# 运行环境

Ubuntu 18.04

git version 2.17.1

java version "1.8.0\_131"

# SimpleDB架构和实现

lab1主要实现数据库的储存结构。Lab中已经给出需要实现的方法，但可能用到的类的成员变量、辅助类和辅助方法等需要自己编写。

## Exercise 1

需要实现TupleDesc和Tuple两个类。

Tuple：元组，即表中储存数据的一行。

TupleDesc：每张表中有一个TupleDesc，用于描述表中元组的属性。

Field: Tuple由Field组成，Field为储存数据的单位。

Tuple类需要用到TupleDesc类的信息，所以先实现TupleDesc类再实现Tuple类。

### （一）TupleDesc类的实现

1、TupleDesc类里只写除了方法，分析可知还需要一个属性TDItem，用数组储存。

1. **private** **final** TDItem[] tdItems;

2、完善Iterator<TDItem> iterator()函数，由提示可知需要补充返回值，返回一个TupleDesc包含的所有的TDItem的迭代器

1. **public** Iterator<TDItem> iterator() {
2. // some code goes here
3. **return** (Iterator<TDItem>) Arrays.asList(tdItems).iterator();
4. }

3、完善TupleDesc()函数，这就是个初始化函数，根据参数不同有两种初始化方法。

1. **public** TupleDesc(Type[] typeAr, String[] fieldAr) {
2. // some code goes here
3. tdItems = **new** TDItem[typeAr.length];
4. **for**(**int** i=0;i<typeAr.length;++i){
5. tdItems[i] = **new** TDItem(typeAr[i],fieldAr[i]);
6. }
7. }

fieldName有可能为null，所以还有下面这种初始方法

1. **public** TupleDesc(Type[] typeAr) {
2. // some code goes here
3. tdItems = **new** TDItem[typeAr.length];
4. **for**(**int** i=0;i<typeAr.length;++i){
5. tdItems[i] = **new** TDItem(typeAr[i],"");
6. }
7. }

4、补充numField()函数的返回值，返回这个TupleDesc中Field的个数，也就是tdItem的长度。

1. **public** **int** numFields() {
2. // some code goes here
3. **return** tdItems.length;
4. }
5. getFieldName()函数返回FieldName即可，i超出索引时返回异常，其他的getFieldType函数也类似。
6. **public** String getFieldName(**int** i) **throws** NoSuchElementException {
7. // some code goes here
8. **if**(i<0 || i>= tdItems.length){
9. **throw** **new** NoSuchElementException("pos " + i + " is not a valid index");
10. }
11. **return** tdItems[i].fieldName;
12. }
13. **public** Type getFieldType(**int** i) **throws** NoSuchElementException {
14. // some code goes here
15. **if**(i<0 || i>=tdItems.length){
16. **throw** **new** NoSuchElementException("pos " + i + " is not a valid index");
17. }
18. **return** tdItems[i].fieldType;
19. }

fieldNameToIndex函数就是遍历tdItems数组，返回fieldname与给出name相同的索引数，若没有则返回异常。

1. **public** **int** fieldNameToIndex(String name) **throws** NoSuchElementException {
2. // some code goes here
3. **for**(**int** i=0;i<tdItems.length;++i){
4. **if**(tdItems[i].fieldName.equals(name)){
5. **return** i;
6. }
7. }
8. **throw** **new** NoSuchElementException("not find fieldName " + name);
9. }

getSize()返回TupleDesc对应Tuple的字节数。Type中存在getLen()方法，可获取INT\_TYPE和STRING\_TYPE字节数。遍历累加可得。

