

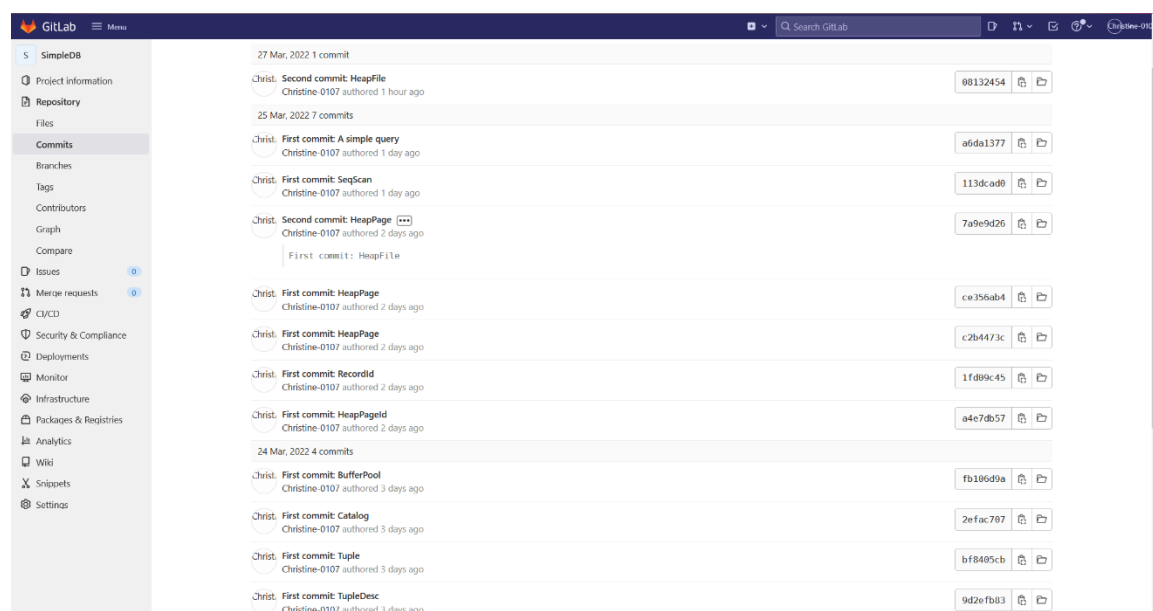
SimpleDB Lab1 实验报告

2010239 李思凡

一、Lab1 概述

通过 Lab1，将实现对硬盘上存储数据的访问。在本次实验中，需要实现管理元组的类 (TupleDesc, Tuple)；实现目录类 (Catalog)；实现缓冲池 (BufferPool)；实现访问数据的类 (HeapPageId, RecordId, HeapPage, HeapFile)；以及实现一个操作类 (SeqScan)。

二、Git Commit History



三、字段和元组 (Fields and Tuples)

(一) TupleDesc 类

1. 设计思路

TupleDesc 类主要描述这个表中的元组应该遵循的模式，即应该有的字段类型和名称。通过辅助类 TDIItem 来定义，一个 TDIItem 对象包括 fieldType 和 fieldName（字段类型和字段名）两个属性。

(1) 首先建立一个元素类型为 TDIItem 的动态数组 tdItems，向其中添加所有的 TDIItem 对象；对于 iterator() 方法，返回动态数组 tdItems 的 iterator()。

(2) 在 TupleDesc 的构造函数里，只需将利用所给 type 和 name 创建 TDIItem

对象，并把所有该对象添加到 `tdItems` 中。

(3) 通过调用动态数组的 `size()`，可获得字段的数量。

(4) 通过遍历 `tdItems`，可以获得每个索引对应元素的类型和名称；

(5) 通过遍历 `tdItems`，可以将名称与给定的 `name` 做比较，若相等则返回对应的索引标号；

(6) 可以遍历得到每个类型的字节数，相加得到总的字节大小；

(7) 也可以遍历并对每个对象调用 `toString()` 方法，按照规定格式拼接成描述 `TupleDesc` 类的字符串。

(8) `merge()` 方法和 `equals()` 方法设计思路见重难点。

2. 重难点

(1) 合并两个 `TupleDesc`。（`merge()` 方法）

开辟长度为两个 `TupleDesc` 长度之和的数组 `type` 和 `name`，先遍历第一个 `TupleDesc`，获取它的类型和名称，填入 `type` 和 `name` 的前面，再遍历第二个 `TupleDesc` 的类型和名称，填入后面。再利用 `type` 和 `name` 数组创建新的 `TupleDesc` 对象并返回。

