南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

数据库系统实验报告

SimpleDB Lab1

王茂增

年级：2021级

专业：计算机科学与技术

指导教师：袁晓洁

2023 年 3 月 20 日

目录

一、 实验简介

SimpleDB是MIT数据库系统课程的一个作业，其中包含了一系列未实现的类和接口，而我们的主要任就是完善这些未完成的部分，搭建一个基于Java的关系数据库管理系统。通过完成SimpleDB作业，我们可以深入了解关系型数据库管理系统的内部机制和原理，掌握事务处理、查询优化、数据存储和索引技术等方面的知识。

本次实验的主要内容是完成Lab 1代码的编写,并通过单元测试来验证代码的正确性。Lab 1主要实现访问磁盘数据所需的核心模块，如Tuple，Catalog，HeapFile以及SeqScan操作等，具体结构如图1所示。下面列出了每一个Exercise实现的主要功能，具体的思路以及代码实现将在实验详解中介绍。

- **Exercise 1：**实现元组的属性行以及元组。
- **Exercise 2：**实现一个数据库的目录，包含了数据库现有的表信息。需要实现添加新表以及提取信息的功能。
- **Exercise 3：**实现BufferPool，负责将内存最近读过的物理页缓存下来。
- **Exercise 4：**实现pageId、记录Id以及page的封装。
- **Exercise 5：**实现磁盘文件接口，通过该接口实现对磁盘的写入操作
- **Exercise 6：**实现扫描功能，即SELECT * FROM table。

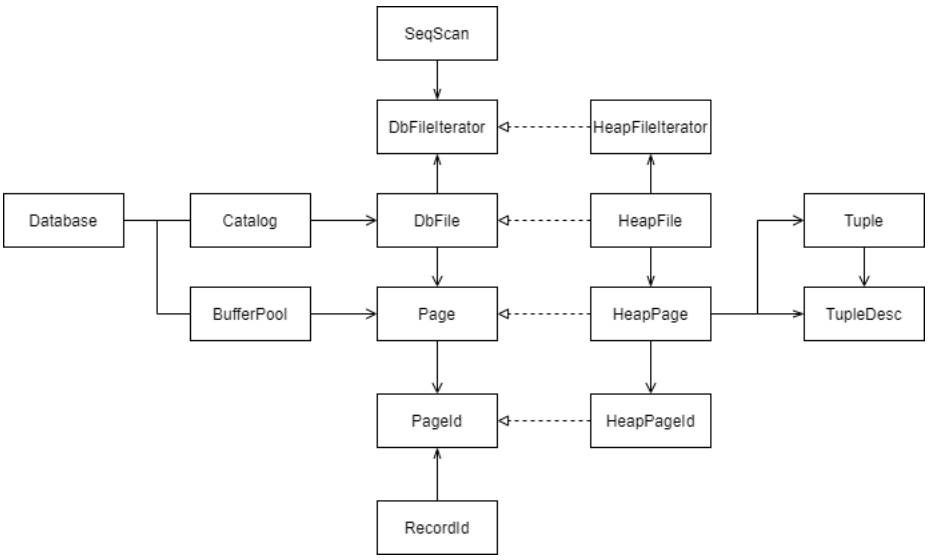

图 1: Lab1结构图

二、 实验详解

（一） Exercise 1

1. 设计思路

在本Exercise中主要实现了TupleDesc类和Tuple类。Tuple可以形象地理解为数据表格中的一行，而TupleDesc定义了表格中每列的属性，即字段。

TupleDesc由一个TDItem类型数组和一个描述字段数量的int型变量组成。其中，一个TDItem对象由一个Type类型变量和一个String类型变量组成，代表字段的类型和名字。其中,Type为一个枚举类型，包括INT_TYPE和STRING_TYPE两类。在实现的过程中，我使用List<TDItem>来存储TDItem，相比于数组的优势在于**可以比较方便地拓展字段**。

Tuple由TupleDesc、Field对象数组以及RecordID构成。TupleDesc指定了Tuple的模式，Field数组储存了该Tuple各字段的值，RecordID指定了元组在磁盘中的位置。

