，保证位图的正确性。

HeapFile

插入操作相对复杂一些，因为当前文件的页可能出现所有slot已经满的情况，所以需要分类讨论。如果文件中的所有页中某一页还有剩余位置，则调用该页的insert函数即可。

若该表中所有的页都已经满了，那么就要创建一个页，将其写入磁盘，并提供BufferPool读出（将此页加载到缓冲区中）,接下来就与上述一样，调用该页的insert函数就好了。

删除操作遍历所有页找到要删除元组所在的页，调用该页的删除函数即可。

BufferPool

插入，通过数据库找到目标文件，在调用文件的插入就好了

删除一样。

感悟：

只有BufferPoolWriteTest中的handleManyDirtyPages函数无法通过，还不知道为啥。。。

其他的HeapPage负责找到目标行，然后对其做删除或者插入。而HeapFile则通过BufferPool拿到Page，再用page的Insert和Delete方法插入删除，是一个层层调用的过程，

类的设计合理有序。

2022年3月10日15时29分33秒

BufferPoolWriteTest中的handleManyDirtyPages函数可以通过了，根本问题在于该测试类内的HeapFileDuplicate类进行插入元组操作时，是先创建一个页面，然后再在页面中加入元组，也就是跳过了HeapFile类的插入元组操作，直接调用页表的插入。而先前一直出错的原因是，该测试函数是直接向文件内添加数据，没有进过HeapFile等类进行操作。在创建页面时，需要创建该页面的RecordId并更新页面的pgNo，而先前的代码忽视了直接向文件添加数据这个情况，导致pgNo一直不变，除非是HeapFile的调用才会使其变化，所以这里将HeapFile内部表的numPages函数进行修改，使页面pgNo随着文件大小而变化。

2022年3月11日11时6分42秒

***public int*** numPages() {  
 *// some code goes here* **pageNum** = (***int***)**file**.length()/BufferPool.*getPageSize*();  
 ***return* pageNum**;  
}

****Exercise 4.****

**2.4. Insertion and deletion**

Now that you have written all of the HeapFile machinery to add and remove tuples, you will implement the Insert and Delete operators.

For plans that implement insert and delete queries, the top-most operator is a special Insert or Delete operator that modifies the pages on disk. These operators return the number of affected tuples. This is implemented by returning a single tuple with one integer field, containing the count.

Insert: This operator adds the tuples it reads from its child operator to the tableid specified in its constructor. It should use the BufferPool.insertTuple() method to do this.

Delete: This operator deletes the tuples it reads from its child operator from the tableid specified in its constructor. It should use the BufferPool.deleteTuple() method to do this.

Implement the skeleton methods in:

* src/java/simpledb/execution/Insert.java
* src/java/simpledb/execution/Delete.java

At this point, your code should pass the unit tests in InsertTest. We have not provided unit tests for Delete. Furthermore, you should be able to pass the InsertTest and DeleteTest system tests.

理解：

插入操作，从子运算符的元组中读取有效元组，插入到构造器中的传入的tableId表内

*//要插入的元组的迭代器****private*** OpIterator **child**;  
*//被插入的表的id****private int* tableId**;  
*//插入操作的事务****private*** TransactionId **tid**;  
*//插入的元组描述符****private*** TupleDesc **tupleDesc**;  
*//插入的元组数量****private int* numOfInsert**;  
*//是否已经访问过****private boolean* accessed**;

删除操作无须传入要删除的元组的tableId，传入的要删除元组迭代器中的元组包含了了这些信息

*//要删除的元组的迭代器****private*** OpIterator **child**;  
*//要删除的元组的描述符****private*** TupleDesc **tupleDesc**;  
*//进行删除的事务****private*** TransactionId **transactionId**;  
*//是否已经访问过****private boolean* accessed**;  
*//删除的总数****private int* numOfdelete**;

感悟：

相对比较容易，按照步骤写就好了。

出现了一个小插曲，在测试InsertTest时出现了栈溢出的错误，调试后发现是在Insert的open函数中重复调用了open函数导致栈溢出。

2022年3月11日星期五13时45分23秒

****Exercise 5.****

**2.5. Page eviction**

In Lab 1, we did not correctly observe the limit on the maximum number of pages in the buffer pool defined by the constructor argument numPages. Now, you will choose a page eviction policy and instrument any previous code that reads or creates pages to implement your policy.

When more than numPages pages are in the buffer pool, one page should be evicted from the pool before the next is loaded. The choice of eviction policy is up to you; it is not necessary to do something sophisticated. Describe your policy in the lab writeup.

Notice that BufferPool asks you to implement a flushAllPages() method. This is not something you would ever need in a real implementation of a buffer pool. However, we need this method for testing purposes. You should never call this method from any real code.

Because of the way we have implemented ScanTest.cacheTest, you will need to ensure that your flushPage and flushAllPages methods do no evict pages from the buffer pool to properly pass this test.

flushAllPages should call flushPage on all pages in the BufferPool, and flushPage should write any dirty page to disk and mark it as not dirty, while leaving it in the BufferPool.

The only method which should remove page from the buffer pool is evictPage, which should call flushPage on any dirty page it evicts.

Fill in the flushPage() method and additional helper methods to implement page eviction in:

* src/java/simpledb/storage/BufferPool.java

If you did not implement writePage() in HeapFile.java above, you will also need to do that here. Finally, you should also implement discardPage() to remove a page from the buffer pool without flushing it to disk. We will not test discardPage() in this lab, but it will be necessary for future labs.

At this point, your code should pass the EvictionTest system test.

Since we will not be checking for any particular eviction policy, this test works by creating a BufferPool with 16 pages (NOTE: while DEFAULT\_PAGES is 50, we are initializing the BufferPool with less!), scanning a file with many more than 16 pages, and seeing if the memory usage of the JVM increases by more than 5 MB. If you do not implement an eviction policy correctly, you will not evict enough pages, and will go over the size limitation, thus failing the test.

You have now completed this lab. Good work!

任务：

完善BufferPool中的evictPage,flushPage,flushAllPage等函数

理解：

驱逐函数，当缓冲池BufferPool中的页面数量达到限定页数时，需要对它内部的页面进行驱逐。

flushPage,flushAllPage函数直接调用write函数即可。

evictPage函数采用的是FIFO策略，直接将第一个页驱逐，同时要注意，若驱逐的页面是脏页需要写入磁盘。

***private synchronized void*** evictPage() ***throws*** DbException, IOException {  
 *// some code goes here  
 // not necessary for lab1* ***if***(*pages*.size()<1) ***throw new*** DbException(**"BufferPool中不存在页面，无法进行驱逐"**);  
  
 ***for***(PageId pageId : *pages*.keySet()){  
 HeapPage page = (HeapPage)*pages*.get(pageId);  
 *//页面存在* ***if***(page!=***null***){  
 *//页面位脏页 写写入磁盘* ***if***(page.isDirty()!=***null***) flushPage(page.getId());  
 *//驱逐  
 pages*.remove(pageId);  
 *//退出* ***return*** ;  
 }  
 }  
}

感悟：

相对较容易，直接写就好，关键的地方是驱逐策略。FIFO相对好实现，后面可以尝试使用LRU策略进行驱逐，但具体还要好好参考一下。

2022年3月11日星期五14时50分40秒

总结：

本次实验主要是完成各种数据库操作，投影，连接，删除，插入等等。本质上还是对不同迭代器进行处理，对各个情况下的迭代。

今天尝试了lab1，lab2的查询测试，因为文件的格式必须是dat类型，所以需要在终端上对txt类型文件进行转换。同时，对txt文件的编写，最后一行数据需要进行换行，否则会导致内部读取不到最后一行的数据（我找了半天才发现这个一直缺少数据的原因，主要还是对HeapFile内部的文件格式转换不了解）。这两个小测试，感觉也挺有意思的，可以对解析器的源码构造了解一下。

2022年3月12日12时37分40秒

LAB3

#### **Optimizer outline**

Recall that the main idea of a cost-based optimizer is to:

* Use statistics about tables to estimate "costs" of different query plans. Typically, the cost of a plan is related to the cardinalities of (number of tuples produced by) intermediate joins and selections, as well as the selectivity of filter and join predicates.
* Use these statistics to order joins and selections in an optimal way, and to select the best implementation for join algorithms from amongst several alternatives.

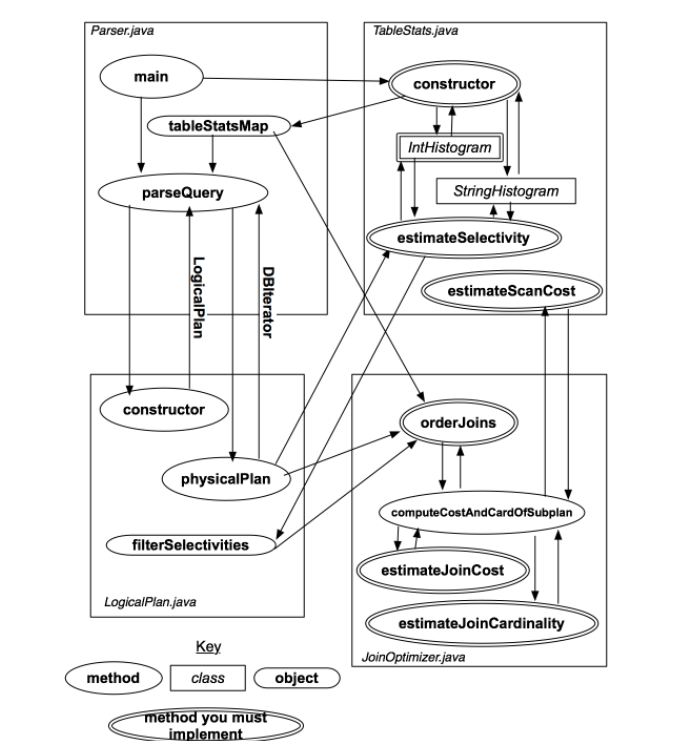
In this lab, you will implement code to perform both of these functions.

The optimizer will be invoked from simpledb/Parser.java. You may wish to review the [lab 2 parser exercise](https://github.com/MIT-DB-Class/simple-db-hw-2021/blob/master/lab2.md" \l "27-query-parser) before starting this lab. Briefly, if you have a catalog file catalog.txt describing your tables, you can run the parser by typing:

java -jar dist/simpledb.jar parser catalog.txt

When the Parser is invoked, it will compute statistics over all of the tables (using statistics code you provide). When a query is issued, the parser will convert the query into a logical plan representation and then call your query optimizer to generate an optimal plan.

**2.1 Overall Optimizer Structure**

Before getting started with the implementation, you need to understand the overall structure of the SimpleDB optimizer. The overall control flow of the SimpleDB modules of the parser and optimize is shown in Figure 1.

*Figure 1: Diagram illustrating classes, methods, and objects used in the parser*

The key at the bottom explains the symbols; you will implement the components with double-borders. The classes and methods will be explained in more detail in the text that follows (you may wish to refer back to this diagram), but the basic operation is as follows:

1. Parser.java constructs a set of table statistics (stored in the statsMap container) when it is initialized. It then waits for a query to be input, and calls the method parseQuery on that query.
2. parseQuery first constructs a LogicalPlan that represents the parsed query. parseQuery then calls the method physicalPlan on the LogicalPlan instance it has constructed. The physicalPlan method returns a DBIterator object that can be used to actually run the query.

In the exercises to come, you will implement the methods that help physicalPlan devise an optimal plan.

**2.2. Statistics Estimation**

Accurately estimating plan cost is quite tricky. In this lab, we will focus only on the cost of sequences of joins and base table accesses. We won't worry about access method selection (since we only have one access method, table scans) or the costs of additional operators (like aggregates).

You are only required to consider left-deep plans for this lab. See Section 2.3 for a description of additional "bonus" optimizer features you might implement, including an approach for handling bushy plans.

**2.2.1 Overall Plan Cost**

We will write join plans of the form p=t1 join t2 join ... tn, which signifies a left deep join where t1 is the left-most join (deepest in the tree). Given a plan like p, its cost can be expressed as:

scancost(t1) + scancost(t2) + joincost(t1 join t2) +

scancost(t3) + joincost((t1 join t2) join t3) +

...

Here, scancost(t1) is the I/O cost of scanning table t1, joincost(t1,t2) is the CPU cost to join t1 to t2. To make I/O and CPU cost comparable, typically a constant scaling factor is used, e.g.:

cost(predicate application) = 1

cost(pageScan) = SCALING\_FACTOR x cost(predicate application)

For this lab, you can ignore the effects of caching (e.g., assume that every access to a table incurs the full cost of a scan) -- again, this is something you may add as an optional bonus extension to your lab in Section 2.3. Therefore, scancost(t1) is simply the number of pages in t1 x SCALING\_FACTOR.

**2.2.2 Join Cost**

When using nested loops joins, recall that the cost of a join between two tables t1 and t2 (where t1 is the outer) is simply:

joincost(t1 join t2) = scancost(t1) + ntups(t1) x scancost(t2) //IO cost

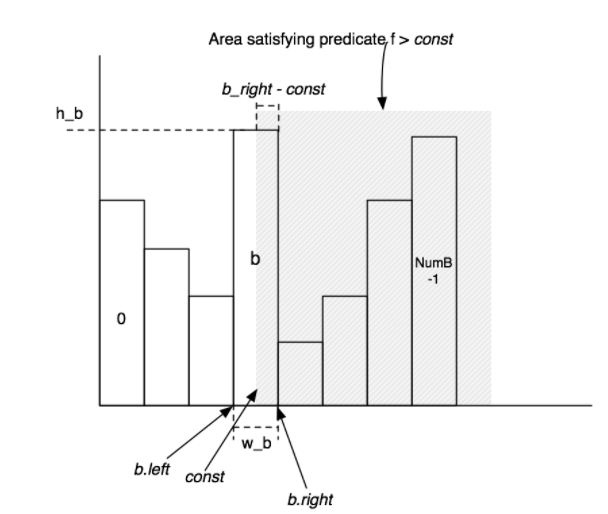
+ ntups(t1) x ntups(t2) //CPU cost

Here, ntups(t1) is the number of tuples in table t1.