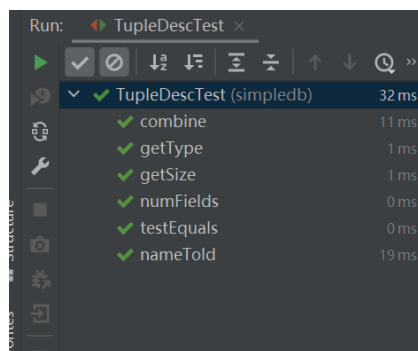
(2) 比较给出的对象与该 `TupleDesc` 是否相等。（`equals()` 方法）

首先判断给出的对象 `obj` 是否为空、是否也为 `TupleDesc` 类型，若为空或不是同一类型则不相等。再比较 `obj` 中字段数量和该 `TupleDesc` 中字段数量是否相等，若数量不等也不相等。最后遍历所有字段，依次比较类型和名称是否相等，若全部相等则两个对象相等。

对其余类对象的比较同理。

3. 改动部分：未改动 API 和测试代码。

4. test 测试结果如下：



（二）Tuple 类

1.设计思路

Tuple 类保存表中所有元组的信息，遵循 TupleDesc 定义的模式。

（1）首先需要在 Tuple 类中添加所需属性：TupleDesc 类的对象 tupleDesc 用来表示所需遵循的模式，元素类型为 Field 的动态数组 tupleField 存各种字段的数据，以及 RecordId 类的对象 recordId 用来标识各记录。

（2）在 Tuple 类的构造函数中，根据传进来的参数初始化 tupleDesc，表示要遵循的模式，并根据该 tupleDesc 中字段的数量初始化动态数组 tupleFields 的大小，并将其中元素先初始化为空值。

（3）获取 TupleDesc 的 getTupleDesc()方法只需返回相应属性。

（4）获取 RecordId 的 getRecordId()方法只需返回相应属性。设置 RecordId 的 setRecordId()方法中，将类中属性 recordId 设置为传进的参数。

（5）通过 setField()方法改变元组中第 i 个字段，调用动态数组 tupleField 的 set(i,f)方法。获取第 i 个字段的值，调用 tupleField 的 get(i)方法并返回。

（6）通过 iterator()方法迭代元组中的所有字段，只需返回 tupleField 的 iterator 方法。

（7）遍历 tupleField 的每个元素获取其 toString 字符串，按照规定格式拼接成 Tuple 类的描述性字符串。

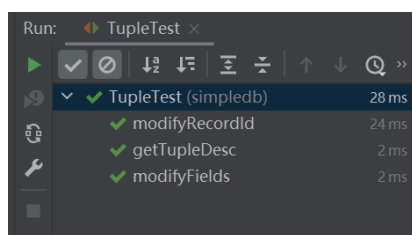
（8）对于 resetTupleDesc()方法，需要根据传进的参数重新初始化 tupleDesc 作为新的模式，并重置 tupleField 中所有元素为空。

2.重难点

Tuple 类的重点在于需要准确确定本类所需添加的属性，并通过构造函数进行初始化。确定后其他的函数就比较容易添加。

3.改动部分：未改动 API 和测试代码。

4.test 测试如下（之后的类完善了 RecordId，因此能通过 modifyRecordId 方法）



四、目录 (Catalog)

1. 设计思路

(1) Catalog 类中记录数据库中所有表和它们的模式。通过观察类中已给出方法的参数列表,可知向 Catalog 类中添加表需要三个参数: DbFile 对象 file (每个 DbFile 代表一个表,其中会为每个表设定一个独特的 ID),表的名称 name,每个表主键字段的名称 pkeyField。通过观察 Catalog 类中已添加函数的信息可知需要通过表的 ID 值定位表信息。因此可为 Catalog 类添加辅助类 Table,每个 Table 对象代表一张表,Table 类中包括三个私有属性: DbFile 对象 dbFile,表的名称 tableName,主键字段的名称 primaryKey。为 Table 类添加构造函数,获取三个私有属性的函数和 toString()函数。