1. **public** **int** getSize() {
2. // some code goes here
3. **int** size = 0;
4. **for**(**int** i=0;i<tdItems.length;++i){
5. size += tdItems[i].fieldType.getLen();
6. }
7. **return** size;

equals()方法判断每一个TDItem是否相等。因此还需要重写TDItem的equals()方法。注意TDItem中fieldName为null的情况。

1. **public** **boolean** equals(Object o) {
2. // some code goes here
3. **if**(**this**.getClass().isInstance(o)) {
4. TupleDesc two = (TupleDesc) o;
5. **if** (numFields() == two.numFields()) {
6. **for** (**int** i = 0; i < numFields(); ++i) {
7. **if** (!tdItems[i].fieldType.equals(two.tdItems[i].fieldType)) {
8. **return** **false**;
9. }
10. }
11. **return** **true**;
12. }
13. }
14. **return** **false**;
15. }

merge()函数就是将两个TupleDesc合并到一起。

1. **public** **static** TupleDesc merge(TupleDesc td1, TupleDesc td2) {
2. // some code goes here
3. Type[] typeAr = **new** Type[td1.numFields() + td2.numFields()];
4. String[] fieldAr = **new** String[td1.numFields() + td2.numFields()];
5. **for**(**int** i=0;i<td1.numFields();++i){
6. typeAr[i] = td1.tdItems[i].fieldType;
7. fieldAr[i] = td1.tdItems[i].fieldName;
8. }
9. **for**(**int** i=0;i<td2.numFields();++i){
10. typeAr[i+td1.numFields()] = td2.tdItems[i].fieldType;
11. fieldAr[i+td1.numFields()] = td2.tdItems[i].fieldName;
12. }
13. **return** **new** TupleDesc(typeAr,fieldAr);
14. }

ToString()函数，将所有的fieldname和fieldtype连接在一起，注意一下格式即可。

1. **public** String toString() {
2. // some code goes here
3. StringBuilder sb = **new** StringBuilder();
4. **for**(**int** i=0;i<tdItems.length-1;++i){
5. sb.append(tdItems[i].fieldName + "(" + tdItems[i].fieldType + "), ");
6. }
7. sb.append(tdItems[tdItems.length-1].fieldName + "(" + tdItems[tdItems.length-1].fieldType + ")");
8. **return** sb.toString();
9. }

### （二）Tuple类的实现

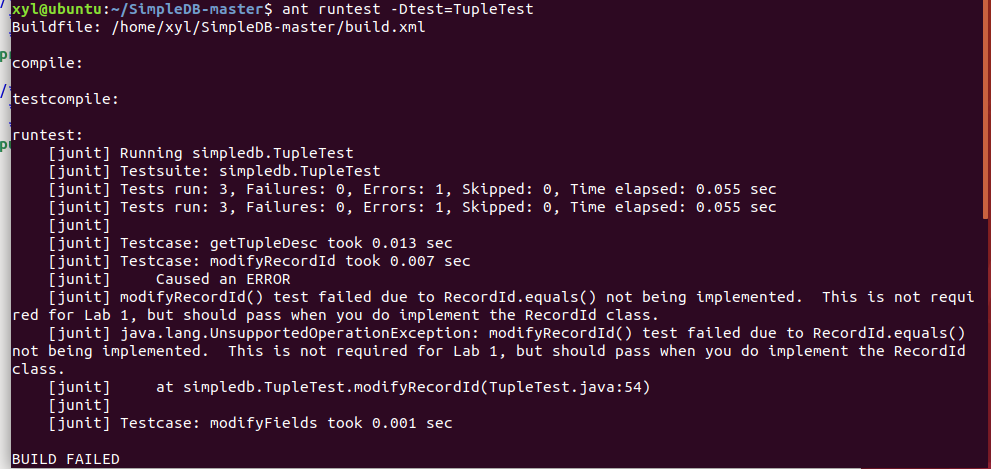
Tuple用数组储存Field即可。其中RecordId相关方法暂时不用实现。

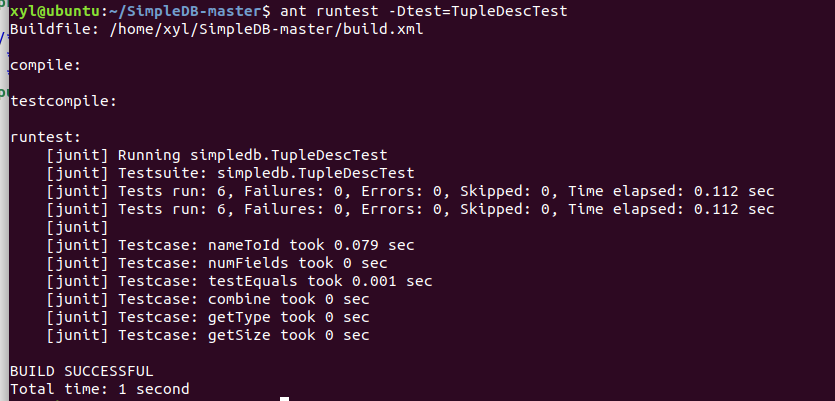
分析给出的set、get函数可知Tuple类有以下属性：

1. **private** TupleDesc tupleDesc;
2. **private** RecordId recordId;
3. **private** **final** Field[] fields;