2. 重难点

1. TupleDesc(Type[] typeAr, String[] fieldAr)构造方法

为实现该构造方法，首先要检测输入的typrAr长度是否大于零以及是否与fieldAr的长度相等，如果不是，则抛出IllegalArgumentException异常。之后，根据输入的数组构建TDItem并加入该列表中，完成类的构造。主体代码如下：

双参数构造方法

```
if (typeAr.length == 0) {
        throw new IllegalArgumentException();
}

//The length of fieldAr must equal to the length of typeAr
if (typeAr.length != fieldAr.length) {
        throw new IllegalArgumentException();
}

fildsNum = typeAr.length;
tdAr = new ArrayList<>();

for (int i = 0; i < fildsNum; i++) {
        tdAr.add(new TDItem(typeAr[i], fieldAr[i]));
}
```

实现该构造方法之后，TupleDesc(Type[] typeAr)构造方法就可以方便的通过将fieldAr设置为空字符串数组然后调用上述构造方法实现，即this(typeAr, new String[typeAr.length])。

2. TupleDesc merge(TupleDesc td1, TupleDesc td2)方法

该方法的主要功能是将两个TupleDesc合并。为了实现上述功能，我们首先需要将两个TupleDesc的tdAr合并，再通过一个以tdAr为输入的构造方法生成一个新的TupleDesc。需要注意的是，由于List.addAll实现的是**浅拷贝**，所以需要在构造方法中实现**深拷贝**。该构造方法在原程序中并未给出，将在后续的改动部分介绍。merge()主体代码如下：

merge 方法

```
List<TDItem> newTdAr = new ArrayList<>();
newTdAr.addAll(td1.tdAr);
newTdAr.addAll(td2.tdAr);
return new TupleDesc(newTdAr);
```

3. **Iterator<TDItem> iterator()迭代器**

该迭代器可用于遍历TupleDesc中的TDItem。主要通过检测下一个下标是否达到或超过numFields来判断是否越界，若越界则需要抛出异常NoSuchElementException，若没有越界则返回下一个元素并将迭代器中的下标加一。主要代码如下：

TDItem迭代器

```
if (!hasNext()) {
        throw new NoSuchElementException();
}
return tdAr.get(cur++);
```

4. **int getSize()方法**

该方法返回由该TupleDesc所代表的Tuple需要占用的字节数，即**每一个fieldType需要的字节数之和**。主要代码如下：

getSize 方法

```
for (TDItem item : tdAr) {
        tdSize += item.fieldType.getLen();
}
```

5. **String toString()方法**

该方法的作用是将TupleDesc转化为特定格式的字符串，以描述其结构。其实现主要利用了**StringBuilder**类，通过调用TDItem中已经实现的toString方法并加上逗号，然后删去最后一个逗号并转化为String类型，得到最终结果。主要代码如下：

toString 方法

```
StringBuilder result = new StringBuilder();
for (TDItem item : tdAr) {
        result.append(item.toString()).append(",");
}
result.deleteCharAt(result.length() - 1);
return result.toString();
```

3. 改动部分

1. **TupleDesc(List<TDItem> tdAr_param)构造方法**

该构造方法通过将tdAr作为参数进行构造。增加该方法是为了方便merge方法以及resetTupleDesc方法的实现。该方法的实现与双参数构造方法类似，不过要注意的一点是需要实现**深拷贝**而不是浅拷贝。主要代码如下：

以tdAr为参数的构造方法

```
if (tdAr_param.size() == 0) {
        throw new IllegalArgumentException();
}

fildsNum = tdAr_param.size();
tdAr = new ArrayList<>();

//deep copy
for (TDItem item : tdAr_param) {
        tdAr.add(new TDItem(item.fieldType, item.fieldName));
}
```

2. **List<TDItem> getTdAr()方法**

该方法会返回该TupleDesc的tdAr，能方便Tuple类中resetTupleDesc方法的实现。由于代码比较简单，故在这里不再给出。

（二） Exercise 2

1. 设计思路

在Exercise 2中需要实现Catalog类。Catalog意为目录，一个数据库只有一个，管理着数据库中所有的表，每张表又分别对应着一个DbFile。DbFile为数据库磁盘文件的接口，储存着表中的所有信息。可通过调用Database.getCatalog()方法获取Catalog。