(2) 为方便后续函数由表的 ID 找表、由表的名称找表的 ID,建立两个 ConcurrentHashMap,分别是<Integer,Table>的 hashTable 和<String,Integer>的 nameToId。在 Catalog 的构造函数中初始化这两个 map。

(3)在 addTable()方法中,通过传进的三个参数创建 Table 类对象 newTable;通过调用 file 对象的 getId()方法获取 tableId。调用 put 方法,向 hashTable 中添加键值对<tableId, newTable>,向 nameToId 中添加键值对<name, tableId>。

(4)在 getTableId()方法中,通过建立的 nameToId 可以方便的使用名称查找 ID,根据查找结果进行返回。若给出的 name 为空值,则抛出异常。

(5)通过给出的 tableId 和 hashTable,可以很方便的找到对应的 Table,在利用 Table 类中的获取方法,很容易获取相应的数据库文件、表名称和主键名称。

(6)在 getTupleDesc()方法中,通过给出的 tableId,调用 getDatabaseFile()即可获取相应的数据库文件,再调用 DbFile 类中的 getTupleDesc()方法即可获得该表对应的元组模式。

(7)在 tableIdIterator()方法中,通过调用 nameToId.values()获得键,即所有 tableId,再返回它的 iterator()。

(8) clear()方法需要将目录中所有表清空,只需调用 hashTable 和 nameToId 的 clear()方法。

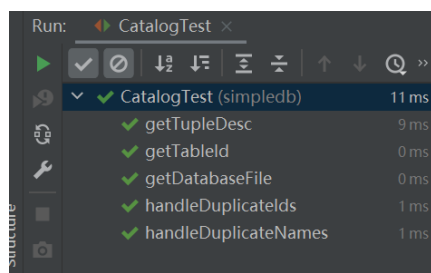
2. 重难点

本类的重难点在于需要通过理解 Catalog 类的内容和观察已有方法,创建

Table 类, 准确定义 Table 类中的三个属性; 并建立两个 HashMap: <Integer,Table> 的 hashTable 和<String,Integer>的 nameToId, 方便后续的查找操作。

3.改动部分: 未改动 API 和测试代码。

4.test 测试结果如下:



Method	Time
✓ CatalogTest (simplifiedb)	11 ms
✓ getTupleDesc	9 ms
✓ getTableId	0 ms
✓ getDatabaseFile	0 ms
✓ handleDuplicateIds	1 ms
✓ handleDuplicateNames	1 ms

五、缓冲池 (BufferPool)

1.设计思路

(1) 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 BufferPool 进行, 将提高访问速度。BufferPool 中 pageSize 字段固定了每页的大小。通过观察 BufferPool 类中给出的构造函数参数, 可知需增加一个属性 numPages 用来存缓冲池中最多容纳的页数。通过观察 getPage()方法可知, 要通过缓冲池获取页, 需要对应的 PageId, 因此可以添加一个 HashMap, 由 PageId 查找 Page, 称为 buffer。

(2) 在 BufferPool 的构造函数中, 根据传进来的参数初始化 numPages, 并初始化 buffer, 长度设为 numPages。

(3) getPage()方法设计思路见重难点。

2.重难点: getPage()方法

利用 buffer 和键 pid 定位 Page, 如果 buffer 中有该 PageId, 则返回它对应的 Page; 若 buffer 中还未包含这一页, 先判断当前 buffer 容纳的页数是否小于最多可容纳页数 numPages; 若小于, 则通过目录定位该数据库文件, 再调用 readPage(pid)方法获取该页, 将该 pageId 和 page 添加到 buffer 中, 并返回该页; 若 buffer 中空间不足, 则抛出异常。