set函数设置属性值，get函数返回属性值，toString函数和TupleDesc类中的类似，不再赘述。

实现Tuple和TupleDesc后，可以通过TupleTest和TupleDescTest单元测试。其中，TupleTest的modifyRecordId()会fail，因为还没有实现RecordId的相关方法。





## Exercise 2

需要实现Catalog类。

Catalog：目录。数据库包含很多张表，每张表有一个TupleDesc，以及这个TupleDesc规范下的很多个Tuple。Catalog管理着数据库中的所有表。由于数据库Database为单例设计，而调用数据库的Catalog需要调用Database.getCatalog()方法，因此Catalog也可以视作单例。

DbFile：为数据库磁盘文件的接口。数据库中每张表对应着一个DbFile，DbFile储存着表中的所有信息。

在Catalog中，似乎并没有关于表的直接表示。因此，可以创建一个Table内部类作为辅助，也许这样更符合直觉。

1. **private** **static** **class** Table{
2. **private** **static** **final** **long** serialVersionUID = 1L;
4. **public** **final** DbFile dbFile;
5. **public** **final** String tableName;
6. **public** **final** String pk;
8. **public** Table(DbFile file,String name,String pkeyField){
9. dbFile = file;
10. tableName = name;
11. pk = pkeyField;
12. }
14. **public** String toString(){
15. **return** tableName + "(" + dbFile.getId() + ":" + pk +")";
16. }
17. }

DbFile中提供了getId()方法，可以获取此Dbfile对应表的tableid。注意，这个id并不是顺序生成，而是哈希生成（lab1后续exercise会涉及)，因此Catalog应使用HashMap储存Table。

1. **private** **final** ConcurrentHashMap<Integer,Table> hashTable;

创建一个空目录，也就是创建一个空的hashtable用于存储table.

1. **public** Catalog() {
2. // some code goes here
3. hashTable = **new** ConcurrentHashMap<Integer,Table>();
4. }

创建一个新表并存储到hashtable中。

1. **public** **void** addTable(DbFile file, String name, String pkeyField) {
2. // some code goes here
3. Table t = **new** Table(file,name,pkeyField);
4. hashTable.put(file.getId(),t);
5. }

set、get函数不再赘述。

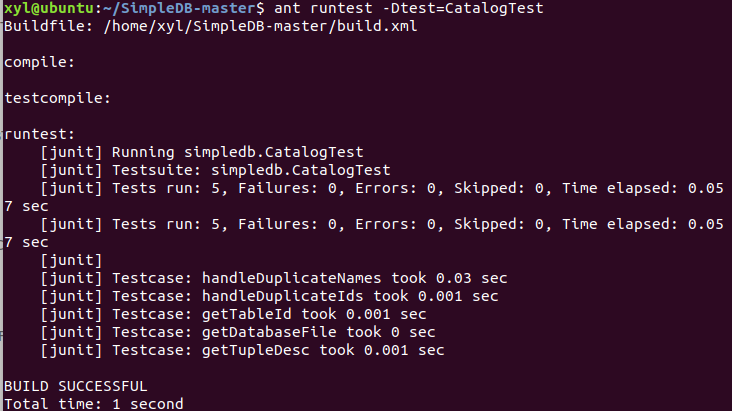
tableId就是哈希时使用的关键值，tableIdIterator返回一个迭代器。

1. **public** Iterator<Integer> tableIdIterator() {
2. // some code goes here
3. **return** hashTable.keySet().iterator();
4. }

删除所有的表，清空存储表的hashTable即可。

1. **public** **void** clear() {
2. // some code goes here
3. hashTable.clear();
4. }