**2.2.3 Filter Selectivity**

ntups can be directly computed for a base table by scanning that table. Estimating ntups for a table with one or more selection predicates over it can be trickier -- this is the filter selectivity estimation problem. Here's one approach that you might use, based on computing a histogram over the values in the table:

* Compute the minimum and maximum values for every attribute in the table (by scanning it once).
* Construct a histogram for every attribute in the table. A simple approach is to use a fixed number of buckets NumB, with each bucket representing the number of records in a fixed range of the domain of the attribute of the histogram. For example, if a field f ranges from 1 to 100, and there are 10 buckets, then bucket 1 might contain the count of the number of records between 1 and 10, bucket 2 a count of the number of records between 11 and 20, and so on.
* Scan the table again, selecting out all of fields of all of the tuples and using them to populate the counts of the buckets in each histogram.
* To estimate the selectivity of an equality expression, f=const, compute the bucket that contains value const. Suppose the width (range of values) of the bucket is w, the height (number of tuples) is h, and the number of tuples in the table is ntups. Then, assuming values are uniformly distributed throughout the bucket, the selectivity of the expression is roughly (h / w) / ntups, since (h/w) represents the expected number of tuples in the bin with value const.
* To estimate the selectivity of a range expression f>const, compute the bucket b that const is in, with width w\_b and height h\_b. Then, b contains a fraction b\_f = h\_b / ntups of the total tuples. Assuming tuples are uniformly distributed throughout b, the fraction b\_part of b that is > const is (b\_right - const) / w\_b, where b\_right is the right endpoint of b's bucket. Thus, bucket b contributes (b\_f x b\_part) selectivity to the predicate. In addition, buckets b+1...NumB-1 contribute all of their selectivity (which can be computed using a formula similar to b\_f above). Summing the selectivity contributions of all the buckets will yield the overall selectivity of the expression. Figure 2 illustrates this process.
* Selectivity of expressions involving less than can be performed similar to the greater than case, loking at buckets down to 0.

  
*Figure 2: Diagram illustrating the histograms you will implement in Lab 5*

In the next two exercises, you will code to perform selectivity estimation of joins and filters.

****Exercise 1: IntHistogram.java****

You will need to implement some way to record table statistics for selectivity estimation. We have provided a skeleton class, IntHistogram that will do this. Our intent is that you calculate histograms using the bucket-based method described above, but you are free to use some other method so long as it provides reasonable selectivity estimates.

We have provided a class StringHistogram that uses IntHistogram to compute selecitivites for String predicates. You may modify StringHistogram if you want to implement a better estimator, though you should not need to in order to complete this lab.

After completing this exercise, you should be able to pass the IntHistogramTest unit test (you are not required to pass this test if you choose not to implement histogram-based selectivity estimation).

任务：

完成intHistogram的代码补充，理解stringHistogram的实现。

理解：

IntHistogram是int类型的值的直方统计图，通过讲义中讲解的公式就可以推导出它的选择性。

*//桶的数量****private int* buckets**;  
*//最小值****private int* min**;  
*//最大值****private int* max**;  
*//范围****private int* width**;  
*//桶****private int***[] **intHistogram**;  
*//直方图的记录条数****private int* ntups**;

主要的问题是如何表达直方图，这里我直接使用数组来代表桶，每当一个范围内的数加入，就在该桶内加1表示添加，在进行比较的操作时，默认直方图中的各项数据是均匀分布的。

之后对大于，等于的情况进行代码编写，套用公式就好了，随后的小于，小于等于，大于等于直接调用大于，等于的情况就好了。

StringHistogtam是string类型统计图，基本是只是调用了int类的直方统计图，源码也比较齐全，不需要补充和测试。

感悟：

IntHistogram的实现相对比较简单，只是单纯的采用数组就好。但是由于数据的各种特殊情况，例如桶数量远远大于表示范围---导致精度较高、传入不在指定范围内的数值等等，这些情况容易出现精度的丢失问题，以及数组越界。所以在编写代码的过程中需要小心处理设计数值计算的过程，基本上在出现公式计算时，都要对公式的数据进行类型转换，以防数值出错。同时，对于传入值的范围问题，只需要对不同情况进行合理分类就可以解决。

还有边界值处理的问题，例如计算某个数值属于那个桶，那个数值所在桶的右边界是多少，这些问题都是细小易出错的地方，需要格外小心。

2022年3月13日12时9分41秒

***int*** bucketPos = valueToIndex(v); *//桶所在的位置* ***int*** right = bucketPos\***width**+**min**+**width**-1; *//桶所在位置的右端点  
 //下述代码应该放在各种情况下的里面 不然会有超出界限的异常  
// int height = intHistogram[bucketPos]; //该桶总数* ***double*** selectivity = 0.0;  
 ***int*** height =0;  
  
 *// some code goes here* ***switch*** (op){  
  
  
 ***case EQUALS***:  
  
 ***if***(v<**min**||v>**max**) ***return*** 0.0;  
 height = **intHistogram**[bucketPos]; *//该桶总数* selectivity = (1.0/**width**)\*(1.0\*height/**ntups**);  
 ***break***;  
  
 ***case GREATER\_THAN***:  
  
 ***if***(v>**max**){  
 ***return*** 0.0;  
 }***else if***(v<**min**){  
 ***return*** 1.0;  
 }  
 height = **intHistogram**[bucketPos]; *//该桶总数* selectivity =((right-v)/**width**)\*(height/**ntups**);  
 *//还有大于右边的所有值* ***double*** sum =0.0;  
 ***for***(***int*** i=bucketPos+1;i<**buckets**;i++){  
 sum+=**intHistogram**[i];  
 }  
 selectivity+=(sum/**ntups**);  
 ***break***;  
  
 ***case LESS\_THAN***:  
 selectivity = 1.0-estimateSelectivity(Predicate.Op.***GREATER\_THAN***, v);  
 ***break***;  
  
 ***case LESS\_THAN\_OR\_EQ***:  
  
 selectivity = estimateSelectivity(Predicate.Op.***LESS\_THAN***,v)+estimateSelectivity(Predicate.Op.***EQUALS***, v);  
 ***break***;  
  
 ***case GREATER\_THAN\_OR\_EQ***:  
  
 selectivity = estimateSelectivity(Predicate.Op.***GREATER\_THAN***,v)+estimateSelectivity(Predicate.Op.***EQUALS***, v);  
 ***break***;  
  
 ***case LIKE***:  
  
 selectivity = avgSelectivity();  
 ***break***;  
  
 ***case NOT\_EQUALS***:  
  
 selectivity = 1-estimateSelectivity(Predicate.Op.***EQUALS***, v);  
 ***break***;  
  
 }  
 ***return*** selectivity;

****Exercise 2: TableStats.java****

The class TableStats contains methods that compute the number of tuples and pages in a table and that estimate the selectivity of predicates over the fields of that table. The query parser we have created creates one instance of TableStats per table, and passes these structures into your query optimizer (which you will need in later exercises).

You should fill in the following methods and classes in TableStats:

* Implement the TableStats constructor: Once you have implemented a method for tracking statistics such as histograms, you should implement the TableStats constructor, adding code to scan the table (possibly multiple times) to build the statistics you need.
* Implement estimateSelectivity(int field, Predicate.Op op, Field constant): Using your statistics (e.g., an IntHistogram or StringHistogram depending on the type of the field), estimate the selectivity of predicate field op constant on the table.
* Implement estimateScanCost(): This method estimates the cost of sequentially scanning the file, given that the cost to read a page is costPerPageIO. You can assume that there are no seeks and that no pages are in the buffer pool. This method may use costs or sizes you computed in the constructor.
* Implement estimateTableCardinality(double selectivityFactor): This method returns the number of tuples in the relation, given that a predicate with selectivity selectivityFactor is applied. This method may use costs or sizes you computed in the constructor.

You may wish to modify the constructor of TableStats.java to, for example, compute histograms over the fields as described above for purposes of selectivity estimation.

After completing these tasks you should be able to pass the unit tests in TableStatsTest.

任务：

完成Tablestats中的代码。

理解：

Tablestats保存了所有表的状态信息，包含了每张表的每个域的直方统计图。

下面的变量是所有表的状态信息集合

***private static final*** ConcurrentMap<String, TableStats> ***statsMap*** = ***new*** ConcurrentHashMap<>();

下述是其中一张表的状态信息

*//表示的表的id****private int* tableId**;  
*//表示的表的页数****private int* pageNum**;  
*//读取每页的io消耗****private int* ioCostPage**;  
*//总元组数****private int* totalTup**;  
*//每个域的直方图****private*** HashMap<Integer,Object> **fieldToHist**;  
*//每个域的最大值最小值****private*** HashMap<Integer,Integer> **max**;  
***private*** HashMap<Integer,Integer> **min**;  
***private final*** TupleDesc **tupleDesc**;

重点在于对表的不同域的统计，只要分好类别，按照int和string两者类型创建对于的直方图并添加数据就好。当需要某个域在某种op下的选择型，直接调用该类型的直方图中的estimateSelectivity函数并返回就ok了。核心代码较长，直接看源码就好，这里不贴出。

感悟：

test测试时，还是出现了一些比较特殊的边界问题，通过对这些边界数据的调试，发现直方图中的选择性评估函数estimateSelectivity还是有一点小漏洞，经过修改后，所有测试案例都可以通过。

2022年3月13日16时55分

****Exercise 3:****

**2.2.4 Join Cardinality**

Finally, observe that the cost for the join plan p above includes expressions of the form joincost((t1 join t2) join t3). To evaluate this expression, you need some way to estimate the size (ntups) of t1 join t2. This join cardinality estimation problem is harder than the filter selectivity estimation problem. In this lab, you aren't required to do anything fancy for this, though one of the optional excercises in Section 2.4 includes a histogram-based method for join selectivity estimation.

While implementing your simple solution, you should keep in mind the following:

* For equality joins, when one of the attributes is a primary key, the number of tuples produced by the join cannot be larger than the cardinality of the non-primary key attribute.
* For equality joins when there is no primary key, it's hard to say much about what the size of the output is -- it could be the size of the product of the cardinalities of the tables (if both tables have the same value for all tuples) -- or it could be 0. It's fine to make up a simple heuristic (say, the size of the larger of the two tables).
* For range scans, it is similarly hard to say anything accurate about sizes. The size of the output should be proportional to the sizes of the inputs. It is fine to assume that a fixed fraction of the cross-product is emitted by range scans (say, 30%). In general, the cost of a range join should be larger than the cost of a non-primary key equality join of two tables of the same size.

****Join Cost Estimation****

The class JoinOptimizer.java includes all of the methods for ordering and computing costs of joins. In this exercise, you will write the methods for estimating the selectivity and cost of a join, specifically:

* Implement estimateJoinCost(LogicalJoinNode j, int card1, int card2, double cost1, double cost2): This method estimates the cost of join j, given that the left input is of cardinality card1, the right input of cardinality card2, that the cost to scan the left input is cost1, and that the cost to access the right input is card2. You can assume the join is an NL join, and apply the formula mentioned earlier.
* Implement estimateJoinCardinality(LogicalJoinNode j, int card1, int card2, boolean t1pkey, boolean t2pkey): This method estimates the number of tuples output by join j, given that the left input is size card1, the right input is size card2, and the flags t1pkey and t2pkey that indicate whether the left and right (respectively) field is unique (a primary key).

After implementing these methods, you should be able to pass the unit tests estimateJoinCostTest and estimateJoinCardinality in JoinOptimizerTest.java.

任务：完成JoinOptimizer中的estimateJoinCardinality以及estimateJoinCost两函数。

理解：

estimateJoinCost：估计两表相连的开销，因为，采取的是简单的内置循环相连，所以时间开销为：

*//时间开销：Scancost(T1)+nTups(T1)\*Scancost(T2)+nTups(T1)\*nTups(T2)  
//扫描一次外表需要的时间+外表的条目数\*内表的一次扫描开销+所有条目进行连接操作的cpu开销****return*** card1\*cost2+cost1+card1\*card2;

estimateJoinCardinality：在不同连接条件下，以及是否包含主键情况下两表的连接基数，根据讲义中的讲解可以简单推出

*基于上述的代码讲义，可以容易的退出计算相连操作表的基数  
\* 按照要点1，对于等价查询，如果某表包含主键，则join操作的基数不大于不含主键属性的表的基数。  
\* 按照要点2，对于等价查询，如果都不包含主键，则join操作的基数最大可能为两表基数的乘积，最小为0，但这里我们将其设置为两表中的较大值  
\* 按照要点3，对于范围查找，则对于两者的查找基数，我们取一个大概值30%，基数\*30%就是我们求的数*

按照上述要求编写代码即可。

感悟：这个练习的代码量比较小，讲义中也明确给了提示，只要理解了就相对容易写。

2022年3月14日11时50分

****Exercise 4****

**2.3 Join Ordering**

Now that you have implemented methods for estimating costs, you will implement the Selinger optimizer. For these methods, joins are expressed as a list of join nodes (e.g., predicates over two tables) as opposed to a list of relations to join as described in class.

Translating the algorithm given in lecture to the join node list form mentioned above, an outline in pseudocode would be:

1. j = set of join nodes

2. for (i in 1...|j|):

3. for s in {all length i subsets of j}

4. bestPlan = {}

5. for s' in {all length d-1 subsets of s}

6. subplan = optjoin(s')

7. plan = best way to join (s-s') to subplan

8. if (cost(plan) < cost(bestPlan))

9. bestPlan = plan

10. optjoin(s) = bestPlan

11. return optjoin(j)

To help you implement this algorithm, we have provided several classes and methods to assist you. First, the method enumerateSubsets(List v, int size) in JoinOptimizer.java will return a set of all of the subsets of v of size size. This method is VERY inefficient for large sets; you can earn extra credit by implementing a more efficient enumerator (hint: consider using an in-place generation algorithm and a lazy iterator (or stream) interface to avoid materializing the entire power set).

Second, we have provided the method:

private CostCard computeCostAndCardOfSubplan(Map<String, TableStats> stats,

Map<String, Double> filterSelectivities,

LogicalJoinNode joinToRemove,

Set<LogicalJoinNode> joinSet,

double bestCostSoFar,

PlanCache pc)

Given a subset of joins (joinSet), and a join to remove from this set (joinToRemove), this method computes the best way to join joinToRemove to joinSet - {joinToRemove}. It returns this best method in a CostCard object, which includes the cost, cardinality, and best join ordering (as a list). computeCostAndCardOfSubplan may return null, if no plan can be found (because, for example, there is no left-deep join that is possible), or if the cost of all plans is greater than the bestCostSoFar argument. The method uses a cache of previous joins called pc (optjoin in the psuedocode above) to quickly lookup the fastest way to join joinSet - {joinToRemove}. The other arguments (stats and filterSelectivities) are passed into the orderJoins method that you must implement as a part of Exercise 4, and are explained below. This method essentially performs lines 6--8 of the psuedocode described earlier.

Third, we have provided the method:

private void printJoins(List<LogicalJoinNode> js,

PlanCache pc,

Map<String, TableStats> stats,

Map<String, Double> selectivities)

This method can be used to display a graphical representation of a join plan (when the "explain" flag is set via the "-explain" option to the optimizer, for example).