3.改动部分: 未改动 API 和测试代码。

六、HeapFile 访问方法—HeapPage 实现

每一个 HeapFile 对象对应一张表，包含一组物理页。

（一）HeapPageId 类

1.设计思路

（1）HeapPageId 是为每个 HeapPage 设立的独特的标识。根据构造方法的参数列表可知，它包括 HeapPage 所属的 HeapFile 所对应的 tableId 和该页在表中的页数 pgNo。应添加属性 tableId 和 pgNo，并在构造方法中初始化。

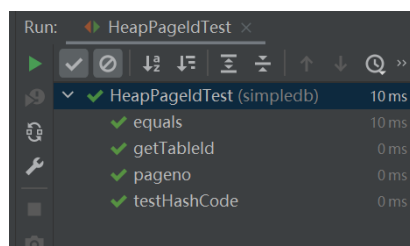
（2）getTableId()和 getPageNumber()方法只需返回相应的属性。

（3）hashCode()方法中需要为该页设置一个哈希码，将 tableId 和 pgNo 对应的字符串拼接，再返回该字符串对应的 hashCode()。

（4）比较给出的对象是否与该 HeapPageId 相等，整体思路与 TupleDesc 类中 equals()方法相同。

2.改动部分：未改动 API 和测试代码

3.test 测试结果如下：



（二）RecordId 类

1.设计思路

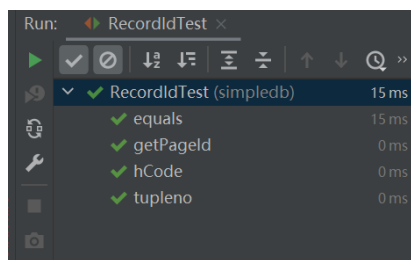
（1）RecordId 是对一张特定表的特定页的特定元组的编号。由构造方法的参数列表可知，它包括元组所在页的 PageId 和该元组在页中的元组数 tupleNo。因此需添加这两个属性，并通过构造方法进行初始化。

（2）getTupleNumber()和 getPageId()方法只需返回相应的属性。

（3）hashCode()方法和 equals()方法的整体思路与 HeapPageId 类中相同。

2.改动部分：未改动 API 和测试代码

3.test 测试结果如下：



(三) HeapPage 类

每个 HeapPage 实例存 HeapFile 中一页的数据，并实现 Page 接口。页的大小由 BufferPool.DEFAULT_PAGE_SIZE 定义。每页里有一些 slots，每个 slot 存一个表中的元组。每页还有一个 header，其中有每一位通过 bitmap 指示每个 tuple，1 代表有效，0 代表无效（被删了或从未初始化）。

1. 设计思路

(1) getNumTuples() 方法需要计算出该页中元组的个数。已知页的大小按位表示为 $\text{BufferPool.getPageSize()} * 8$ ，每个元组的大小按位表示为 $\text{tupleDesc.getSize()} * 8$ ，还需为每个元组留出 1 位给 header 作映射。因此该页中元组的个数为 $\text{tuples per page} = \text{floor}((\text{page size} * 8) / (\text{tuple size} * 8 + 1))$ ，代入即可。

(2) getHeaderSize() 方法需要计算出该页中 header 的字节数。计算方法为 $\text{header bytes} = \text{ceiling}(\text{tuples per page} / 8)$ ，调用函数代入 tuples per page 即可。

(3) getId() 方法只需返回属性 pid。

(4) getNumEmptySlots() 方法需要计算出该页中空的 slots 的个数。只需对所有的 slots 进行遍历，调用 isSlotUsed(i) 函数判断第 i 个是否填有元组，统计所有非空的个数。

(5) isSlotUsed() 方法和 iterator() 方法的设计思路见重难点。

2. 重难点

(1) 判断第 i 个 slot 是否填有元组 (isSlotUsed() 方法)

第 i 个 slot 前 header 中应该已经填满 header 中 $i/8$ 个字节，令 $\text{pre} = i/8$ ；第 i 个 slot 应该对应 header 中第 $(i/8 + 1)$ 个字节的 $i \% 8$ 位，令 $\text{remain} = i \% 8$ 。slot 对应的位为 $\text{header}[\text{pre}] \gg \text{remain}$ ，判断它是否为 1，若为 1 则填有元组。

