南開大學
Nankai University

计算机学院

数据库系统实验报告

SimpleDB Lab-1

姓名：曹珉浩

学号：2113619

专业：计算机科学与技术

2023 年 3 月 17 日

目录

1 exercise-1 **2**
1.1 Tuple 2
1.2 TupleDesc 2

2 exercise-2 **4**

3 exercise-3 **4**

4 exercise-4 **5**
4.1 HeapPagedId 和 RecordId 5
4.2 HeapPage 5

5 exercise-5 **7**

6 exercise-6 **9**

7 Commit **10**

1 exercise-1

1.1 Tuple

元组是数据库中的行，由注释可知其包含由 TupleDesc 对象指定的指定模式并包含 Field 对象每个字段的数据，它继承自 Serializable 接口，Serializable 类是 Java 语言中的一个接口，用于表示一个类的对象可以被序列化为一个字节序列并在网络上传输或存储到磁盘上，这个接口没有任何的方法，只是标识这个类可以被序列化

读下面程序的 getter and setter 以及注释，添加了几个成员变量：

- TupleDesc td：指定模式
- LinkedList<Field> data：保存字段
- RecordId rid：

Tuple 类中的 getter and setter 很容易理解且容易书写，下面主要介绍 toString() 的设计思路：

在 Java 中，String 是不可变对象，每一个看上去会改变 String 的操作实际上都是创建了一个新对象，因此直接用 String 接受变长字符串会大大的影响效率，Java 中提供了两种用于构造字符串的类：StringBuffer 和 StringBuilder，StringBuilder 效率更高，但线程不安全，考虑到数据库的并发性，我们采取 StringBuffer 处理变长字符串，同时根据上述格式，得到代码如下：

```
1 public String toString() {
2       StringBuffer str=new StringBuffer();
3       for(int i=0;i<data.size()-1;i++) {
4         str.append(data.get(i).toString()+"\t");
5       }
6       str.append(data.get(data.size()-1)); //保持格式
7       return str.toString();
8   }
```

1.2 TupleDesc

TupleDesc 的作用是描述元组，读构造函数的注释可知内部有一个 array 变量，记录字段值的类型和名字，我们首先完成了简单的构造函数和 getter，setter 的编写，在完成获取字段类型和名字的函数时，注意到要考虑异常 (越界或负值)：

```
1 public String getFieldName(int i) throws NoSuchElementException {
2       if(i<0||i>=this.numFields())
3         throw new NoSuchElementException();
4       return items.get(i).fieldName;
5   }
```

在该类中，有一个重要的方法就是搜索：给定一个字段名，搜索它存不存在，设计这个函数时要注意 null 异常和比较对象时应使用类中的 equals() 方法，比较他们的值，简单遍历实现：

```
public int fieldNameToIndex(String name) throws NoSuchElementException {
    if(name==null)
      throw new NoSuchElementException();
    for(int i=0;i<items.size();i++) {
      if(name.equals(items.get(i).fieldName))
        return i;
    }
    throw new NoSuchElementException();
}
```

还有一个获取 record 所占内存大小的函数 getSize()，在静态内部类 TDItem 中，给定了组成 record 的类型 Type，读 Type.java 可知 Type 是一个枚举类，它只有 int 和 String 两种类型，类中已经实现了方法 getLen()，用于获取占用的字节，因此 getSize() 设计如下：

```
public int getSize() {
  int bytes=0;
  for(int i=0;i<items.size();i++)
    bytes+=this.getFieldType(i).getLen();
  return bytes;
 }
```

最后一个比较关键的功能是 equals() 函数，它用来比较两个对象的值是否相同，注意到传入的参数为 Object，因此可能存在把其他类的对象传入函数的可能，在判断时，可以判断一下类名是否相同，如果相同则可以将 Object 强制转型，这样就可以调用 TupleDesc 类中的方法了：

```
public boolean equals(Object o) {
  if(o==null)
    return false; //空对象返回 False
    if(o.getClass().equals(TupleDesc.class)) {
      TupleDesc temp=(TupleDesc) o; //类型转换
      if(this.items.size()!=temp.items.size())
        return false;
      for(int i=0;i<this.items.size();i++) //
       if(!items.get(i).fieldType.equals(temp.items.get(i).fieldType))
         return false;  //类型不同名字肯定不同，无需再比较了
       return true;
     }
     return false; //类型不对直接返回 False;
 }
```