Catalog中存储着多个表，tableId可以唯一标识每一个表。在本次实验中，我将Table对应文件的绝对路径进行哈希方法的结果作为tableId。除tableId之外，每个表还有DbFile, pkeyField和Name成员变量，为了存储这些属性，我利用**线程安全**的**ConcurrentHashMap**来安全地存储tableId到这些属性的**映射关系**。同时，我还提供了name到TableId的哈希表name2id，可以提高getTableId()方法的效率，并且可以更简便地发现addTable()时产生的命名冲突。

2. 重难点

1. **void addTable(DbFile file, String name, String pkeyField)方法**

该方法主要用于往数据库中添加表结构。在实现的过程中，首先检测输入参数是否为null，如果是，则抛出IllegalArgumentException异常。之后，检测具有该名称的表是否已经加入，如果加入，则先删除原来的表，然后加入新表。主要代码如下：

addTable方法

```
if (file == null || name == null || pkeyField == null) {
        throw new IllegalArgumentException();
}
int tableId = file.getId();

//if conflicted,delete the original
if (name2id.containsKey(name)) {
        int id = name2id.get(name);
        id2dbFile.remove(id);
        id2name.remove(id);
        id2pkey.remove(id);
        name2id.remove(name);
}

id2dbFile.put(tableId, file);
id2name.put(tableId, name);
id2pkey.put(tableId, pkeyField);
name2id.put(name, tableId);
```

实现该方法之后，另外的参数较少的构造方法可直接通过给缺少的参数赋一个特殊的值来方便地进行构建。

（三） Exercise 3

1. 设计思路

该Exercise需要实现BufferPool类中的getPage()方法。BufferPool负责将内存最近读取过的物理页缓存下来。在数据库中，所有的读写操作都是先通过BuggerPool完成的。每个数据库只有一个BufferPool，可通过Database.getCatalog()方法获取。其中的成员变量NUM_PAGES表示Buffer_Pool能缓存的最大页数，如果当前页数超过最大页数，则需要相应的驱逐机制，这将在后续实验中实现。

在实现的过程中，我仍然利用**ConcurrentHashMap**存储下了pid到Page、tip和perm的映射关系。根据实验手册所说，这次实验还不需要实现锁定等操作，所以我并没有在getPage时上锁。

2. 重难点

1. **Page getPage(TransactionId tid, PageId pid, Permissions perm)方法**

   该方法主要用于读取页面。在实现该方法的过程中，首先检测BufferPool中是否含有该pid所对应的页面，如果有，则直接返回。如果没有，则利用HeapFile类中的ReadPage方法读取页面，并检测当前页面数量是否超出最大页面数，若是，则抛出DbException异常。若没超过，则将新页面加入BufferPool中。

getPage方法

```
if(pid2page.containsKey(pid)){
        return pid2page.get(pid);
}

Page newPage = Database.getCatalog().getDatabaseFile(pid.getTableId()).
    readPage(pid);
if(pid2page.size()>NUM_PAGES){
        throw new DbException("The bufferpool is full!");
}
pid2page.put(pid,newPage);
pid2perm.put(pid,perm);
pid2tid.put(pid,tid);

return newPage;
```

（四） Exercise 4

1. 设计思路

在Exercise 4中需要实现HeapPageId、RecordId和HeapPage三个类。HeapPageId是PageId的实现，储存了Page所在Table的tableId，以及自身在HeapPage中的序号pgNo。RecordId与Tuple绑定，储存了Tuple所在Page的PageId，以及自身在Page中的序号tupleNo。HeapPage提供了硬盘读写数据的方式。一个HeapPage包含多个slot，而一个slot存储着一个Tuple。除了slots之外，每个HeapPage都包含一个headers，其中的每一位都指示着一个slot是否使用（headers的最后一个元素可能有些位没有对应的slot）。