(2) 获取所有元组的 iterator() 方法

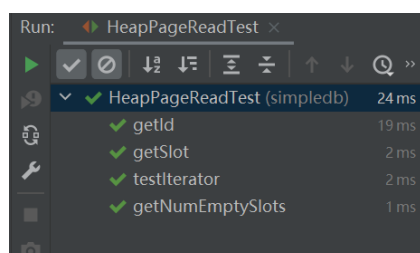
可以创建一个辅助类 HeapPageTupleIterator，实现 Iterator 接口。在其中添加

元素类型为 Tuple 的迭代器 tupleIterator 和动态数组 tupleArrayList。在构造方法中遍历所有 slots，向 tupleArrayList 中添加对应的元组，并将它调用 iterator() 方法的返回值赋给 tupleIterator。重写 hasNext() 和 next() 方法，返回 tupleIterator 的 hasNext() 和 next()。重写 remove() 方法，按要求抛出异常。

在 HeapPage 类中的 iterator 方法内，创建 HeapPageTupleIterator 类对象并返回即可。

3. 改动部分：未改动 API 和测试代码

4. test 测试结果如下：



七、HeapFile 访问方法—HeapFile 类实现

1. 设计思路

(1) HeapFile 类实现 DbFile 接口，由存元组的页组成，每个 HeapFile 实例对应一张表。根据 HeapFile 的构造方法的参数列表可知，需要添加属性 File 类对象 file 和 TupleDesc 类对象 tupleDesc，并通过构造函数初始化。两个属性值通过相应的 get 方法返回获取。

(2) getId() 方法需要为每个 HeapFile 实例生成一个独特的 ID 值（也是对应的表的 tableId）。可以用 HeapFile 的绝对路径生成 HashCode 作为 ID。

(3) numPages() 方法需要计算该 HeapFile 中有多少页。文件的大小为 file.length()，每页的固定大小由 BufferPool 中定义。因此 page number = floor(file length / page size)。

(4) readPage() 方法和 iterator() 方法的设计思路见重难点。

2. 重难点

(1) readPage() 方法，读取磁盘上某一特定页。

由 HeapPage 类的构造方法可知，要创建一个 HeapPage 对象，需要相应的 HeapPageId 对象和一个向其中写入该页数据的字节数组；由 HeapPageId 类的构

造方法可知，需要相应的 `tableId` 和 `pgNo` 来创建一个 `HeapPageId` 对象。

通过参数 `pid`，调用两个 `get` 方法，可以得到相应的 `tableId` 和 `pgNo`。调用 `HeapPage` 类中的静态方法 `createEmptyPageData`，可以得到一个空页对应的字节数组 `bytes`。通过文件输入流读取该 `HeapFile` 文件，跳过前 `pgNo*pageSize` 个字节，定位到该页对应的第一个字节。从当前位置开始，读取 `pageSize` 个字节，将数据存到 `bytes` 数组中。利用得到的 `tableId`，`pgNo` 和 `bytes` 创建一个 `HeapPage` 对象，返回该对象。

若读取过程中发现文件不存在，抛出异常。

(2) 获取该 `HeapFile` 上所有元组的 `iterator()` 方法。

创建辅助类 `HeapFileIterator`，实现接口 `DbFileIterator`。迭代器需要通过 `BufferPool` 获取各页，需要 `transactionId`，`HeapPageId`。因此需要属性 `HeapFile` 对象 `heapFile`，`TransactionId` 对象 `tid`，开始迭代时的页数 `openPgNo`，以及元素类型为 `Tuple` 的迭代器 `tupleIterator`。通过构造方法进行初始化。

需要编写方法获取根据当前 `heapFile` 和 `openPgNo` 获取元组的迭代器 `tupleIterator`，添加方法 `getTupleIterator(int pgNo)`。通过 `heapFile` 的 `getId()` 方法生成 `tableId`，与 `pgNo` 一起构成 `HeapPageId` 对象 `pid`。通过 `Database.getBufferPool().getPage()` 方法，经过缓冲池获得 `HeapPage` 对象 `heapPage`。`heapPage` 调用在 `HeapPage` 类中添加好的 `iterator()` 方法即可返回该页对应的 `tupleIterator`。