2 exercise-2

Catalog 跟踪数据库中的所有可用表及其关联的模式, 目前，这是一个存根目录，在使用之前必须由用户程序填充表, 最终，它应该转换为从磁盘读取目录表的目录。

读 addTable() 的注释可知，这个方法用于向目录中添加 table，而 java 文件中没有现成的 Table 类，需要我们自己创建，注释给出了 Table 中的字段：

- file: 是要添加的表的内容，存放在给定的 DbFile 类中，同时告诉我们，file 中的 getID 方法是 Catalog 类中两个方法的参数
- name:table 的名字，可能是空值，并且要求是：如果名字冲突了要用新的，即检索到之前的 table，将其 remove 后再重新添加新 table
- pKeyField: 主键的名字

我们用一个 ArrayList 作为 Table 的容器，在实现 addTable() 的过程中，要注意删除的要求，还有一种特殊的情况，就是 name 可能为空，这时候比较名字就不好用了，应该比较 fileID，且在 Java 中 null.equals(null) 会报错，所以我们在判断的时候应该让 fileID 的比较在前，由于逻辑短路，就可以绕开这个报错，具体代码实现如下：

```
1 public void addTable(DbFile file, String name, String pkeyField) {
2   for(int i=0;i<tables.size();i++){
3     if(file.getId()==tables.get(i).file.getId()
4       ||name.equals(tables.get(i).name))
5       tables.remove(i);
6     }
7   tables.add(new Table(file,name,pkeyField));
8  }
```

剩下的都是简单的 getter 操作，不再赘述

3 exercise-3

Bufferpool 负责在内存中缓存最近从磁盘读取的页面，它由固定数量的页面组成 (构造函数中的 numPages)，BufferPool 最多应存储 numPages 个页面，在本实验中，若超过 numPages 个请求，就抛出 DbException

读 DEFAULT_PAGES 的注释可知：这是由其他类使用的，BufferPool 应该使用 numPages 属性来代替，因此需要我们定义一个 numPages 属性，并放到构造函数中

在本实验中，我们还需要实现一个 getPage 的功能：Page 类已经给出，它是一个接口，用来表示在线程池中的页面，通常，DbFiles 将从磁盘读取和写入页面。读 getPage 的注释可知：我们应该在缓冲池中查找检索到的页面。如果它存在，应将其返回，如果它不存在，它应该被添加到缓冲池并返回，因为 Page 是一个接口，不能被直接返回，HeapPage 实现了这个接口，我们可以定义 Page=new HeapPage()，通过向上转型达到要求；如果空间不足，在本实验中就抛出异常，则在本实验中，getPage 的实现如下：

```
public Page getPage(TransactionId tid, PageId pid, Permissions perm)
  throws TransactionAbortedException, DbException, IOException {
  for(int i=0;i<pages.size();i++)
    if(pages.get(i).getId()==pid)
      return pages.get(i);
    if(pages.size()==this.numPages)
      throw new DbException(null);
    Page temp=new HeapPage((HeapPageId)pid,null);
      return temp;
  }
```

4 exercise-4

4.1 HeapPagedId 和 RecordId

HeapPagedId 是 HeapPage 对象的唯一标识符，RecordId 是对特定表的特定页上的特定元组的引用，两个类中 equals 等函数和之前的构造方法基本一致，主要说明两个 hashcode 的构造：

1. 在 HeapPagedId 中，根据提示让我们参考 Bufferpool，所以我们应该调用 Bufferpool 中的 hashcode 方法，但我们不能在每一个 HeapPageId 中内置一个 Bufferpool，因为在任何时候，Bufferpool 总是唯一的 (单例)，因此我们借助单例对象 Database(Database 也永远只存在一个)，调用 get 方法，得到哈希函数如下：

```
public int hashCode() {
      return Database.getBufferPool().hashCode();
  }
```

2. 在 RecordId 中，根据要求，两个经 equals() 判断相等的对象应该具有相同的哈希值，而在 equals 方法中，我们是根据对象的 pid 和 tupleNo 进行比较的，因为 pid 为 PageId 型，不能直接参与 int 运算，读 PageId 的 Java 文件，我们看到 PageId 对象有两个返回 int 的方法，我们全部加以应用，得到哈希函数如下：

```
public int hashCode() {
  return this.tupleno+this.pid.getPageNumber()+this.pid.getTableId();
  }
```

4.2 HeapPage

HeapPage 的每个实例存储一页 HeapFiles 的数据，并实现 BufferPool 使用的 Page 接口。在这个类中首先有几个用于计算数量的函数，SimpleDB 将堆文件存储在磁盘上的格式与存储在内存中的格式大致相同。每个文件由磁盘上连续排列的页面数据组成。每页包含一个或多个表示标题的字节，后