实现Catalog后，可通过CatalogTest单元测试。



## Exercise 3

buffer pool（在SimpleDB中是BufferPool类）负责将内存最近读过的物理页缓存下来。所有的读写操作通过buffer pool读写硬盘上不同文件，BufferPool里的numPages参数确定了读取的固定页数，在之后的lab中，需要实现淘汰机制(eviction policy)。在这个lab中，只需要实现构造函数和BufferPool.getPage()方法，BufferPool应该存取最多numPages个物理页，当前lab中如果页的数量超过numPages，先不实现eviction policy，先扔出一个DbException错误。

增加两个属性

1. **private** **final** **int** numPages;
2. **private** **final** ConcurrentHashMap<Integer,Page> pageStore;
3. **public** BufferPool(**int** numPages) {
4. // some code goes here
5. **this**.numPages = numPages;
6. pageStore = **new** ConcurrentHashMap<Integer,Page>();
7. }
8. **public**  Page getPage(TransactionId tid, PageId pid, Permissions perm)
9. **throws** TransactionAbortedException, DbException {
10. // some code goes here
11. **if**(!pageStore.containsKey(pid.hashCode())){
12. DbFile dbfile = Database.getCatalog().getDatabaseFile(pid.getTableId());
13. Page page = dbfile.readPage(pid);
14. pageStore.put(pid.hashCode(),page);
15. }
16. **return** pageStore.get(pid.hashCode());
17. }

exercise 3 暂时没有单元测试。

## Exercise 4

### HeapPageId

PageId的实现。储存了HeapPage所在HeapFile 的tableid，以及自身在HeapFile中的序号。

1. 完善类属性和构造函数。
2. **private** **final** **int** tableId;
3. **private** **final** **int** pgNo;
4. **public** HeapPageId(**int** tableId, **int** pgNo) {
5. // some code goes here
6. **this**.tableId = tableId;
7. **this**.pgNo = pgNo;
8. }
9. 两个get函数直接返回属性值即可。
10. **public** **int** getTableId() {
11. // some code goes here
12. **return** tableId;
13. }
14. **public** **int** getPageNumber() {
15. // some code goes here
16. **return** pgNo;
17. }

3、hashCode()函数：返回tableId和 pgNo连接起来的字符串的哈希码，使用字符串类自带hashCode()函数。

1. **public** **int** hashCode() {
2. // some code goes here
3. String hash = "" + tableId +pgNo;
4. **return** hash.hashCode();
5. }

4、判断两个对象是否为相同，首先要确保这个对象是PageId类的实例，然后判断tableId和pgNo是否相同。

1. **public** **boolean** equals(Object o) {
2. // some code goes here
3. **if**(o **instanceof** PageId){
4. PageId pi = (PageId) o;
5. **if**(pi.getTableId() == tableId && pi.getPageNumber() == pgNo){
6. **return** **true**;
7. }
8. }
9. **return** **false**;
10. }

### RecordId

RecordId：与Tuple绑定，储存了Tuple所在Page的 PageId，以及自身在Page中的序号。在理解了RecordId的用途后，就可以去实现之前exercise 1里Tuple的RecordId相关方法了。

1. 完善类属性和构造方法
2. **private** **final** PageId pid;
3. **private** **final** **int** tupleno;
4. **public** RecordId(PageId pid, **int** tupleno) {
5. // some code goes here
6. **this**.pid = pid;
7. **this**.tupleno = tupleno;
8. }
9. get函数直接返回属性值。
10. **public** **int** getTupleNumber() {
11. // some code goes here
12. **return** tupleno;
13. }
14. **public** PageId getPageId() {
15. // some code goes here
16. **return** pid;
17. }

3、如果两个RecordId表示的是同一个tuple返回true，RecordId的特异性由PageId和tupleno决定。

1. **public** **boolean** equals(Object o) {
2. // some code goes here
3. **if**(o **instanceof** RecordId){
4. RecordId ro = (RecordId) o;
5. **if**(ro.getPageId().equals(pid) && ro.getTupleNumber() == tupleno){
6. **return** **true**;
7. }
8. }
9. **return** **false**;
10. }