Fourth, we have provided a class PlanCache that can be used to cache the best way to join a subset of the joins considered so far in your implementation of Selinger (an instance of this class is needed to use computeCostAndCardOfSubplan).

****Join Ordering****

In JoinOptimizer.java, implement the method:

List<LogicalJoinNode> orderJoins(Map<String, TableStats> stats,

Map<String, Double> filterSelectivities,

boolean explain)

This method should operate on the joins class member, returning a new List that specifies the order in which joins should be done. Item 0 of this list indicates the left-most, bottom-most join in a left-deep plan. Adjacent joins in the returned list should share at least one field to ensure the plan is left-deep. Here stats is an object that lets you find the TableStats for a given table name that appears in the FROM list of the query. filterSelectivities allows you to find the selectivity of any predicates over a table; it is guaranteed to have one entry per table name in the FROM list. Finally, explain specifies that you should output a representation of the join order for informational purposes.

You may wish to use the helper methods and classes described above to assist in your implementation. Roughly, your implementation should follow the psuedocode above, looping through subset sizes, subsets, and sub-plans of subsets, calling computeCostAndCardOfSubplan and building a PlanCache object that stores the minimal-cost way to perform each subset join.

After implementing this method, you should be able to pass all the unit tests in JoinOptimizerTest. You should also pass the system test QueryTest

任务：根据讲义中给出的伪代码，以及提供的方法与类补充完成伪代码的实现。

理解：

理解讲义中的各个类的含义，以及函数的功能，直接套用就可以了。

CostCard ：包含了某个计划的开销信息。

PlanCache:缓存了所有集合的最优计划，以及他们的最优开销。

computeCostAndCardOfSubplan：根据传入的集合，以及各项信息，计算出最优的计划。

enumerateSubsets：本次连接操作在所有连接节点的全排列集合。

*1. j = set of join nodes  
 2. for (i in 1...|j|):  
 3. for s in {all length i subsets of j}  
 4. bestPlan = {}  
 5. for s' in {all length d-1 subsets of s}  
 6. subplan = optjoin(s')  
 7. plan = best way to join (s-s') to subplan  
 8. if (cost(plan) < cost(bestPlan))  
 9. bestPlan = plan  
 10. optjoin(s) = bestPlan  
 11. return optjoin(j)*

***public*** List<LogicalJoinNode> orderJoins(  
 Map<String, TableStats> stats,  
 Map<String, Double> filterSelectivities, ***boolean*** explain)  
 ***throws*** ParsingException {  
  
 *// some code goes here  
 //创建要返回的集合  
  
 /\*  
 \*/  
 //pc就是伪代码中的optjoin，用来保存某个连接操作集合的最优的计划* PlanCache pc = ***new*** PlanCache();  
 Set<LogicalJoinNode> Wholeset = ***new*** HashSet<LogicalJoinNode>();  
 ***int*** logicalJoinNodeNum = **joins**.size();  
 *//保存包含所有节点的集合，便于返回操作* ***for***(***int*** i=0;i<**joins**.size();i++){  
 Wholeset.add(**joins**.get(i));  
 }  
  
 *//1.2  
 // 对此次所有连接操作的node节点进行全排列，获得所有节点的全部集合* ***for***(***int*** i = 1;i<=logicalJoinNodeNum;i++){  
 *//传入代表所有连接的joins集合，并设置子集合大小为i 获取子集为i个连接操作的所有集合* Set<Set<LogicalJoinNode>> sets = enumerateSubsets(**joins**, i);  
 *//3  
 // 将上述获取的集合分解为多个子集，获取其所有子集中最优的连接计划* ***for***(Set<LogicalJoinNode> set : sets){  
 *//保存包含所有节点的集合，便于返回操作  
// if(set.size()== joins.size()){  
// Wholeset = set;  
// }  
  
 //4  
 // 截止目前的最佳计划* CostCard bestplan =***new*** CostCard();  
 *//迄今为止的最优花费，初始为最大值，保证下面得到更好的计划* ***double*** bestCostSoFar = Double.***MAX\_VALUE***;  
 *//5  
 // 对所有子集，剔除每一个该子集内的节点，joinSet - {joinToRemove} 并通过pc迅速来找到最优子计划* ***for***(LogicalJoinNode joinNodeToRemove : set ){  
 *//6.7.8  
 //此方法计算联接joinToRemove到joinSet-{joinToRemove}的最佳方式* CostCard plan = computeCostAndCardOfSubplan(stats, filterSelectivities,  
 joinNodeToRemove, set, bestCostSoFar, pc);  
 *//9.  
 // 返回不为空，说明存在更优的计划* ***if***(plan!=***null***){  
 *//替换成最佳计划* bestplan = plan ;  
 *//更新新的最佳花费* bestCostSoFar = plan.**cost**;  
 }  
 }  
 *//10  
 // 不为空，则添加最优计划到option中* ***if***(bestplan!=***null***){  
 pc.addPlan(set, bestplan.**cost**, bestplan.**card**, bestplan.**plan**);  
 }  
 }  
  
 }  
  
 *//This method can be used to display a graphical representation of a join plan  
 // (when the "explain" flag is set via the "-explain" option to the optimizer, for example).* ***if***(explain){  
 printJoins(**joins**, pc, stats, filterSelectivities);  
 }  
  
 *//11  
 //通过本次连接操作中所有节点的集合，在计划缓冲器中找到保存的最优计划* ***return*** pc.getOrder(Wholeset);  
 }

感悟:

难点在于对伪代码的理解，虽然讲义中讲解的很明确了，但是一时间多个函数与类的设计还是比较难以掌握。所以还是通过反复仔细阅读讲义以及源码，最后掌握伪代码的含义，以及各个函数，类的作用。

2022年3月14日16时4分43秒

总结：

本次实验着重于对优化器的实现，通过实验1我们补充了两种类型int，string的直方图，通过直方图，我们得以计算不同谓语条件下某个表的特定域的选择性，为实验2统计不同表的状态信息保存铺垫了一波。在实验2中只需要对不同表的不同域设置应的直方图并保存，就实现了对表的状态信息的保存。接下来的实验3实现相连操作下的开销预估以及连接基数的估计，只要按照实验中的讲义所给的计算公式就好了，（这里采取的是简单的内置循环连接操作，所以相对容易点）。最后的实验4是本次实验的重中之重，通过给定的伪代码思想，计算出在一个完整的查询操作下--包含了多个相连子操作的多连接操作下的最优方案的开销。

### **LAB4**

前导材料

**2. Transactions, Locking, and Concurrency Control**

Before starting, you should make sure you understand what a transaction is and how strict two-phase locking (which you will use to ensure isolation and atomicity of your transactions) works.

In the remainder of this section, we briefly overview these concepts and discuss how they relate to SimpleDB.

**2.1. Transactions**

A transaction is a group of database actions (e.g., inserts, deletes, and reads) that are executed atomically; that is, either all of the actions complete or none of them do, and it is not apparent to an outside observer of the database that these actions were not completed as a part of a single, indivisible action.

**2.2. The ACID Properties**

To help you understand how transaction management works in SimpleDB, we briefly review how it ensures that the ACID properties are satisfied:

* ****Atomicity****: Strict two-phase locking and careful buffer management ensure atomicity.
* ****Consistency****: The database is transaction consistent by virtue of atomicity. Other consistency issues (e.g., key constraints) are not addressed in SimpleDB.
* ****Isolation****: Strict two-phase locking provides isolation.
* ****Durability****: A FORCE buffer management policy ensures durability (see Section 2.3 below).

**2.3. Recovery and Buffer Management**

To simplify your job, we recommend that you implement a NO STEAL/FORCE buffer management policy.

As we discussed in class, this means that:

* You shouldn't evict dirty (updated) pages from the buffer pool if they are locked by an uncommitted transaction (this is NO STEAL).
* On transaction commit, you should force dirty pages to disk (e.g., write the pages out) (this is FORCE).

To further simplify your life, you may assume that SimpleDB will not crash while processing a transactionComplete command. Note that these three points mean that you do not need to implement log-based recovery in this lab, since you will never need to undo any work (you never evict dirty pages) and you will never need to redo any work (you force updates on commit and will not crash during commit processing).

**2.4. Granting Locks**

You will need to add calls to SimpleDB (in BufferPool, for example), that allow a caller to request or release a (shared or exclusive) lock on a specific object on behalf of a specific transaction.

We recommend locking at page granularity; please do not implement table-level locking (even though it is possible) for simplicity of testing. The rest of this document and our unit tests assume page-level locking.

You will need to create data structures that keep track of which locks each transaction holds and check to see if a lock should be granted to a transaction when it is requested.

You will need to implement shared and exclusive locks; recall that these work as follows:

* Before a transaction can read an object, it must have a shared lock on it.
* Before a transaction can write an object, it must have an exclusive lock on it.
* Multiple transactions can have a shared lock on an object.
* Only one transaction may have an exclusive lock on an object.
* If transaction t is the only transaction holding a shared lock on an object o, t may upgrade its lock on o to an exclusive lock.

If a transaction requests a lock that cannot be immediately granted, your code should block, waiting for that lock to become available (i.e., be released by another transaction running in a different thread). Be careful about race conditions in your lock implementation --- think about how concurrent invocations to your lock may affect the behavior. (you way wish to read about [Synchronization](http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html) in Java).

****Exercise 1.****

Write the methods that acquire and release locks in BufferPool. Assuming you are using page-level locking, you will need to complete the following:

* Modify getPage() to block and acquire the desired lock before returning a page.
* Implement unsafeReleasePage(). This method is primarily used for testing, and at the end of transactions.
* Implement holdsLock() so that logic in Exercise 2 can determine whether a page is already locked by a transaction.

You may find it helpful to define a LockManager class that is responsible for maintaining state about transactions and locks, but the design decision is up to you.

You may need to implement the next exercise before your code passes the unit tests in LockingTest.

任务：编写一个LockManager类对事务进行页级的锁的管理。

通过LockManager类的内置函数完善BufferPool内的getPage，unsafeReleasePage， holdsLock函数。

理解：难点在LockManager类的实现。

*锁管理器--对页和事务进行管理  
1.对每个页保存持有该页的锁的事务以及锁类型的信息，因为可能有多个事务持有该页所以使用一个list结构保存  
 --这里用一个锁状态来保持LockStat来保存  
2.对每个等待持有某个已经被加上排他锁的页的事务进行保存*

*3.在LockManager内实现了LockStat类用来保存某个事务以及其锁类型*

*//锁表 被加了锁的页表****private*** Map<PageId, List<LockStat>> **lockTable**;  
*//等待锁释放的事务****private*** Map<TransactionId,PageId> **waitTable**;

*//锁状态类****private class*** LockStat{  
 TransactionId **transactionId**;  
 *//区分读锁和排他锁* Permissions **permissions**;}

之后就实现LockManager内的读锁和写锁分别的赋锁函数，根据锁表的信息分别对要求向某个页加锁的特定事务进行分类讨论。若可以赋锁，直接调用lock函数，并传入需要的信息，若不可以赋锁则调用wait函数对申请锁的操作进行阻塞，一段时间后，再次申请锁，反复前述操作，直到申请成功。代码主要参照讲义中的下面几点。

* Before a transaction can read an object, it must have a shared lock on it.
* Before a transaction can write an object, it must have an exclusive lock on it.
* Multiple transactions can have a shared lock on an object.
* Only one transaction may have an exclusive lock on an object.
* If transaction t is the only transaction holding a shared lock on an object o, t may upgrade its lock on o to an exclusive lock.

剩下的就是一些对锁的释放操作，主要是注意细节就好。

最后，LockManager实现好之后。BufferPool内的getPage，unsafeReleasePage， holdsLock函数。三个函数调按照提示调用LockManager内的函数就好了。

感悟：

该练习的实现是参考别人的代码实现的，我读完先导材料后一头雾水，不知道如何实现一个锁管理类，内部应该如何保存信息，如何表示锁。参考了别人的代码后，我自主完成了相应代码逻辑，在测试案例中出现了不少错误，我一步一步调试，发现了在实现锁管理类的内部函数如lock，unlock等存在一些细枝末节的漏洞，少了一些条件的判断，补全后，测试案例全部通过。

****Exercise 2.****

##### **2.5. Lock Lifetime**

You will need to implement strict two-phase locking. This means that transactions should acquire the appropriate type of lock on any object before accessing that object and shouldn't release any locks until after the transaction commits.

Fortunately, the SimpleDB design is such that it is possible to obtain locks on pages in BufferPool.getPage() before you read or modify them. So, rather than adding calls to locking routines in each of your operators, we recommend acquiring locks in getPage(). Depending on your implementation, it is possible that you may not have to acquire a lock anywhere else. It is up to you to verify this!

You will need to acquire a shared lock on any page (or tuple) before you read it, and you will need to acquire an exclusive lock on any page (or tuple) before you write it. You will notice that we are already passing around Permissions objects in the BufferPool; these objects indicate the type of lock that the caller would like to have on the object being accessed (we have given you the code for the Permissions class.)

Note that your implementation of HeapFile.insertTuple() and HeapFile.deleteTuple(), as well as the implementation of the iterator returned by HeapFile.iterator() should access pages using BufferPool.getPage(). Double check that these different uses of getPage() pass the correct permissions object (e.g., Permissions.READ\_WRITE or Permissions.READ\_ONLY). You may also wish to double check that your implementation of BufferPool.insertTuple() and BufferPool.deleteTupe() call markDirty() on any of the pages they access (you should have done this when you implemented this code in lab 2, but we did not test for this case.)

After you have acquired locks, you will need to think about when to release them as well. It is clear that you should release all locks associated with a transaction after it has committed or aborted to ensure strict 2PL. However, it is possible for there to be other scenarios in which releasing a lock before a transaction ends might be useful. For instance, you may release a shared lock on a page after scanning it to find empty slots (as described below).

Ensure that you acquire and release locks throughout SimpleDB. Some (but not necessarily all) actions that you should verify work properly:

* Reading tuples off of pages during a SeqScan (if you implemented locking in BufferPool.getPage(), this should work correctly as long as your HeapFile.iterator() uses BufferPool.getPage().)
* Inserting and deleting tuples through BufferPool and HeapFile methods (if you implemented locking in BufferPool.getPage(), this should work correctly as long as HeapFile.insertTuple() and HeapFile.deleteTuple() use BufferPool.getPage().)

You will also want to think especially hard about acquiring and releasing locks in the following situations:

* Adding a new page to a HeapFile. When do you physically write the page to disk? Are there race conditions with other transactions (on other threads) that might need special attention at the HeapFile level, regardless of page-level locking?
* Looking for an empty slot into which you can insert tuples. Most implementations scan pages looking for an empty slot, and will need a READ\_ONLY lock to do this. Surprisingly, however, if a transaction t finds no free slot on a page p, t may immediately release the lock on p. Although this apparently contradicts the rules of two-phase locking, it is ok because t did not use any data from the page, such that a concurrent transaction t' which updated p cannot possibly effect the answer or outcome of t.