跟实际页面内容的页面大小字节。每个元组的内容需要元组大小 *8 位，标头需要 1 位。因此，单个页面中可以容纳的元组数量为：

$$\_tuplesperpage = floor((\_pagesize * 8)/(\_tuplesize * 8 + 1))$$

其中 _tuple size 是页面中元组的大小 (以字节为单位)。这里的想法是每个元组都需要在标头中额外存储一位。我们计算页面中的位数 (通过将页面大小乘以 8)，并将此数量除以元组中的位数 (包括这个额外的标头位) 以获得每页中的元组数。floor 操作向下舍入到最接近的元组整数 (我们不想在页面上存储部分元组)

一旦我们知道每页的元组数，存储标题所需的字节数就是:

$$headerBytes = ceiling(tupsPerPage/8)$$

上限操作向上舍入到最接近的整数字节数 (我们永远不会存储少于一个完整字节的标头信息)，根据公式很容易写出这两个计算公式，代码不再赘述。

在本实验中，还有一个关键函数，用于判断槽 (slot) 的状态：每个字节的低位 (最低有效位) 表示文件中较早的槽的状态。因此，第一个字节的最低位表示页面中的第一个槽是否正在使用。这第一个字节的第二个最低位表示页面中的第二个槽是否正在使用，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槽不对应，因为槽的数量可能不是 8 的倍数。

```
public boolean isSlotUsed(int i) {
  int headerNum=i/8;
  int movement=i-i/8*8;
  byte array=(byte)(this.header[headerNum]>>movement&1);
  String str=new String(String.valueOf(array));
  if(str.equals("1"))
      return true;
  return false;
  }
```

首先，参数 i 为索引位置，i/8 计算该 page 中 header 的数量 (一个字节是 8 位)，即 headerNum，用于该类中 header 数组的索引，i/8*8 的作用是得到起始位索引，i 减去这个值就是要判断的那个位置，然后和 1 按位与操作取得该位的值，如果是 1 就代表被使用了。最后，还有一个迭代器函数，根据要求，此迭代器不应返回空槽中的元组，所以我们需要一个临时变量，记录不为空的槽，然后返回这个临时变量的迭代器：

```
public Iterator<Tuple> iterator() {
  ArrayList<Tuple> temp=new ArrayList<Tuple>();
  for(int i=0;i<this.tuples.length;i++) { //每个槽可以容纳一个元组
    if(this.isSlotUsed(i))
      temp.add(this.tuples[i]);
     }
   Iterator<Tuple> it=temp.iterator();
```

```
    return it;
  }
```

5 exercise-5

HeapFile 是 DbFile 的一种实现，它以无特定顺序存储元组集合，元组存储在页面上，每个页面都是固定大小，文件只是这些页面的集合，HeapFile 与 HeapPage 密切配合，在这个文件中，首先有一个关键的函数：读取页面。

在读取页面函数中，传入的参数为页面的 id，每个页有固定的大小 4096byte，利用 pid 和页的大小我们可以算出偏移量：

int offset=pid.getPageNumber()*BufferPool.getPageSize();

在读取页时，传入的 pid 是随机的，这是因为数据库中的数据是以页为单位进行管理和存储的，而这些页的物理位置在磁盘上是随机的，因此读取页的顺序也是随机的。如果顺序查找的话会效率很低，因此我们需要随机访问文件内容，以便任意偏移处读取和写入页面，Java 的 RandomAccessFile 类是一个可以访问文件任意位置的类，它可以读取和写入文件，还可以在文件中进行查找和修改。我们以只读的方式打开文件，并开一个字节数组记录文件内容，最后我们通过 pid 和读取到的内容构造 HeapPage 对象并向上转型返回 (因为 Page 是一个接口，不能实例化)，代码如下：

```
public Page readPage(PageId pid)  {
  int size=BufferPool.getPageSize();
  int offset=pid.getPageNumber()*size;
  byte[] data=new byte[size];
  try {
  RandomAccessFile random=new RandomAccessFile(f,"r");
  random.seek(offset); //定位到该页的位置
  int bytes=random.read(data);
  if(bytes==-1)
    return null; //没读到数据，返回空对象
  random.close();
  HeapPage page=new HeapPage((HeapPageId) pid, data);
  return page;
  }
  catch (FileNotFoundException e) {
 e.printStackTrace();
 } catch (IOException e) {
 e.printStackTrace();
}
 return null;
}
```