4、根据tableId和pgNo判断PageId是否相同，根据tableId、pgNo和tupleno就可以判断是否为同一RecordId,然后连接成字符串返回哈希码即可。

1. **public** **int** hashCode() {
2. // some code goes here
3. String hash = "" + pid.getTableId()+pid.getPageNumber() + tupleno;
4. **return** hash.hashCode();
5. }

### HeapPage

1、get函数直接计算并返回属性值。

SimpleDB数据库的每个tuple需要tuple size \* 8 bits 的内容大小和1 bit的header大小。因此，在一页中可以包含的tuple数量计算公式是：\_tuples per page\_ = floor((\_page size\_ \* 8) / (\_tuple size\_ \* 8 + 1))。其中，tuple size是页中单个tuple 的bytes大小。

一旦知道了每页中能够保存的tuple数量，需要的header的物理大小是：headerBytes = ceiling(tupsPerPage/8)。

1. **private** **int** getNumTuples() {
2. // some code goes here
3. //\_tuples per page\_ = floor((\_page size\_ \* 8) / (\_tuple size\_ \* 8 + 1))
4. **int** num = (**int**)Math.floor((BufferPool.getPageSize()\*8\*1.0)/(td.getSize()\*8+1));
5. **return** num;
6. }
7. **private** **int** getHeaderSize() {
9. // some code goes here
10. // headerBytes = ceiling(tupsPerPage/8)
11. **return** (**int**)Math.ceil(getNumTuples()\*1.0/8);
12. }
13. **public** HeapPageId getId() {
14. // some code goes here
15. **return** pid;
16. }

2、直接遍历slot，返回未被使用的slot的个数。

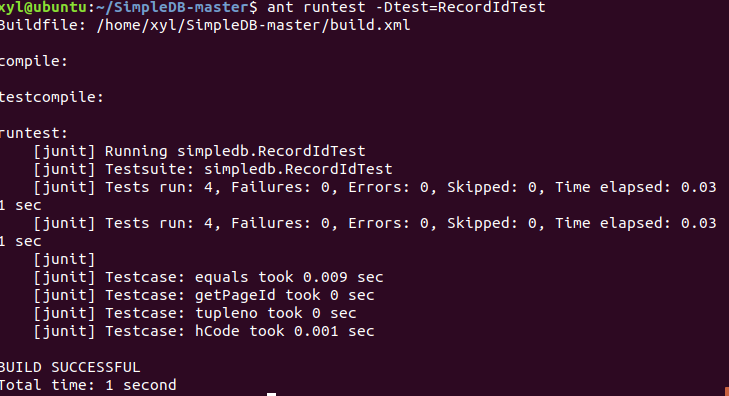
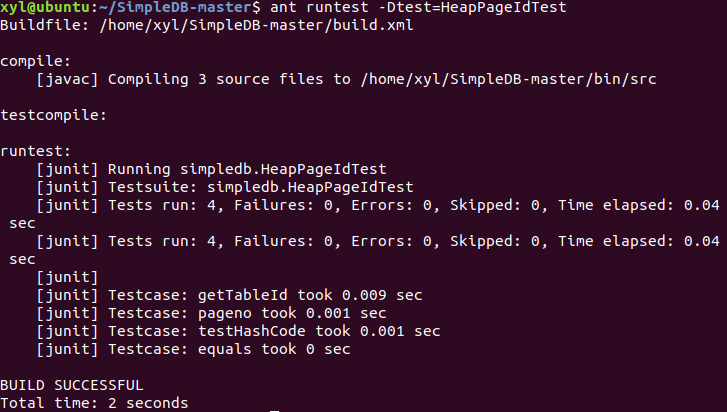
1. **public** **int** getNumEmptySlots() {
2. // some code goes here
3. **int** cnt = 0;
4. **for**(**int** i=0;i<numSlots;++i){
5. **if**(!isSlotUsed(i)){
6. ++cnt;
7. }
8. }
9. **return** cnt;
10. }
11. header的储存方式为byte数组，一个byte有8个字节，所以