At this point, your code should pass the unit tests in LockingTest.

任务：练习2的任务基本上都在练习1完成

感悟：讲义中的要求--对寻找空闲slot时使用读锁，并在找到空闲slot时迅速释放读锁。这一段我还是不太理解，我感觉这一段看起来确实可行，但是为了继续后续的操作还是需要获取对应页表的写锁才能继续执行写操作，单纯的读锁查再释放显得有点多余，也不好实现。

****Exercise 3.****

#### **2.6. Implementing NO STEAL**

Modifications from a transaction are written to disk only after it commits. This means we can abort a transaction by discarding the dirty pages and rereading them from disk. Thus, we must not evict dirty pages. This policy is called NO STEAL.

You will need to modify the evictPage method in BufferPool. In particular, it must never evict a dirty page. If your eviction policy prefers a dirty page for eviction, you will have to find a way to evict an alternative page. In the case where all pages in the buffer pool are dirty, you should throw a DbException. If your eviction policy evicts a clean page, be mindful of any locks transactions may already hold to the evicted page and handle them appropriately in your implementation.

Implement the necessary logic for page eviction without evicting dirty pages in the evictPage method in BufferPool.

任务：实现非窃取策略

理解：

在驱逐页函数内实现非窃取策略，不应该驱逐任何脏页，页面的重新写入磁盘这一操作应该在事务提交之后。所以驱逐页面时，需要选择驱逐一个clean页面，但该clean页面可能以及被某个事务锁定，需要在lockmanager中解锁。如果全是dirty页面就抛出异常。

****Exercise 4.****

**2.7. Transactions**

In SimpleDB, a TransactionId object is created at the beginning of each query. This object is passed to each of the operators involved in the query. When the query is complete, the BufferPool method transactionComplete is called.

Calling this method either commits or aborts the transaction, specified by the parameter flag commit. At any point during its execution, an operator may throw a TransactionAbortedException exception, which indicates an internal error or deadlock has occurred. The test cases we have provided you with create the appropriate TransactionId objects, pass them to your operators in the appropriate way, and invoke transactionComplete when a query is finished. We have also implemented TransactionId.

Implement the transactionComplete() method in BufferPool. Note that there are two versions of transactionComplete, one which accepts an additional boolean ****commit**** argument, and one which does not. The version without the additional argument should always commit and so can simply be implemented by calling transactionComplete(tid, true).

When you commit, you should flush dirty pages associated to the transaction to disk. When you abort, you should revert any changes made by the transaction by restoring the page to its on-disk state.

Whether the transaction commits or aborts, you should also release any state the BufferPool keeps regarding the transaction, including releasing any locks that the transaction held.

At this point, your code should pass the TransactionTest unit test and the AbortEvictionTest system test. You may find the TransactionTest system test illustrative, but it will likely fail until you complete the next exercise.

任务：

实现BufferPool中的transactionComplete函数。

理解：

在事务提交时，若事务正常完成，则将该事务写过的页面重新写入磁盘，若事务被中断就把事务修改过的页面重新读入缓冲池BufferPool内，变相实现事务回滚操作。但，无论事务成功与否都需要释放当前事务的所有锁。

*true--commit  
 When you commit, you should flush dirty pages associated to the transaction to disk.*

*false--abort  
 When you abort, you should revert any changes made by the transaction  
 by restoring the page to its on-disk state.  
 无论提交还是终止都需要释放相应事务的所有锁*

感悟：在事务中断条件下的代码实现时，没有理解将事务恢复状态的含义。一开始，我只是简单的把在缓冲池内的被事务修改过的页面进行删除，本以为这样就是变相实现了将页面状态恢复到修改前的状态（回滚），但是，AbortEvictTest函数始终无法通过。修改之后，从原文件内读取本应被修改的页面覆盖被修改过的页面后就可以了。

*//应该将脏页回滚成原本在磁盘上的状态 所以要重新读入磁盘中的对应页面再写入缓冲区 起到重新加载的效果* ***for***(PageId pid : *pages*.keySet()){  
 HeapPage page = (HeapPage)*pages*.get(pid);  
 ***if***(page.isDirty()!=***null***&&page.isDirty().equals(tid)){  
 HeapFile databaseFile = (HeapFile)Database.*getCatalog*().getDatabaseFile(page.getId().getTableId());  
 *//获取到了要被修改前的页面* Page beforeModifyPage = databaseFile.readPage(page.getId());  
 *//将该页面覆盖掉被中断事物处理的页面  
 pages*.put(page.**pid**,beforeModifyPage);  
  
 }  
 }  
  
}

2022年3月21日星期一21时45分28秒

****Exercise 5.****

**2.8. Deadlocks and Aborts**

It is possible for transactions in SimpleDB to deadlock (if you do not understand why, we recommend reading about deadlocks in Ramakrishnan & Gehrke). You will need to detect this situation and throw a TransactionAbortedException.

There are many possible ways to detect deadlock. A strawman example would be to implement a simple timeout policy that aborts a transaction if it has not completed after a given period of time. For a real solution, you may implement cycle-detection in a dependency graph data structure as shown in lecture. In this scheme, you would check for cycles in a dependency graph periodically or whenever you attempt to grant a new lock, and abort something if a cycle exists. After you have detected that a deadlock exists, you must decide how to improve the situation. Assume you have detected a deadlock while transaction t is waiting for a lock. If you're feeling homicidal, you might abort ****all**** transactions that t is waiting for; this may result in a large amount of work being undone, but you can guarantee that t will make progress. Alternately, you may decide to abort t to give other transactions a chance to make progress. This means that the end-user will have to retry transaction t.

Another approach is to use global orderings of transactions to avoid building the wait-for graph. This is sometimes preferred for performance reasons, but transactions that could have succeeded can be aborted by mistake under this scheme. Examples include the WAIT-DIE and WOUND-WAIT schemes.

Implement deadlock detection or prevention in src/simpledb/BufferPool.java. You have many design decisions for your deadlock handling system, but it is not necessary to do something highly sophisticated. We expect you to do better than a simple timeout on each transaction. A good starting point will be to implement cycle-detection in a wait-for graph before every lock request, and you will receive full credit for such an implementation. Please describe your choices in the lab writeup and list the pros and cons of your choice compared to the alternatives.

You should ensure that your code aborts transactions properly when a deadlock occurs, by throwing a TransactionAbortedException exception. This exception will be caught by the code executing the transaction (e.g., TransactionTest.java), which should call transactionComplete() to cleanup after the transaction. You are not expected to automatically restart a transaction which fails due to a deadlock -- you can assume that higher level code will take care of this.

We have provided some (not-so-unit) tests in test/simpledb/DeadlockTest.java. They are actually a bit involved, so they may take more than a few seconds to run (depending on your policy). If they seem to hang indefinitely, then you probably have an unresolved deadlock. These tests construct simple deadlock situations that your code should be able to escape.

Note that there are two timing parameters near the top of DeadLockTest.java; these determine the frequency at which the test checks if locks have been acquired and the waiting time before an aborted transaction is restarted. You may observe different performance characteristics by tweaking these parameters if you use a timeout-based detection method. The tests will output TransactionAbortedExceptions corresponding to resolved deadlocks to the console.

Your code should now should pass the TransactionTest system test (which may also run for quite a long time depending on your implementation).

At this point, you should have a recoverable database, in the sense that if the database system crashes (at a point other than transactionComplete()) or if the user explicitly aborts a transaction, the effects of any running transaction will not be visible after the system restarts (or the transaction aborts.) You may wish to verify this by running some transactions and explicitly killing the database server.

任务：

完成死锁检测代码。

理解：

死锁处理只需要在发现死锁时抛出TransactionAbortException就可以了，高层次的代码会进行死锁处理。

所以，该练习的重点在于死锁检测。按照讲义中的提示，可以设计一个简单的超时死锁检测，一个事务申请锁的时间过久，就认定为出现死锁状态，需要抛出异常。这里我是通过次数计算来进行死锁检测，两者原理基本相同，也比较容易。

死锁检测也可以采用图的环检测来实现。

死锁发生的原因是：

申请事务申请的资源的持有者正在直接或间接地等待申请者所持有的资源。

此处的资源就是页。我们需要对申请者申请的页的持有者进行递归判断，如果发现持有者有对申请者事务持有资源的申请（直接），说明包含一个只有两个事务的死锁环；或者持有者拥有的对某个资源的申请的资源持有者的申请，该持有者对最初的持有的资源也有申请的话（间接），说明存在一个包含多个事务的死锁环。

|  |
| --- |
| 如图，括号内P1,P2,P3为资源,T1,T2,T3为事务 |
|  | \* 虚线以及其上的字母R加上箭头组成了拥有关系，如果是字母W则代表正在等待写锁 |
|  | \* 例如下图左上方T1到P1的一连串符号表示的是T1此时拥有P1的读锁 |
|  | \* 图的边缘可以是虚线的转折点，例如为了表示T2正在等待P1 |
|  | \* <p> |
|  | \* // T1---R-->P1<------- |
|  | \* // W |
|  | \* // ---------------------- |
|  | \* // W |
|  | \* // ---T2---R-->P2<------- |
|  | \* // W |
|  | \* // ---------------------- |
|  | \* // W |
|  | \* // ---T3---R-->P3 |
|  | \* <p> |
|  | \* 上图的含义是，Ti拥有了对Pi的读锁(1<=i<=3) |
|  | \* 因为T1在P1上有了读锁，所以T2正在等待P1的写锁 |
|  | \* 同理，T3正在等待P2的写锁 |
|  |  |

*/\*\*  
 \* 死锁检测  
 \** ***@param tid*** *\** ***@param pid*** *\** ***@return*** *\* 死锁发生的原因是：  
 \* 申请事务申请的资源的持有者正在直接或间接地等待申请者所持有的资  
 \*  
\* \*/****public synchronized boolean*** deadlockDetect(TransactionId tid,PageId pid){  
 *//申请者所需资源的持有者* List<LockStat> holders = **lockTable**.get(pid);  
 *//持有者不存在或者为0 说明申请者可以直接申请 不存在死锁情况* ***if***(holders==***null***||holders.size()==0){  
 ***return false***;  
 }  
  
 *//获取申请者所持有的资源 用于判断抽有者是否直接或间接申请申请者的资源* ArrayList<PageId> applicantPages = getAllTranscationStat(tid);  
  
 *//开始寻找是否存在环  
 //对每个持有者进行查找* ***for***(LockStat holder : holders){  
 TransactionId transactionId = holder.getTransactionId();  
 *//去掉T1，因为虽然上图没画出这种情况，但T1可能同时也在其他Page上有读锁，这会影响判断结果* ***if*** (!transactionId.equals(tid)) {  
 ***boolean*** isWaitting = isWaitting(holder,applicantPages,tid);  
 ***if***(isWaitting) ***return true***;  
 }  
  
 }  
  
 ***return false***;  
}  
  
*/\*\*  
 \*在持有者中寻找是否存在正在等待申请者持有的资源（直接或间接）  
 \** ***@param holder*** *\** ***@param applicantPages*** *\** ***@param tid*** *\** ***@return*** *\*/****private boolean*** isWaitting(LockStat holder, ArrayList<PageId> applicantPages, TransactionId tid) {  
 *//首先判断当前持有者是否有正在等待的锁申请* TransactionId holderTransactionId = holder.getTransactionId();  
 PageId pageId = **waitTable**.get(holderTransactionId);  
 *//不存在等待的资源* ***if***(pageId==***null***){  
 ***return false***;  
 }  
  
 *//存在等待的资源 就对申请者持有的资源继续迭代 查看是否存在直接等待申请者的资源* ***for***(PageId applicantpage : applicantPages){  
 *//持有者正在等待申请者持有的资源 说明存在死锁* ***if***(applicantpage.equals(pageId)){  
 ***return true***;  
 }  
 }  
  
 *//来到这里 说明不存在直接申请持有者资源的死锁状态，但可能存在间接的* List<LockStat> lockStats = **lockTable**.get(pageId);  
 *//对当前的持有者进行递归的判断  
 //如果waitingPage的拥有者们(去掉toRemove)中的某一个正在等待pids中的某一个，说明是tid间接在等待* ***if***(lockStats==***null***||lockStats.size()==0) ***return false***;  
 ***for***(LockStat lockStat : lockStats){  
 TransactionId transactionId = lockStat.getTransactionId();  
 ***if***(!transactionId.equals(tid)){  
 ***boolean*** waitting = isWaitting(lockStat, applicantPages, tid);  
 ***if***(waitting){  
 ***return true***;  
 }  
 }  
 }  
  
 ***return false***;  
}

感悟：难点在于对死锁环的检测，此处需要递归去查找并判断是否存在互相持有并申请彼此所需资源的情况。同时在递归过程中需要取出申请者事务，因为申请者事务可能持有多个资源的读锁，如果不去除，可能会出现判断错误。这些代码的思路是借鉴的。

2022年3月22日星期二13时52分50秒

总结：

实验4的重点是练习1，通过练习1我们构造了一个简单的锁管理器。通过该锁管理器，在BufferPool的getPage函数内对每个事务申请页时进行加锁的操作。练习2，3都是在锁管理器的基础上对代码进行基本补充，如驱逐函数的优化。练习4完善了事务提交和终止时的处理，提交时才对缓冲池内的脏页写回，中止时需要对缓冲池内的页面进行回滚操作，使其回到修改前的状态，这里我直接从文件内读取对应页面到缓冲池内进行覆盖操作，变相实现回滚操作。最后的难点练习5要求实现死锁检测，有两种方法：简单的超时统计和较难的图环检测，详细见练习5的理解。

感悟AGAIN

以下是在做实验5时遇到了调试很多天都没有调试过的bug情况下，回头才发现虽然实验4都完成了并通过了单元测试以及一个系统EvictAbort测试，但是还漏掉了一个System/TranscationTest忘了测试！！！难受啊！今天暂时不改了，改了一天，心累(T\_T)

发现时间：2022年3月29日星期二21时41分13秒

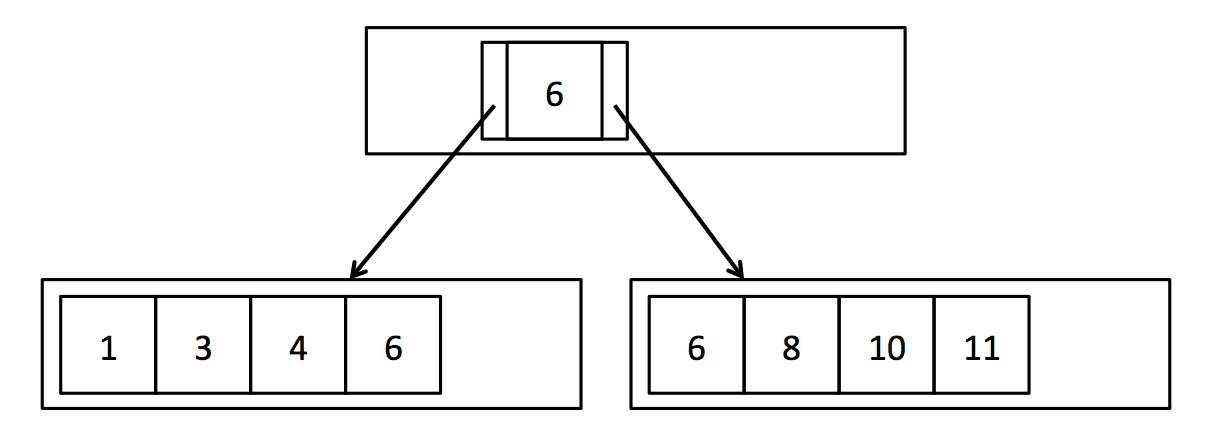
修改后正常通过。

LAB5

#### **2. Search**

Take a look at index/ and BTreeFile.java. This is the core file for the implementation of the B+Tree and where you will write all your code for this lab. Unlike the HeapFile, the BTreeFile consists of four different kinds of pages. As you would expect, there are two different kinds of pages for the nodes of the tree: internal pages and leaf pages. Internal pages are implemented in BTreeInternalPage.java, and leaf pages are implemented in BTreeLeafPage.java. For convenience, we have created an abstract class in BTreePage.java which contains code that is common to both leaf and internal pages. In addition, header pages are implemented in BTreeHeaderPage.java and keep track of which pages in the file are in use. Lastly, there is one page at the beginning of every BTreeFile which points to the root page of the tree and the first header page. This singleton page is implemented in BTreeRootPtrPage.java. Familiarize yourself with the interfaces of these classes, especially BTreePage, BTreeInternalPage and BTreeLeafPage. You will need to use these classes in your implementation of the B+Tree.