2. 重难点

1. **int getNumTuples()方法**

   该方法首先检测是否已经计算过numSlots，如果是，则直接返回。如果还没计算过，则通过公式$(\boldsymbol{PageSize}*8)/(\boldsymbol{TupleSize}*8+1)$计算得出一个Page能存储的slot数量。

getNumTuples方法

```
if (numSlots != 0) {
        return numSlots;
}
return (BufferPool.getPageSize() * 8) / (td.getSize() * 8 + 1);
```

2. **int getHeaderSize()方法**

   该方法通过公式$\lceil \boldsymbol{tupleNum}/8 \rceil$计算得出一个Page需要的hears大小。

getHeaderSize方法

```
return (int) Math.ceil(getNumTuples() / 8.0);
```

3. **boolean isSlotUsed(int i)方法**

在headers中，每个slot对应的位按照大端序排列，即headers中每一元素的低位对应着靠前位置的slot。为实现该方法，首先将访问header[i / 8]元素，然后将1向左移动(i%8)位，此时只有从右往左数第k位的值为1。将两者相与，如果结果不等于0，则可得到第i个slot对应的headers为1，说明第i个slot已经使用过了。主要代码如下：

isSlotUsed方法

```
return (header[i / 8] & (1 << (i % 8))) != 0;
```

4. **Iterator<Tuple> iterator()**

为了实现该方法，通过numSlots - getNumEmptySlots()算出使用过的slot的数量，然后遍历slot，每次都要判断是否使用以及是否越界，如果未使用，则返回该Tuple，并将当前已返回的Tuple数目加一。下面给出其中next()方法的实现代码：

迭代器

```
if (!hasNext()) {
        throw new NoSuchElementException();
}
while (!isSlotUsed(++slotsCur)) ;
usedSlotsCur++;
return tuples[slotsCur];
```

（五） Exercise 5

1. 实现思路

在Exercise 5中需要实现HeapFile类。HeapFile类实现了DbFile的接口，BufferPool正是通过DbFile来读取Page的。该类包含TupleDesc、file、numPage三个成员变量，分别存储文件的模式、文件的地址以及页的数目。其中，页的数目可以通过将file的总字节数除以页的大小得到。

2. 重难点

1. **Page readPage(PageId pid)方法**

为了实现该方法，我们需要使用RandomAccessFile打开文件进行真正的读取，而不是通过BufferPool中的getPage方法，因为其实BufferPool中的getPage方法中调用了HeapFile中的readPage，两者不能循环调用。然后，我们通过公式$\boldsymbol{pageId * pageSize}$得到该页在文件中的起始位置，然后读取pageSize字节的数据生成Page。主要代码如下：

readPage方法

```
Page page = null;
byte[] data = new byte[BufferPool.getPageSize()];

try (RandomAccessFile raf = new RandomAccessFile(getFile(), "r")) {
        int pos = pid.getPageNumber() * BufferPool.getPageSize();
        raf.seek(pos);
        raf.read(data, 0, BufferPool.getPageSize());
        page = new HeapPage((HeapPageId) pid, data);
} catch (IOException e) {
        e.printStackTrace();
}

return page;
```

2. **DbFileIterator iterator(TransactionId tid)方法**

该方法通过迭代器遍历该HeapFile对应Table的所有元组。为实现该方法主要需要实现hasNext()方法，当tupleIterator读完一个页面时可以跳到下一个页面，然后就可以继续调用tupleIterator.next()方法，直到将最后一个页面读取完。在hasNext()方法的实现过程中，首先要判断tupleIterator是否为null，如果为null则返回false。如果不为null，则调用tupleIterator.hasNext()方法，如果为真，则返回true。如果读取完当前页面后还有页面没有读取，则通过BufferPool中的getPage方法得到下一页的tupleIterator，然后通过返回方法tupleIterator.hasNext()的结果判断是否读取完毕。如果在前面都没返回，说明现在已经读取到了最后一个页面的最后一个Tuple，此时直接返回false。next()方法代码如下：

迭代器

```
if (tupleIterator == null) {
        return false;
}

if (tupleIterator.hasNext()) {
        return true;
}

if (++pagePos < numPages()) {
        HeapPageId pid = new HeapPageId(getId(), pagePos);
        tupleIterator = ((HeapPage) Database.getBufferPool()
        .getPage(tid, pid, Permissions.READ_ONLY)).iterator();
        return tupleIterator.hasNext();
}

return false;
```