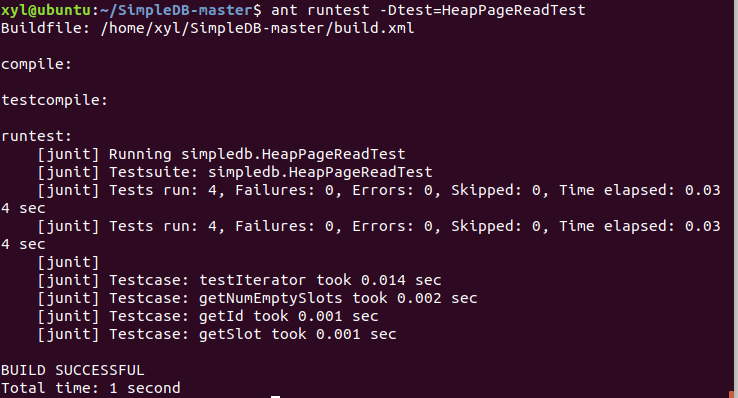
byteIndex = i / 8

bitIndex = i % 8。

1. **public** **boolean** isSlotUsed(**int** i) {
2. **int** byteIndex = i / 8;
3. **int** bitIndex = i % 8;
4. **int** flag = (header[byteIndex] >> bitIndex) & 1;
5. **return** flag == 1;
6. }
7. 只需要注意迭代器中存储的都是有效的Tuple。
8. **public** Iterator<Tuple> iterator() {
9. // some code goes here
10. ArrayList<Tuple> filledTuples = **new** ArrayList<Tuple>();
11. **for**(**int** i=0;i<numSlots;++i){
12. **if**(isSlotUsed(i)){
13. filledTuples.add(tuples[i]);
14. }
15. }
16. **return** filledTuples.iterator();
17. }

在实现HeapPageId、RecordId和HeapPage三个类后，可以通过HeapPageIdTest、RecordIdTest和HeapPageReadTest单元测试。





## Exercise 5

1. 完善属性和构造函数。
2. **private** **final** File file;
3. **private** **final** TupleDesc td;
4. **public** HeapFile(File f, TupleDesc td) {
5. // some code goes here
6. **this**.file = f;
7. **this**.td = td;
8. }

2、getFile()和getTupleDesc()直接返回属性值。

1. **public** File getFile() {
2. // some code goes here
3. **return** file;
4. }
5. **public** TupleDesc getTupleDesc() {
6. // some code goes here
7. **return** td;
8. }

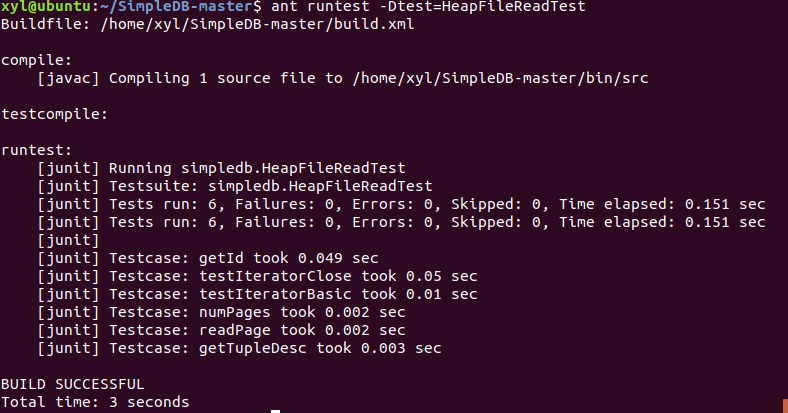
3、getId()方法返回对应表的tableid，可利用File中getAbsoluteFile().hashCode()方法生成。

1. **public** **int** getId() {
2. // some code goes here
3. **return** file.getAbsoluteFile().hashCode();
4. }

4、readPage()方法中，可以使用RandomAccessFile类实现对文件任意位置的访问。readPage()方法仅会被BufferPool中的getPage()方法调用，而在其他位置需要获取page时，均要通过BufferPool调用。这也是BufferPool的意义所在。