Your first job is to implement the findLeafPage() function in BTreeFile.java. This function is used to find the appropriate leaf page given a particular key value, and is used for both searches and inserts. For example, suppose we have a B+Tree with two leaf pages (See Figure 1). The root node is an internal page with one entry containing one key (6, in this case) and two child pointers. Given a value of 1, this function should return the first leaf page. Likewise, given a value of 8, this function should return the second page. The less obvious case is if we are given a key value of 6. There may be duplicate keys, so there could be 6's on both leaf pages. In this case, the function should return the first (left) leaf page.

  
*Figure 1: A simple B+ Tree with duplicate keys*

Your findLeafPage() function should recursively search through internal nodes until it reaches the leaf page corresponding to the provided key value. In order to find the appropriate child page at each step, you should iterate through the entries in the internal page and compare the entry value to the provided key value. BTreeInternalPage.iterator() provides access to the entries in the internal page using the interface defined in BTreeEntry.java. This iterator allows you to iterate through the key values in the internal page and access the left and right child page ids for each key. The base case of your recursion happens when the passed-in BTreePageId has pgcateg() equal to BTreePageId.LEAF, indicating that it is a leaf page. In this case, you should just fetch the page from the buffer pool and return it. You do not need to confirm that it actually contains the provided key value f.

Your findLeafPage() code must also handle the case when the provided key value f is null. If the provided value is null, recurse on the left-most child every time in order to find the left-most leaf page. Finding the left-most leaf page is useful for scanning the entire file. Once the correct leaf page is found, you should return it. As mentioned above, you can check the type of page using the pgcateg() function in BTreePageId.java. You can assume that only leaf and internal pages will be passed to this function.

Instead of directly calling BufferPool.getPage() to get each internal page and leaf page, we recommend calling the wrapper function we have provided, BTreeFile.getPage(). It works exactly like BufferPool.getPage(), but takes an extra argument to track the list of dirty pages. This function will be important for the next two exercises in which you will actually update the data and therefore need to keep track of dirty pages.

Every internal (non-leaf) page your findLeafPage() implementation visits should be fetched with READ\_ONLY permission, except the returned leaf page, which should be fetched with the permission provided as an argument to the function. These permission levels will not matter for this lab, but they will be important for the code to function correctly in future labs.

****Exercise 1: BTreeFile.findLeafPage()****

Implement BTreeFile.findLeafPage().

After completing this exercise, you should be able to pass all the unit tests in BTreeFileReadTest.java and the system tests in BTreeScanTest.java.

任务：完成b+树文件的搜索操作。

理解：搜索操作在目前的实验中算是相对简单的，而且讲义中的提示也很直接明了。只要熟悉了b+树结构，在b+树内部进行递归查找就好。在实现递归时，需要在内部页的进行迭代，利用内部页的迭代器对其内部的entry进行一一遍历比较，直到找到大于等于目标值的范围，找到后就比较大小，选择递归的是左孩子还是右孩子，直到递归到叶子页，就说明找到了目标值所在页。

感悟：难点在于此实验中有许多关于b+树的类，理解并掌握其构成是相对难的,在这个练习中花费的主要时间在阅读源码，了解类上。

Take a look at index/ and BTreeFile.java. This is the core file for the implementation of the B+Tree and where you will write all your code for this lab. Unlike the HeapFile, the BTreeFile consists of four different kinds of pages. As you would expect, there are two different kinds of pages for the nodes of the tree: internal pages and leaf pages. Internal pages are implemented in BTreeInternalPage.java, and leaf pages are implemented in BTreeLeafPage.java. For convenience, we have created an abstract class in BTreePage.java which contains code that is common to both leaf and internal pages. In addition, header pages are implemented in BTreeHeaderPage.java and keep track of which pages in the file are in use. Lastly, there is one page at the beginning of every BTreeFile which points to the root page of the tree and the first header page. This singleton page is implemented in BTreeRootPtrPage.java. Familiarize yourself with the interfaces of these classes,

especially BTreePage, BTreeInternalPage and BTreeLeafPage. You will need to use these classes in your implementation of the B+Tree.

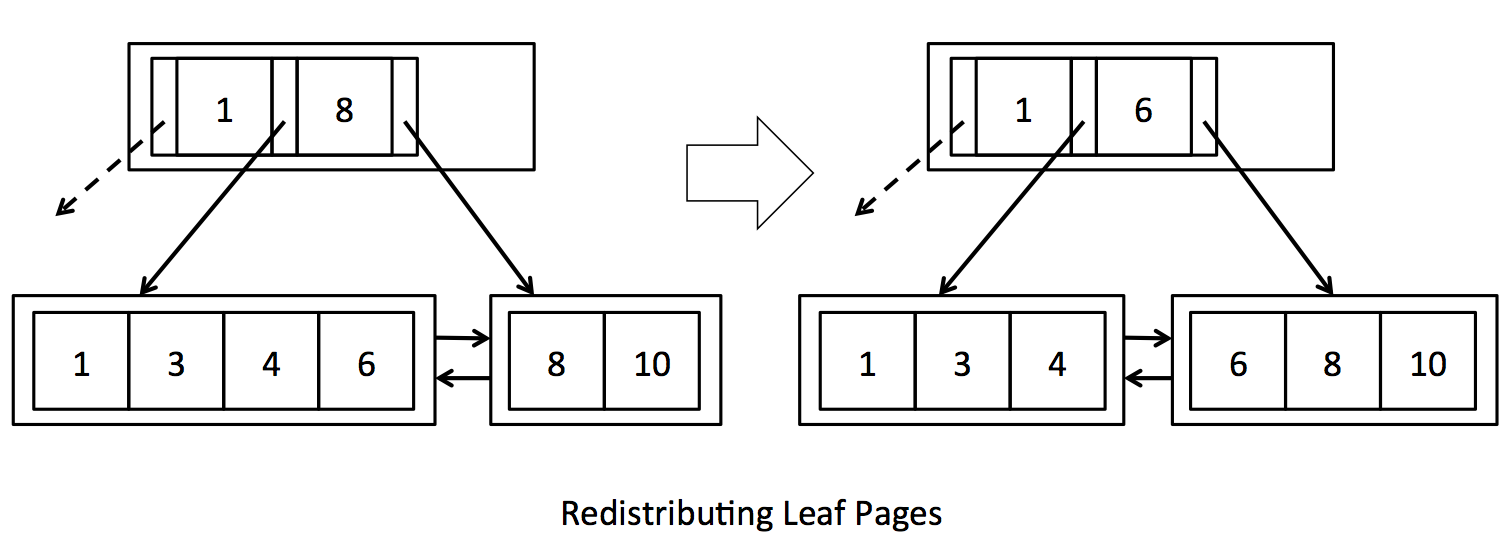
2022年3月22日22时6分8秒

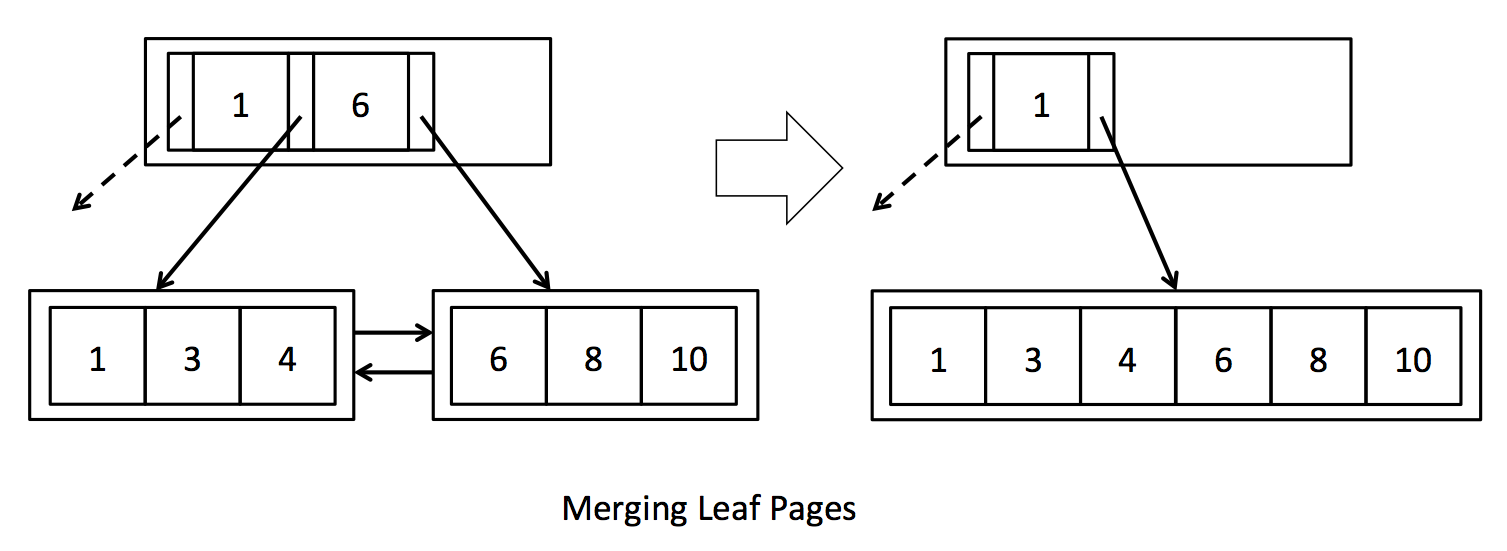
#### **3. Insert**

In order to keep the tuples of the B+Tree in sorted order and maintain the integrity of the tree, we must insert tuples into the leaf page with the enclosing key range. As was mentioned above, findLeafPage() can be used to find the correct leaf page into which we should insert the tuple. However, each page has a limited number of slots and we need to be able to insert tuples even if the corresponding leaf page is full.

As described in the textbook, attempting to insert a tuple into a full leaf page should cause that page to split so that the tuples are evenly distributed between the two new pages. Each time a leaf page splits, a new entry corresponding to the first tuple in the second page will need to be added to the parent node. Occasionally, the internal node may also be full and unable to accept new entries. In that case, the parent should split and add a new entry to its parent. This may cause recursive splits and ultimately the creation of a new root node.

In this exercise you will implement splitLeafPage() and splitInternalPage() in BTreeFile.java. If the page being split is the root page, you will need to create a new internal node to become the new root page, and update the BTreeRootPtrPage. Otherwise, you will need to fetch the parent page with READ\_WRITE permissions, recursively split it if necessary, and add a new entry. You will find the function getParentWithEmptySlots() extremely useful for handling these different cases. In splitLeafPage() you should "copy" the key up to the parent page, while in splitInternalPage() you should "push" the key up to the parent page. See Figure 2 and review section 10.5 in the text book if this is confusing. Remember to update the parent pointers of the new pages as needed (for simplicity, we do not show parent pointers in the figures). When an internal node is split, you will need to update the parent pointers of all the children that were moved. You may find the function updateParentPointers() useful for this task. Additionally, remember to update the sibling pointers of any leaf pages that were split. Finally, return the page into which the new tuple or entry should be inserted, as indicated by the provided key field. (Hint: You do not need to worry about the fact that the provided key may actually fall in the exact center of the tuples/entries to be split. You should ignore the key during the split, and only use it to determine which of the two pages to return.)



  
*Figure 2: Splitting pages*

Whenever you create a new page, either because of splitting a page or creating a new root page, call getEmptyPage() to get the new page. This function is an abstraction which will allow us to reuse pages that have been deleted due to merging (covered in the next section).

We expect that you will interact with leaf and internal pages using BTreeLeafPage.iterator() and BTreeInternalPage.iterator() to iterate through the tuples/entries in each page. For convenience, we have also provided reverse iterators for both types of pages: BTreeLeafPage.reverseIterator() and BTreeInternalPage.reverseIterator(). These reverse iterators will be especially useful for moving a subset of tuples/entries from a page to its right sibling.

As mentioned above, the internal page iterators use the interface defined in BTreeEntry.java, which has one key and two child pointers. It also has a recordId, which identifies the location of the key and child pointers on the underlying page. We think working with one entry at a time is a natural way to interact with internal pages, but it is important to keep in mind that the underlying page does not actually store a list of entries, but stores ordered lists of m keys and m+1 child pointers. Since the BTreeEntry is just an interface and not an object actually stored on the page, updating the fields of BTreeEntry will not modify the underlying page. In order to change the data on the page, you need to call BTreeInternalPage.updateEntry(). Furthermore, deleting an entry actually deletes only a key and a single child pointer, so we provide the funtions BTreeInternalPage.deleteKeyAndLeftChild() and BTreeInternalPage.deleteKeyAndRightChild() to make this explicit. The entry's recordId is used to find the key and child pointer to be deleted. Inserting an entry also only inserts a key and single child pointer (unless it's the first entry), so BTreeInternalPage.insertEntry() checks that one of the child pointers in the provided entry overlaps an existing child pointer on the page, and that inserting the entry at that location will keep the keys in sorted order.