（六） Exercise 6

1. 实现思路

在Exercise 6中，需要实现SeqScan类。该类具有成员变量tid、tableAlias、tableId以及TupleIterator。tableAlias是表的别名；tableId是表的ID，指定了要扫描的表；TupleIterator是Tuple的迭代器，将通过它完成扫描表的操作。

2. 重难点

1. **Page readPage(PageId pid)方法**

该方法用于返回一个在输入的tableId所对应的tupleDesc的fieldName中加上alias前缀的TupleDesc，同时要注意进行**深拷贝**。实现时首先检测是否已经存在这个TupleDesc，如果存在则直接返回。如果不存在，则首先得到tableId对应的TupleDesc，然后将fieldType复制，将fieldName加上alias.前缀。为了防止alias或fieldName为空，需要特殊处理一下alias以及fieldName为null的情况，然后将结果拼接成新的fieldName并生成新的TupleDesc。

readPage方法

```
if (td != null) {
        return td;
}

TupleDesc desc = Database.getCatalog().getTupleDesc(tableId);
int fieldNum = desc.numFields();
Type[] types = new Type[fieldNum];
String[] names = new String[fieldNum];
for (int i = 0; i < fieldNum; i++) {
        types[i] = desc.getFieldType(i);
        String prefix = (getAlias() == null ? "null." : getAlias() + ".");
        String fieldName = desc.getFieldName(i);
        fieldName = (fieldName == null ? "null" : fieldName);
        names[i] = prefix + fieldName;
}

return new TupleDesc(types, names);
```

2. **Tuple next()方法**

该方法返回下一个该Table的下一个Tuple。直接调用TupleIterator.next()方法即可，不过要注意对其进行**深拷贝**并返回，代码较为简单，故不再给出。

三、 总结

本次实验中，我完成了SimpleDB的Lab 1部分，迈出了搭建一个基于Java的关系型数据库管理系统的第一步。本次实验不仅提高了我对Java语言的掌握程度，还让我深入理解和直观体会到了关系型数据库管理系统访问磁盘数据的过程和原理：

首先通过Catalog的addTable方法将磁盘中的文件作为Table加入数据库中，此时只是加入了文件的路径，并没有读取数据。如果要访问某个Page中的数据，则会首先在BufferPool寻找这个Page，如果找到则可在HeapPage中通过迭代器等方法访问数据；如果没有在BufferPool中找到，则HeapFile会通过RandomAccessFile类将指定Page中的数据读入BufferPool，然后再进行访问。

四、 提交记录

```
Author: mzwangg <mzwangg@gmail.com>
Date:   Sun Mar 19 23:11:33 2023 +0800

    Lab_1 Update

commit a0158f6ccf8ee35a0d3312e2c2a3a1da9bc3207e
Author: mzwangg <mzwangg@gmail.com>
Date:   Sat Mar 18 23:07:13 2023 +0800

    Lab_1 Exercise_6

commit f90bc9241045e7c88bb14b3072e044fba579b507
Author: mzwangg <mzwangg@gmail.com>
Date:   Fri Mar 17 23:13:43 2023 +0800

    Lab_1 Exercise_5

commit 9012c331aeefbc7e1c3e309816c8d96e9e60be3b
Author: mzwangg <mzwangg@gmail.com>
Date:   Thu Mar 16 11:28:03 2023 +0800

    Lab_1 Exercise_3、4

commit 101e26f9d0b5f1ecdb1eca384ff5eb07af90cfd8
Author: mzwangg <mzwangg@gmail.com>
Date:   Tue Mar 14 18:30:59 2023 +0800

    Lab_1 Exercise_2

commit 05f20c41896477cc20e88f861961484304 64e64a
Author: mzwangg <mzwangg@gmail.com>
Date:   Tue Mar 14 14:13:04 2023 +0800

    Lab_1 Exercise_1
```

图 2: 提交记录

五、 源代码

https://gitlab.com/mzwangg/SimpleDB