在本实验中,还需要实现一个返回迭代器的函数,要注意的是,迭代器必须使用 BufferPool.getPage() 方法来访问 HeapFile 中的页面。DbFileIterator 是一个所有 DbFile 必须实现的接口，因此要在返回函数中继承该类 (采用了匿名内部类的方式)，并通过向上转型返回，在接口中要实现几个抽象方法：打开迭代器 (open)，从运算符获取下一个元组 (Next)，是否到达末尾 (hasNext)，重置 (rewind) 和关闭 (close)，要在我们的内部类中定义几个字段来方便地实现我们的方法：

- boolean isOpen ：用于迭代器的打开和关闭
- Iterator<Tuple> it ：用于 next 操作
- int id ：记录读取到哪页，HeapPageId 构造函数的第二个参数
- BufferPool bufferpool=Database.getBufferPool() ：根据提示，要使用 BufferPool 中的方法，但 BufferPool 对象不是单例的,要保证数据一致性只能通过单例 Database 来获取同一个 BufferPool
- HeapPage page ：页对象

BufferPool.getPage() 方法需要三个参数：事务编号 tid，PageId(通过 HeapPageId 向上转型，传入第一个参数为 getId()，第二个参数为 pgNo，是我们定义的 id 字段) 和访问权限 (只读即可)。返回一个 HeapPage，迭代器访问页面时，不能简单的直接使用这个函数，我们还需要做设置每个页面的迭代器 (数据库中每个页的迭代器是不同的，因为元组的性质不同) 和判断拿到的 HeapPage 是否为空等工作，因此在匿名内部类中，添加如下的 getPage() 方法：

```
private boolean getPage(int id) throws DbException, TransactionAbortedException {
  if(!isOpen)
    throw new DbException("closed");
  page=(HeapPage) bufferpool.getPage(tid,new
  HeapPageId(getId(),id),Permissions.READ_ONLY);
  if(page==null)
    return false;
  it=page.iterator();
    return true;
}
```

在迭代器中，开关和重置状态函数比较容易实现，下面介绍 next 函数的实现方法：要判断迭代器是否还有下一个元素，要考虑几件事情：迭代器是否处于打开状态，迭代器是否为空，迭代器是否到了这个页的末尾，这个页是否是最后一页等

```
public boolean hasNext() throws DbException, TransactionAbortedException {
if(!isOpen||(id==numPages()&&!it.hasNext()))
  return false;
if(it!=null&it.hasNext())
  return true;
else {
  getPage(id++);
```

```
8   return it.hasNext();
9   }
10 }
```

6 exercise-6

Operators 负责查询计划的实际执行，它们实现了关系代数的运算，每个运算符都实现 DbFileIterator 接口。OpIterator 是所有 SimpleDB 运算符都应该实现的迭代器接口，如果迭代器未打开，则所有方法都不应起作用，并且应抛出 IllegalStateException。本实验中的 SeqScan 是一种顺序扫描访问方法的实现，它以无特定顺序读取表的每个元组，在此类中，除了三个字段之外，还要有一个 DbFileIterator 的接口，在构造函数中利用 tableid(表的序号) 和 tid(事务序号) 创建，tableid 的用处是在目录 (Catalog) 寻找指定文件，因为目录是唯一的，要使用单例 Database 来寻找，然后把迭代器挂在找到的这个文件上：

```
1 private DbFileIterator it;
2 it=Database.getCatalog().getDatabaseFile(tableid).iterator(tid);
```

然后是 getTupleDesc() 方法，该方法返回带有来自底层 HeapFile 的字段名称 TupleDesc，前缀为构造函数中的 tableAlias 字符串，别名和名称应该用 “.” 分隔，根据上述要求，编写代码如下：

```
1  public TupleDesc getTupleDesc() {
2    TupleDesc oldtTupleDesc = Database.getCatalog().getTupleDesc(tableid);
3    Type[] fieldTypes = new Type[oldtTupleDesc.numFields()];
4    String[] fieldWithPrefix = new String[oldtTupleDesc.numFields()];
5    for (int i = 0; i < fieldWithPrefix.length; i++) {
6      fieldTypes[i] = oldtTupleDesc.getFieldType(i);
7      String prefix = (tableAlias == null || tableAlias == "") ? "null" : tableAlias;
8      String oldfieldName = oldtTupleDesc.getFieldName(i);
9      String fieldName = (oldfieldName == null || oldfieldName == "") ? "null" :
10     oldtTupleDesc.getFieldName(i);
11     fieldWithPrefix[i] = prefix + "." + fieldName;
12   }
13   return new TupleDesc(fieldTypes, fieldWithPrefix);
14 }
```

7 Commit

图 7.1: Commit History