In both splitLeafPage() and splitInternalPage(), you will need to update the set of dirtypages with any newly created pages as well as any pages modified due to new pointers or new data. This is where BTreeFile.getPage() will come in handy. Each time you fetch a page, BTreeFile.getPage() will check to see if the page is already stored in the local cache (dirtypages), and if it can't find the requested page there, it fetches it from the buffer pool. BTreeFile.getPage() also adds pages to the dirtypages cache if they are fetched with read-write permission, since presumably they will soon be dirtied. One advantage of this approach is that it prevents loss of updates if the same pages are accessed multiple times during a single tuple insertion or deletion.

Note that in a major departure from HeapFile.insertTuple(), BTreeFile.insertTuple() could return a large set of dirty pages, especially if any internal pages are split. As you may remember from previous labs, the set of dirty pages is returned to prevent the buffer pool from evicting dirty pages before they have been flushed.

****Warning****: as the B+Tree is a complex data structure, it is helpful to understand the properties necessary of every legal B+Tree before modifying it. Here is an informal list:

1. If a parent node points to a child node, the child nodes must point back to those same parents.
2. If a leaf node points to a right sibling, then the right sibling points back to that leaf node as a left sibling.
3. The first and last leaves must point to null left and right siblings respectively.
4. Record Id's must match the page they are actually in.
5. A key in a node with non-leaf children must be larger than any key in the left child, and smaller than any key in the right child.
6. A key in a node with leaf children must be larger or equal than any key in the left child, and smaller or equal than any key in the right child.
7. A node has either all non-leaf children, or all leaf children.
8. A non-root node cannot be less than half full.

We have implemented a mechanized check for all these properties in the file BTreeChecker.java. This method is also used to test your B+Tree implementation in the systemtest/BTreeFileDeleteTest.java. Feel free to add calls to this function to help debug your implementation, like we did in BTreeFileDeleteTest.java.

****N.B.****

The checker method should always pass after initialization of the tree and before starting and after completing a full call to key insertion or deletion, but not necessarily within internal methods.

A tree may be well formed (and therefore pass checkRep()) but still incorrect. For example, the empty tree will always pass checkRep(), but may not always be correct (if you just inserted a tuple, the tree should not be empty). \*\*\*

****Exercise 2: Splitting Pages****

Implement BTreeFile.splitLeafPage() and BTreeFile.splitInternalPage().

After completing this exercise, you should be able to pass the unit tests in BTreeFileInsertTest.java. You should also be able to pass the system tests in systemtest/BTreeFileInsertTest.java. Some of the system test cases may take a few seconds to complete. These files will test that your code inserts tuples and splits pages correcty, and also handles duplicate tuples.

任务：完成函数splitLeafPage，splitInternalPage.

理解：讲义十分详细，具体，按照引导和提示所给的思路以及辅组函数就可以变相分裂函数代码。主要的难点在一些细节上的把控，如分裂函数时，对函数分裂成两部分时，如何删除和插入。以及创建了新的叶子节点或内部页节点时，如何恢复其父母以及兄弟节点的连接。这些都是容易出错的细节。

主要的代码逻辑

splitLeafPage：  
*//1.创建页面*BTreeLeafPage newRight=(BTreeLeafPage) getEmptyPage(tid,dirtypages,BTreePageId.***LEAF***);  
*//开始复制操作 分裂成完整的两部分*Iterator<Tuple> iterator = page.reverseIterator();  
***int*** numTuples = page.getNumTuples();  
***int*** count =0;  
***for***(***int*** i=0;i<numTuples/2;i++){  
 Tuple next = iterator.next();  
 page.deleteTuple(next);  
 newRight.insertTuple(next);  
  
}  
  
*//2.保存原叶子节点的右兄弟*BTreePageId rightSiblingId = page.getRightSiblingId();  
  
*//3.将分裂后的右表中的第一个元组作为该新表的父节点*Tuple newParentNodeField = newRight.iterator().next();  
BTreeInternalPage parentNode = getParentWithEmptySlots(tid, dirtypages, page.getParentId()  
 , newParentNodeField.getField(**keyField**));  
  
*//4向父母节点中“copy”新的entry*BTreeEntry newEntry = ***new*** BTreeEntry(newParentNodeField.getField(**keyField**),page.getId(),newRight.getId());  
parentNode.insertEntry(newEntry);  
  
*//5，重新连接父母节点以及兄弟节点  
 //重连父母节点*updateParentPointers(tid, dirtypages,parentNode);*//父母节点  
 //连接兄弟*page.setRightSiblingId(newRight.getId());*//左页连右页*newRight.setLeftSiblingId(page.getId());*//右页连左页*newRight.setRightSiblingId(rightSiblingId);  
***if***(rightSiblingId!=***null***){  
 BTreeLeafPage page1 = (BTreeLeafPage)getPage(tid, dirtypages, rightSiblingId, Permissions.***READ\_ONLY***);  
 page1.setLeftSiblingId(newRight.getId());  
}  
  
*//6.找到新插入的元组应该所在的节点****return*** (field.compare(Op.***GREATER\_THAN\_OR\_EQ***,newParentNodeField.getField(**keyField**))?newRight:page);

splitInternalPage

*//1.创造新的内部页，复制一半的entry到新页内  
//2.保存新页的第一个元组，删除该元组  
//3.递归获取父母节点，将第一个元组放入父母节点  
//4.重新连接父母节点  
//5.返回field应该在的内部页  
//1.*BTreeInternalPage newRight =(BTreeInternalPage) getEmptyPage(tid, dirtypages, BTreePageId.***INTERNAL***);  
Iterator<BTreeEntry> bTreeEntryIterator = page.reverseIterator();  
*//if(bTreeEntryIterator==null||!bTreeEntryIterator.hasNext()) throw new DbException("");****int*** numOfEntry = page.getNumEntries();  
  
***for***(***int*** i=0;i<=numOfEntry/2;i++){  
 BTreeEntry next = bTreeEntryIterator.next();  
 page.deleteKeyAndRightChild(next);  
 newRight.insertEntry(next);  
}  
*//2.*BTreeEntry firstOfRight = newRight.iterator().next();  
BTreeEntry newEntry = ***new*** BTreeEntry(firstOfRight.getKey(),page.getId(),newRight.getId());  
  
*//3.*BTreeInternalPage parent = getParentWithEmptySlots(tid, dirtypages, page.getParentId(), firstOfRight.getKey());  
newRight.deleteKeyAndRightChild(firstOfRight);  
parent.insertEntry(newEntry);  
  
*//4.  
//此步很容易被忽略 因为创建了新的内部页，原先属于左页的子节点都需要更新*updateParentPointers(tid,dirtypages,newRight);  
updateParentPointers(tid,dirtypages,parent);  
  
  
***return*** field.compare(Op.***GREATER\_THAN\_OR\_EQ***,firstOfRight.getKey())?newRight:page;

感悟：

在进行测试，出现了一些边界上的错误，以及代码逻辑的错误

*逻辑错误：出错原因是对树索引结构文件还是不是非常熟悉，需要多看看源码加深理解*

*//下面注释的代码是错误的，如果调用findLeafPage函数去寻找新的父母节点下目标域field所在的叶子节点  
//会出现不存在内部页的异常，因为分割函数就是在插入元组时叶子节点空间不足才调用的，而在该分割函数内的  
//新建的父母节点其实还不存在于B+树文件内，只是我们在分割时的一个中间变量，无法通过findLeafPage找到.  
//所以此处的页面返回应该通过插入值field与新建右页newRigt第一个元组的比较来选择返回哪个页面  
//BTreeLeafPage leafPage = findLeafPage(tid, parentNode.getId(), field);  
//return leafPage;*

*//下面注释代码是在测试案例testSplitInternalPages出现叶子节点的原因  
//因为在内部页分裂之后，新右页第一个entry需要被传递新的父母节点中，并且在新的父母节点中连接  
//page与newRight作为其左右子节点，实现entry的“pushed up”  
//而此处我是将原本应该分割的页中的entry作为新的entry加入父母节点中，这毫无疑问是错误的  
//会导致在本应该出现内部页的迭代中出现属于叶子节点的页  
//parent.insertEntry(newEntry);*BTreeEntry newEntry = ***new*** BTreeEntry(firstOfRight.getKey(),page.getId(),newRight.getId());

*边界错误：不够仔细，在测试案例中未通过时反复变量才发现*

*//原本的边界条件是i<numOfEntry*

***for***(***int*** i=0;i<=numOfEntry/2;i++){  
 BTreeEntry next = bTreeEntryIterator.next();  
 page.deleteKeyAndRightChild(next);  
 newRight.insertEntry(next);  
}

上述的插入操作其实本质上并不难，只要仔细阅读讲义，（讲义真的很详细了），剩下的就是一些逻辑上的修剪，以及细节的调配。分裂叶子节点以及内部页节点逻辑基本相通，主要区别就是在右页的第一个元组的copy和push。

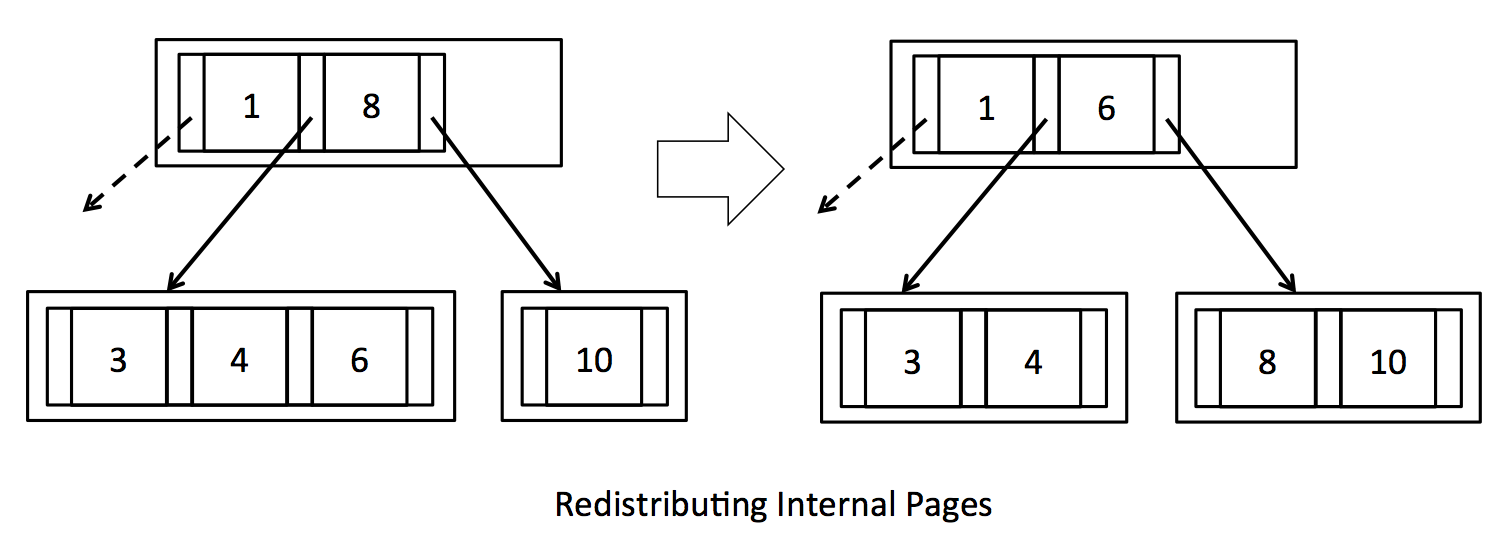
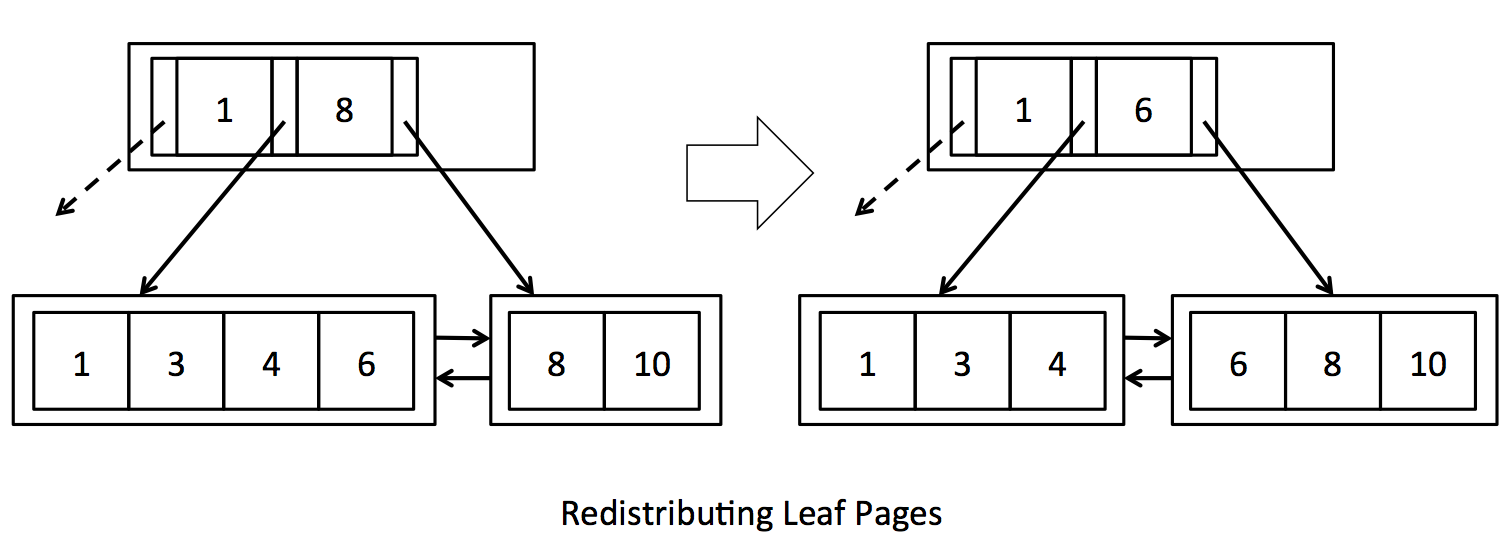
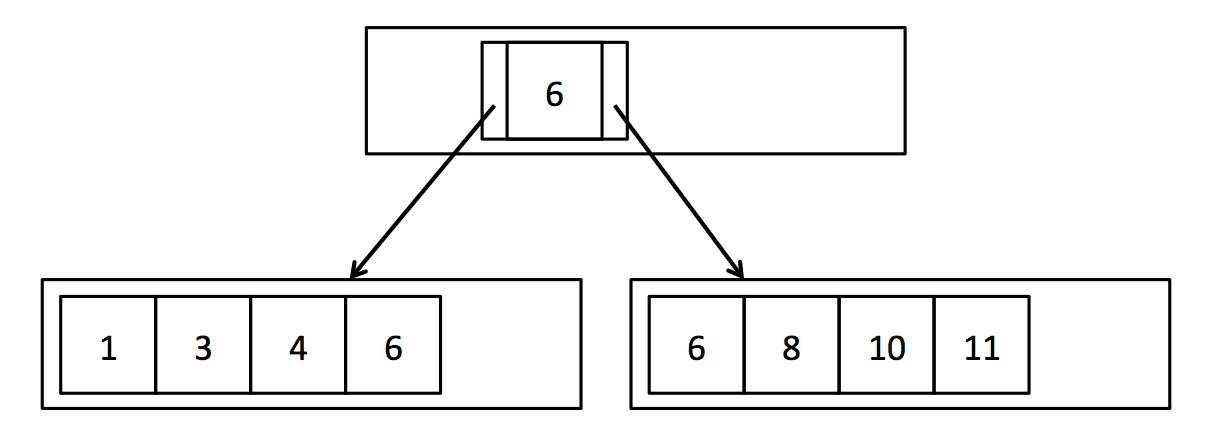
22022年3月23日星期三22时32分26秒