1. **public** Page readPage(PageId pid) {
2. // some code goes here
3. **int** tableId = pid.getTableId();
4. **int** pgNo = pid.getPageNumber();
6. RandomAccessFile f = **null**;
7. **try**{
8. f = **new** RandomAccessFile(file,"r");
9. **if**((pgNo+1)\*BufferPool.getPageSize() > f.length()){
10. f.close();
11. **throw** **new** IllegalArgumentException(String.format("table %d page %d is invalid", tableId, pgNo));
12. }
13. **byte**[] bytes = **new** **byte**[BufferPool.getPageSize()];
14. f.seek(pgNo \* BufferPool.getPageSize());
15. // big end
16. **int** read = f.read(bytes,0,BufferPool.getPageSize());
17. **if**(read != BufferPool.getPageSize()){
18. **throw** **new** IllegalArgumentException(String.format("table %d page %d read %d bytes", tableId, pgNo, read));
19. }
20. HeapPageId id = **new** HeapPageId(pid.getTableId(),pid.getPageNumber());
21. **return** **new** HeapPage(id,bytes);
22. }**catch** (IOException e){
23. e.printStackTrace();
24. }**finally** {
25. **try**{
26. f.close();
27. }**catch** (Exception e){
28. e.printStackTrace();
29. }
30. }
31. **throw** **new** IllegalArgumentException(String.format("table %d page %d is invalid", tableId, pgNo));
32. }

实现HeapFile类后，可以通过HeapFileReadTest。



## Exercise 6

实现SeqScan类。

1. 完善类属性和构造函数。
2. **private** final TransactionId tid;
3. **private** **int** tableId;
4. **private** String tableAlias;
5. **private** DbFileIterator it;
7. **public** SeqScan(TransactionId tid, **int** tableid, String tableAlias) {
8. // some code goes here
9. **this**.tid = tid;
10. **this**.tableId = tableid;
11. **this**.tableAlias = tableAlias;
12. }
13. getAlias()函数返回属性值。
14. **public** String getAlias()
15. {
16. // some code goes here
17. **return** tableAlias;
18. }
19. getTupleDesc()函数不能直接返回原tupleDesc，需要为tupleDesc的fieldName加上表的别名前缀，即alias.fieldName。
20. **public** TupleDesc getTupleDesc() {
21. // some code goes here
22. TupleDesc tupleDesc = Database.getCatalog().getTupleDesc(tableId);
23. **int** numFields = tupleDesc.numFields();
24. Type[] types = **new** Type[numFields];
25. String[] fieldNames = **new** String[numFields];
26. String prefix = "null.";
27. **if**(getAlias() != null) {
28. prefix = getAlias() + ".";
29. }
30. **for**(**int** i = 0; i < numFields; i++) {
31. types[i] = tupleDesc.getFieldType(i);
32. fieldNames[i] = prefix + tupleDesc.getFieldName(i);
33. }
34. **return** **new** TupleDesc(types, fieldNames);
35. }
36. reset()函数重置属性值。
37. **public** **void** reset(**int** tableid, String tableAlias) {
38. // some code goes here
39. **this**.tableId = tableid;
40. **this**.tableAlias = tableAlias;
41. }
42. 关于迭代器的几个函数，对exercise 5中实现的iterator做简单封装即可。
43. **public** **void** open() throws DbException, TransactionAbortedException {
44. // some code goes here
45. it = Database.getCatalog().getDatabaseFile(tableId).iterator(tid);
46. it.open();
47. **public** boolean hasNext() throws TransactionAbortedException, DbException {
48. // some code goes here
49. **if**(it == null){
50. **return** **false**;
51. }
52. **return** it.hasNext();
53. }
54. **public** Tuple next() throws NoSuchElementException,
55. TransactionAbortedException, DbException {
56. // some code goes here
57. **if**(it == null){
58. **throw** **new** NoSuchElementException("no next tuple");
59. }
60. Tuple t = it.next();
61. **if**(t == null){
62. **throw** **new** NoSuchElementException("no next tuple");
63. }
64. **return** t;
65. }
66. **public** **void** close() {
67. // some code goes here
68. it = null;
69. }
70. **public** **void** rewind() throws DbException, NoSuchElementException,
71. TransactionAbortedException {
72. // some code goes here
73. it.rewind();
74. }

实现SeqScan类后，可以通过ScanTest系统测试。