接着需要实现接口 `DbFileIterator` 定义的几个方法。`open()` 方法需要打开此迭代器，设置开始迭代时的页数 `openPgNo` 为 0，调用 `getTupleIterator()` 方法，传进 `openPgNo` 即可。

对于 `hasNext()` 方法，首先判断 `openPgNo` 是否为空，若不为空，循环执行直到所有页都被迭代器访问过。当 `tupleIterator.hasNext()` 返回为真时，表示迭代器里还有元组，返回真；反之，则使 `openPgNo` 加 1，调用 `getTupleIterator()` 方法向 `tupleIterator` 中添加下一页的元组。对于 `next()` 方法，若 `hasNext()` 为真，则返回 `tupleIterator` 的 `next()` 方法；否则抛出异常。

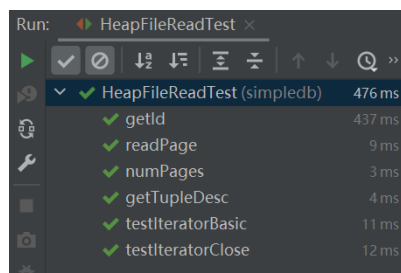
对于 `close()` 方法，只需将开启页数 `openPgNo` 和元组迭代器 `tupleIterator` 置空。`rewind()` 方法要求重置迭代器，只需先调用 `close()` 关闭，再调用 `open()` 打开。

迭代器。

完成辅助类后，对于 `HeapFile` 类中的 `iterator()` 方法，只需创建一个 `HeapFileIterator` 对象并返回即可。

3.改动部分：未改动 API 和测试代码

4.test 测试结果如下：



Method	Time (ms)
HeapFileReadTest (simplifiedb)	476 ms
getId	437 ms
readPage	9 ms
numPages	3 ms
getTupleDesc	4 ms
testIteratorBasic	11 ms
testIteratorClose	12 ms

八、操作符 Operators (SeqScan 类)

1.设计思路

(1) 数据库的操作符负责查询语句的具体执行，基于迭代器实现，每一个 `operator` 都实现了一个 `DbFileIterator` 接口。`SeqScan` 操作符顺序扫描特定表的页中的所有元组，需要用到在上一个类 `HeapFile` 中实现的 `iterator()` 方法。

(2) 根据 `SeqScan` 类的操作要求和构造方法的参数列表，可以确定需要添加的属性为 `TransactionId` 类的对象 `tid`，表的 `tableId`，表的别名 `tableAlias`，要访问的 `dbFile` 和迭代器 `dbFileIterator`。通过构造方法初始化。通过 `getAlias()` 方法获取 `tableAlias`。通过 `reset()` 函数进行重置。

(3) `getTupleDesc()` 方法需要返回该表对应的元组模式，并且需要按照要求在模式的 `fieldName` 域前加上前缀 `tableAlias`。通过 `dbFile` 的 `getTupleDesc()` 方法即可获取模式，为每个 `fieldName` 前加上 `tableAlias` 后，创建一个 `TupleDesc` 对象并返回。

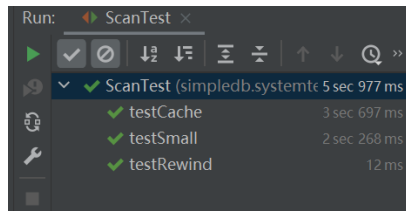
(4) `hasNext()`, `next()`, `close()`, `rewind()` 方法，只需通过 `dbFileIterator` 调用 `HeapFile` 类中实现的 `iterator()` 方法即可。

2.重难点

`SeqScan` 类的重点在于把握它的访问是通过 `HeapFile` 类中的迭代方法 `iterator()` 实现的，只需合理调用即可。

3.改动部分：未改动 API 和测试代码

4.test 测试结果如下：



九、一次简单的查询

通过创建 test 数据和 test 代码，通过命令行运行测试，得到查询结果。

```
D:\SimpleDB\SimpleDB>ant
Buildfile: D:\SimpleDB\SimpleDB\build.xml

compile:

dist:

BUILD SUCCESSFUL
Total time: 0 seconds

D:\SimpleDB\SimpleDB>java -classpath dist/simpledb.jar simpledb.test
1      1      1
2      2      2
```