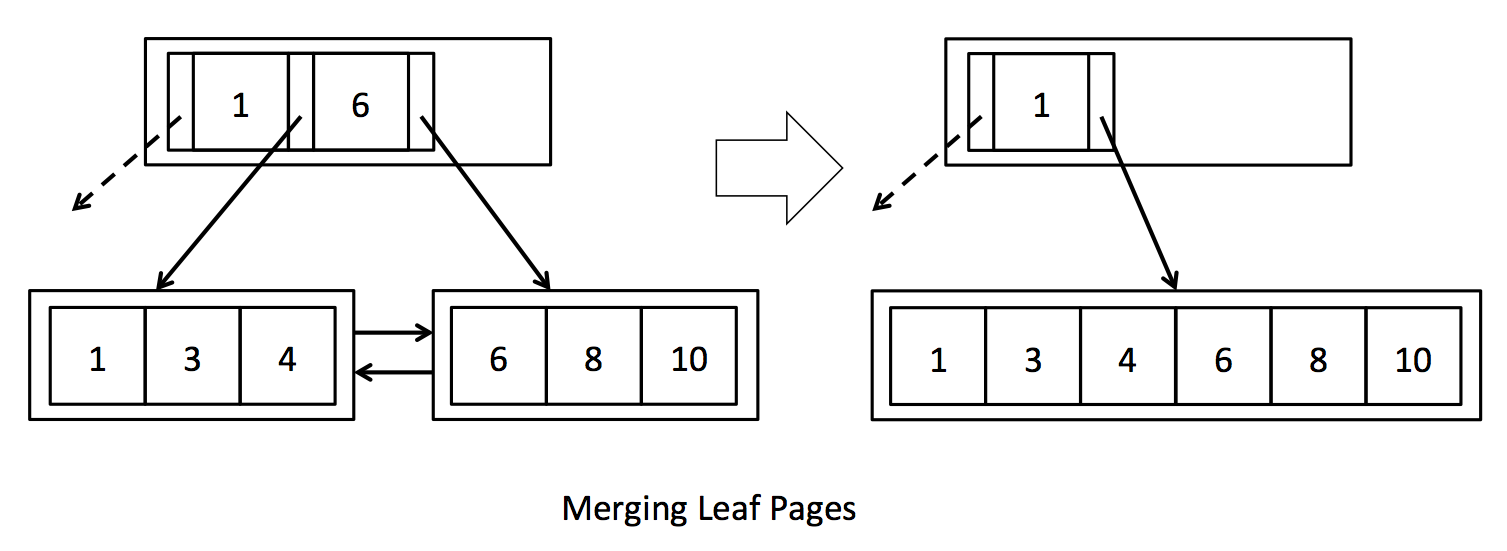
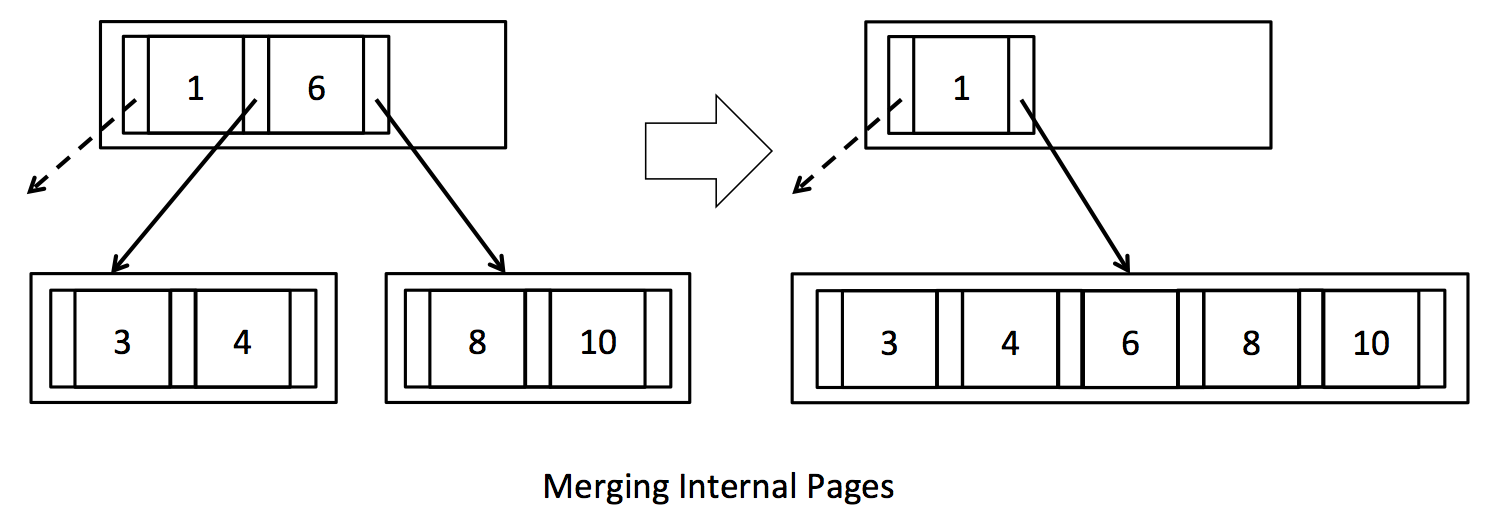
调试了一星期的bug终于都过了，具体的splitRootPage和splitInternalPage通过了。

2022年3月31日星期三22时32分26秒

#### **4. Delete**

In order to keep the tree balanced and not waste unnecessary space, deletions in a B+Tree may cause pages to redistribute tuples (Figure 3) or, eventually, to merge (see Figure 4). You may find it useful to review section 10.6 in the textbook.

  
  
*Figure 3: Redistributing pages*

  
  
*Figure 4: Merging pages*

As described in the textbook, attempting to delete a tuple from a leaf page that is less than half full should cause that page to either steal tuples from one of its siblings or merge with one of its siblings. If one of the page's siblings has tuples to spare, the tuples should be evenly distributed between the two pages, and the parent's entry should be updated accordingly (see Figure 3). However, if the sibling is also at minimum occupancy, then the two pages should merge and the entry deleted from the parent (Figure 4). In turn, deleting an entry from the parent may cause the parent to become less than half full. In this case, the parent should steal entries from its siblings or merge with a sibling. This may cause recursive merges or even deletion of the root node if the last entry is deleted from the root node.

In this exercise you will implement stealFromLeafPage(), stealFromLeftInternalPage(), stealFromRightInternalPage(), mergeLeafPages() and mergeInternalPages() in BTreeFile.java. In the first three functions you will implement code to evenly redistribute tuples/entries if the siblings have tuples/entries to spare. Remember to update the corresponding key field in the parent (look carefully at how this is done in Figure 3 - keys are effectively "rotated" through the parent). In stealFromLeftInternalPage()/stealFromRightInternalPage(), you will also need to update the parent pointers of the children that were moved. You should be able to reuse the function updateParentPointers() for this purpose.

In mergeLeafPages() and mergeInternalPages() you will implement code to merge pages, effectively performing the inverse of splitLeafPage() and splitInternalPage(). You will find the function deleteParentEntry() extremely useful for handling all the different recursive cases. Be sure to call setEmptyPage() on deleted pages to make them available for reuse. As with the previous exercises, we recommend using BTreeFile.getPage() to encapsulate the process of fetching pages and keeping the list of dirty pages up to date.

****Exercise 3: Redistributing pages****

Implement BTreeFile.stealFromLeafPage(), BTreeFile.stealFromLeftInternalPage(), BTreeFile.stealFromRightInternalPage().

After completing this exercise, you should be able to pass some of the unit tests in BTreeFileDeleteTest.java (such as testStealFromLeftLeafPage and testStealFromRightLeafPage). The system tests may take several seconds to complete since they create a large B+ tree in order to fully test the system.

****Exercise 4: Merging pages****

Implement BTreeFile.mergeLeafPages() and BTreeFile.mergeInternalPages().

Now you should be able to pass all unit tests in BTreeFileDeleteTest.java and the system tests in systemtest/BTreeFileDeleteTest.java.

任务：完成要求的函数

理解：

删除操作直接在指定的叶子节点中删除元组，相对容易。但是为了保证树的均衡，在删除元组之后需要对树进行合并或者偷取操作。当一个叶子节点中的元组数量少于设定的最小值时，需要将左右页面进行合并保证查找时效率不会下降，同时，当元组数量大于设定值时，需要保证左右孩子的元组数量尽量均衡，所以需要从数量多的叶子节点那里偷取一些元组保持平衡。

主要的删除操作在deleteTuple函数内进行，同时该函数对树节点情况进行分类讨论，分别调用对应的merge或者steal函数，所以，主要的任务就是完善merge类函数和steal函数。

代码逻辑如下：

Steal

stealFromLeafPage

*//1.判断是对哪个节点进行steal操作  
// 2.计算左右两兄弟的差值  
// 3.先对左右子兄弟进行分配操作  
//4.分配完成，更新父节点中指向两孩子的entry的key//  
  
//1*Iterator<Tuple> iterator=***null***;  
***if*** (isRightSibling) {  
 iterator = sibling.iterator();  
} ***else*** {  
 iterator = sibling.reverseIterator();  
}  
  
*//2****int*** numToShare = (sibling.getNumTuples()-page.getNumTuples())/2;  
  
*//3****while***((numToShare--)!=0){  
 Tuple next = iterator.next();  
 sibling.deleteTuple(next);  
 page.insertTuple(next);  
}  
  
*//4*Tuple updateParentNode = sibling.iterator().next();  
entry.setKey(updateParentNode.getField(**keyField**));  
  
*//要进行更新否则无效*parent.updateEntry(entry);

stealFromLeftInternalPage

*//1.先将父母节点的key向需要添加元素的节点移动*Iterator<BTreeEntry> iterator = leftSibling.reverseIterator();  
BTreeEntry next = iterator.next();  
 *//细节 需要把右子树原先的最左entry的左子树作为新节点的右子树，同时，需要吧左子树的最右entry的右子树作为新节点的左子树，避免节点的丢失*BTreeEntry copyParentEntry = ***new*** BTreeEntry(parentEntry.getKey(), next.getRightChild(), page.iterator().next().getLeftChild());  
page.insertEntry(copyParentEntry);  
*//2.计算左右节点的差值，因为需要补充一个到父母节点，所以差值减1，最后差值除2就是剩余的还需要移动到待分配节点的entry数量****int*** entryNum = (leftSibling.getNumEntries()-page.getNumEntries()-1)/2;  
***while***((entryNum--)!=0){  
 leftSibling.deleteKeyAndRightChild(next);  
 page.insertEntry(next);  
 next = iterator.next();  
}  
*//3.将左边的最后一个移动到父母节点上（只要复制key值，并删除entry就好了*leftSibling.deleteKeyAndRightChild(next);  
parentEntry.setKey(next.getKey());  
parent.updateEntry(parentEntry);  
*//4.重连父子节点*updateParentPointers(tid,dirtypages,page);  
updateParentPointers(tid,dirtypages,parent);

Merge

mergeLeafPages

*//1.将右子节点中的tuple全部移动到左子节点  
//2.保存原先右子节点的右兄弟  
//3.将右子节点删除，利用setEmptyPage函数，使页面可以重复利用  
//4.删除多余的父节点，deleteParentEntry() 可以有效处理多种情况  
//5.更新兄弟节点关系*Iterator<Tuple> iterator = rightPage.iterator();  
*//1****while***(iterator.hasNext()){  
 Tuple next = iterator.next();  
 rightPage.deleteTuple(next);*//先删除后插入* leftPage.insertTuple(next);  
}  
*//2.*BTreePageId rightSiblingId = rightPage.getRightSiblingId();  
*//3.*setEmptyPage(tid, dirtypages,rightPage.getId().getPageNumber());  
*//4.*deleteParentEntry(tid,dirtypages,leftPage,parent,parentEntry);  
*//5.*leftPage.setRightSiblingId(rightSiblingId);  
***if***(rightSiblingId!=***null***){  
 BTreeLeafPage bTreeLeafPage =(BTreeLeafPage)getPage(tid,dirtypages,rightSiblingId,Permissions.***READ\_ONLY***);  
 bTreeLeafPage.setLeftSiblingId(leftPage.getId());  
}

mergeInternalPages

*//1.先将父母entry pull到左子树中 主要要更新该entry的左右子树再插入  
//2.将右子树中的entry删除并复制到左子树中  
//3.删除右子树  
//4.更新父子节点关系  
//5.删除原先的父母entry  
//1.*Iterator<BTreeEntry> iterator = rightPage.iterator();  
BTreeEntry next = iterator.next();  
BTreeEntry newInsert = ***new*** BTreeEntry(parentEntry.getKey(),leftPage.reverseIterator().next().getRightChild(),next.getLeftChild());  
leftPage.insertEntry(newInsert);  
  
*//2.*rightPage.deleteKeyAndRightChild(next);  
leftPage.insertEntry(next);  
***while***(iterator.hasNext()){  
 next = iterator.next();  
 rightPage.deleteKeyAndRightChild(next);  
 leftPage.insertEntry(next);  
}  
  
  
*//3.*setEmptyPage(tid,dirtypages,rightPage.getId().getPageNumber());  
*//4.*updateParentPointers(tid,dirtypages,leftPage);  
*//5.*deleteParentEntry(tid,dirtypages,leftPage,parent,parentEntry);

感悟：代码逻辑相对好理解，讲义也说得很明确，测试过程中出现了一些小细节问题

2022年4月2日星期六15时22分54秒

#### **5. Transactions**

You may remember that B+ trees can prevent phantom tuples from showing up between two consecutive range scans by using next-key locking. Since SimpleDB uses page-level, strict two-phase locking, protection against phantoms effectively comes for free if the B+ tree is implemented correctly. Thus, at this point you should also be able to pass BTreeNextKeyLockingTest.

Additionally, you should be able to pass the tests in test/simpledb/BTreeDeadlockTest.java if you have implemented locking correctly inside of your B+ tree code.

If everything is implemented correctly, you should also be able to pass the BTreeTest system test. We expect many people to find BTreeTest difficult, so it's not required, but we'll give extra credit to anyone who can run it successfully. Please note that this test may take up to a minute to complete.

感悟：除了BTreeTest没通过，其他都正常通过。

总结：

总体来说，这单元的各个练习都是考察B+树的基础实现原理。从第一个Search练习，对指定元组的查找，通过递归实现查找。第二个练习Insert，相对复杂，需要BTreeLeafPage和BTreeIntrenalPage进行分类讨论，容易在entry的copy或者push过程不小心误删或者医遗忘其左右孩子。这也是我耗时最久的地方。最后一个delete看起来讲义的文本量大，要实现的内容多，但是并不是很难理解，只要捋顺了实现过程，分别实现merge和steal操作就好，这一部分耗时相对较短，出错的地方也很快改正。

2022年4月2日星期六15时36分52秒

LAB6

#### **Rollback**

Read the comments in LogFile.java for a description of the log file format. You should see in LogFile.java a set of functions, such as logCommit(), that generate each kind of log record and append it to the log.

Your first job is to implement the rollback() function in LogFile.java. This function is called when a transaction aborts, before the transaction releases its locks. Its job is to un-do any changes the transaction may have made to the database.

Your rollback() should read the log file, find all update records associated with the aborting transaction, extract the before-image from each, and write the before-image to the table file. Use raf.seek() to move around in the log file, and raf.readInt() etc. to examine it. Use readPageData() to read each of the before- and after-images. You can use the map tidToFirstLogRecord (which maps from a transaction id to an offset in the heap file) to determine where to start reading the log file for a particular transaction. You will need to make sure that you discard any page from the buffer pool whose before-image you write back to the table file.

As you develop your code, you may find the Logfile.print() method useful for displaying the current contents of the log.

****Exercise 1: LogFile.rollback()****

Implement LogFile.rollback().

After completing this exercise, you should be able to pass the TestAbort and TestAbortCommitInterleaved sub-tests of the LogTest system test.

任务:完成logFile内的rollback函数

理解：

*/\*  
本应该从tid第一次出现的位置读取所有tid事务的更新记录，进行逆序重做的操作  
 但是update操作时是对整个页面进行更新，所有这里采用简化的方式，直接读取  
 事务tid第一次更新的页面操作，将数据直接恢复倒最初状态，不去一步一步回滚。  
\*/*

RollBack简单的读取先前信息写回即可。

Rollback操作其实就是读取记录条，如果是要回滚事务的更新记录，那就读取接下来的beforPage旧页，直接将该旧页写入文件，并把缓冲池内的页面丢弃掉（保证下次读取页面时，从文件中读取回滚了的页面）。

感悟：

Rollback操作并不复杂，代码量也不是非常大。而主要的时间花费在理解logFile结构上。理解了日志文件内5中记录的格式，就可以编写相应的代码。

2022年4月4日星期一12时7分15秒

*//1.找到事务tid所在得到位置  
 //2.开始读取文件，对不同记录分别处理  
 //4读到updateRecord，正式回滚文件，在抛弃缓冲中的该页，并且删除tidTofirstRecord记录  
 //  
  
 //1.* Long offset = **tidToFirstLogRecord**.get(tid.getId());  
 ***if***(offset!=***null***)  
 **raf**.seek(offset);  
  
 ***while*** (***true***) {  
 ***try*** {  
 *//2.* ***int*** type = **raf**.readInt();  
 ***long*** record\_tid = **raf**.readLong();  
 *//3.* ***switch*** (type) {  
 ***case UPDATE\_RECORD***:  
 Page before = readPageData(**raf**);  
 Page after = readPageData(**raf**);  
 ***if***(tid.getId()==record\_tid){  
 Database.*getCatalog*().getDatabaseFile(before.getId().getTableId()).writePage(before);  
 Database.*getBufferPool*().discardPage(before.getId());  
 **tidToFirstLogRecord**.remove(tid);  
 }  
 ***break***;  
 ***case CHECKPOINT\_RECORD***:  
 ***int*** numXactions = **raf**.readInt();  
 ***while*** (numXactions-- > 0) {  
 ***long*** xid = **raf**.readLong();  
 ***long*** xoffset = **raf**.readLong();  
 }  
 ***break***;  
 }  
  
 **raf**.readLong();  
 } ***catch*** (EOFException e) {  
 ***break***;  
 }  
 }  
  
 }  
}

#### **Recovery**

If the database crashes and then reboots, LogFile.recover() will be called before any new transactions start. Your implementation should:

1.Read the last checkpoint, if any.

2.Scan forward from the checkpoint (or start of log file, if no checkpoint) to build the set of loser transactions. Re-do updates during this pass. You can safely start re-do at the checkpoint because LogFile.logCheckpoint() flushes all dirty buffers to disk.

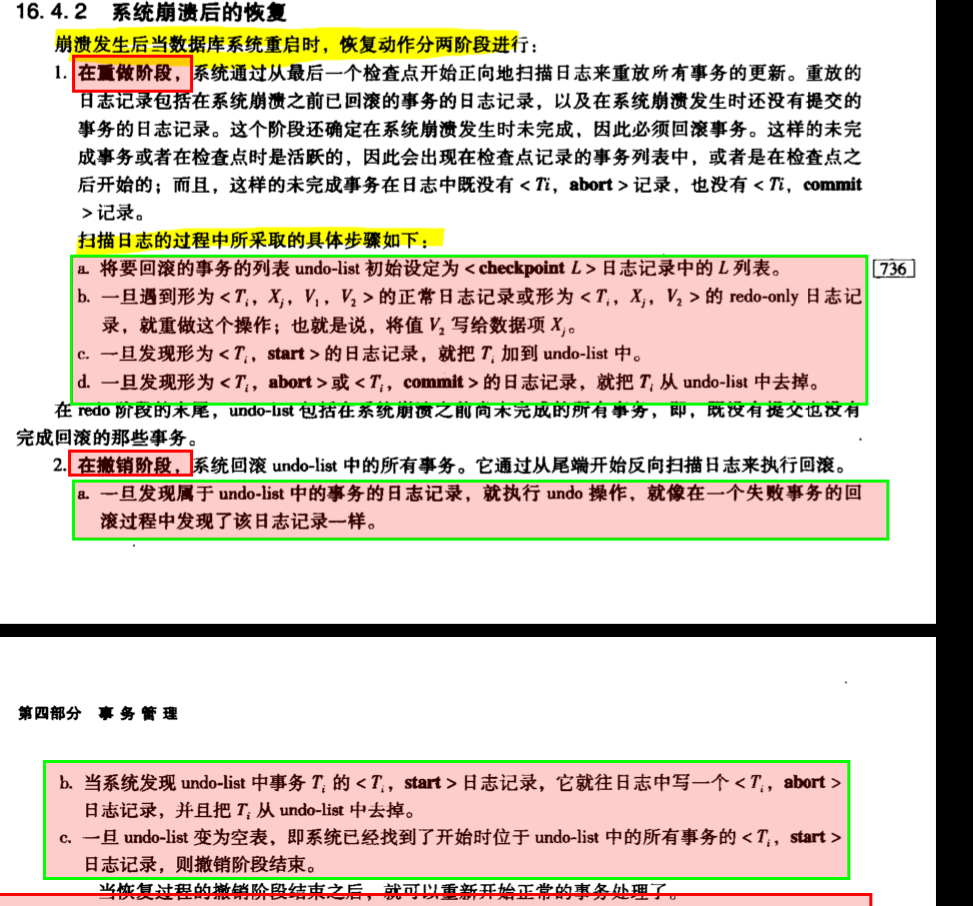
3.Un-do the updates of loser transactions.

****Exercise 2: LogFile.recover()****

Implement LogFile.recover().

After completing this exercise, you should be able to pass all of the LogTest system test.

任务：完成recocver函数的编写

理解：算法实现基本原理：

代码： *//1.找到最后一个检查点，获取该检查点时的活跃事务id，并用集合L保存  
 //2.重做阶段：开始从保存点向下查找每条记录，对每条更新记录进行redo操作。  
 // 若出现新的事务，加入集合。若出现事务的abort和commit记录，将该事务移出L  
 //3.扫描完成，进入撤销阶段  
 //4.撤销阶段：总日志末尾反向移动，每碰到日志中的一个事务update记录，就执行undo操作  
 // 并加入一条abort记录，并从L中移除该事务  
 //5.不断重复，直到集合L为空  
  
 // The first long integer of the file represents the offset of the  
 //last written checkpoint, or -1 if there are no checkpoints  
 //1.* print();  
 **raf**.seek(0);  
 ***long*** lastCheckPoint = **raf**.readLong();  
 *//所有需要undo操作的页面* Map<Long, Long> tid2offset = ***new*** HashMap<>();  
 *//某一事务操作的页面集合* Map<Long, List<PageId>> tid2pages = ***new*** HashMap<>();  
 *//页面id对应的页面* Map<PageId, Page> pages = ***new*** HashMap<>();  
  
 *//获取活跃的事务* ***if*** (lastCheckPoint != -1) {  
 *//定位倒checkpoint所在位置* **raf**.seek(lastCheckPoint);  
 *//获取checkpoint记录信息* ***int*** type = **raf**.readInt();*//类型* ***if*** (type == ***CHECKPOINT\_RECORD***) {  
 **raf**.readLong();*//事务id* ***int*** nums = **raf**.readInt();  
 ***for*** (***int*** i = 0; i < nums; i++) {  
 Long tid = **raf**.readLong();  
 Long offset = **raf**.readLong();  
 tid2offset.put(tid, offset);  
 }  
 }  
 **raf**.readLong();  
 }  
 *//为已存在的活跃事务初始化一个页集合* Set<Long> longs = tid2offset.keySet();  
 ***for*** (Long tid : longs) {  
 *//需要该集合在rollback函数与rocver进行建立连接，在重做阶段找到要撤销的事务，将其与偏移量加在一起，为rollback使用。* **tidToFirstLogRecord**.put(tid,tid2offset.get(tid));  
 tid2pages.put(tid, ***new*** ArrayList<>());  
 }  
  
  
  
 *//2.* ***while*** (***true***) {  
 ***try*** {  
 ***int*** type = **raf**.readInt();  
 ***long*** record\_tid = **raf**.readLong();  
  
 ***switch*** (type) {  
 *//更新就保存页面，直到遇见commit和abort再进行处理* ***case UPDATE\_RECORD***:  
 Page before = readPageData(**raf**);  
 Page after = readPageData(**raf**);  
 **raf**.readLong();  
 *//只保存after页面 等到遇到commit时再正式更新* tid2pages.get(record\_tid).add(after.getId());  
 pages.put(after.getId(), after);  
 ***break***;  
  
 ***case BEGIN\_RECORD***:  
 *//出现begin，将该事务加入* ***long*** offset = **raf**.readLong();  
 **tidToFirstLogRecord**.put(record\_tid,offset);  
 tid2offset.put(record\_tid,offset);  
 tid2pages.put(record\_tid,***new*** ArrayList<>());  
 ***break***;  
  
 ***case ABORT\_RECORD***:  
 *//出现终止 剔除tid2offset中的对应事务* **tidToFirstLogRecord**.remove(record\_tid);  
 ArrayList<PageId> pageArrayList = (ArrayList<PageId>) tid2pages.get(record\_tid);  
 tid2pages.remove(record\_tid);  
 ***for*** (PageId pageId : pageArrayList) {  
 pages.remove(pageId);  
 }  
 **raf**.readLong();  
 ***break***;  
  
 ***case COMMIT\_RECORD***:  
 *//将tid2offset中的事务提交* ArrayList<PageId> pagelist = (ArrayList<PageId>) tid2pages.get(record\_tid);  
 ***for*** (PageId pageId : pagelist) {  
 Page page = pages.get(pageId);  
 Database.*getCatalog*().getDatabaseFile(page.getId().getTableId()).writePage(page);  
 page.setBeforeImage();  
 }  
 **tidToFirstLogRecord**.remove(record\_tid);  
 tid2pages.remove(record\_tid);  
 **raf**.readLong();  
 ***break***;  
 }  
 } ***catch*** (EOFException e) {  
 ***break***;  
 }  
 }  
  
  
  
*//4.撤销阶段***currentOffset** = **raf**.getFilePointer();  
*//获取需要进行撤销操作的事务的记录位置，方便重做后写入abort信息****for*** (Long tid : tid2offset.keySet()) {  
 **raf**.seek(**currentOffset**);  
 **raf**.writeInt(***ABORT\_RECORD***);  
 **raf**.writeLong(tid);  
 **raf**.writeLong(**currentOffset**);  
 *//开始回滚* force();  
 rollback(tid);  
 *//将abort记录设置在文件末尾* **currentOffset** = **raf**.length(); }  
}  
**raf**.writeLong(**currentOffset**);  
print();

感悟：难点在于对日志文件格式的理解，如果不熟练，很容易忽略一些值的读入，导致后续的记录读取出错，可以使用print函数来实时观察日志文件，进而调试。

2022年4月5日星期二14时5分39秒

总结：

本实验实现了简单日志文件系统，只是在事务出现异常崩溃时恢复相应的事务。崩溃时，只需要找到检查点（if exist）,重做接下来的日志记录，并收集需要回滚的事务，再从文件末尾反向进行回滚操作。总的来说，该恢复机制比较简易。

总结

总算把基本实验全部搞完了，零零散散的也花费了差不多一个月的时间。

虽然很多方面的代码实现都是参考的，但是在实现过程中，逐渐上手并梳理整个实验的过程，将自己对项目的理解与感悟都记录下来了。通过这个项目，我也学会了不少东西。(具体见对应章节）。

虽然项目大体完成了，但是某些地方还是可以改进的：

1. 使用lru策略取代fifo策略来实现换页的机制。
2. 使用蟹行协议（我也不太懂是什么东东）

后续可以找时间进行了解和优化。。。