

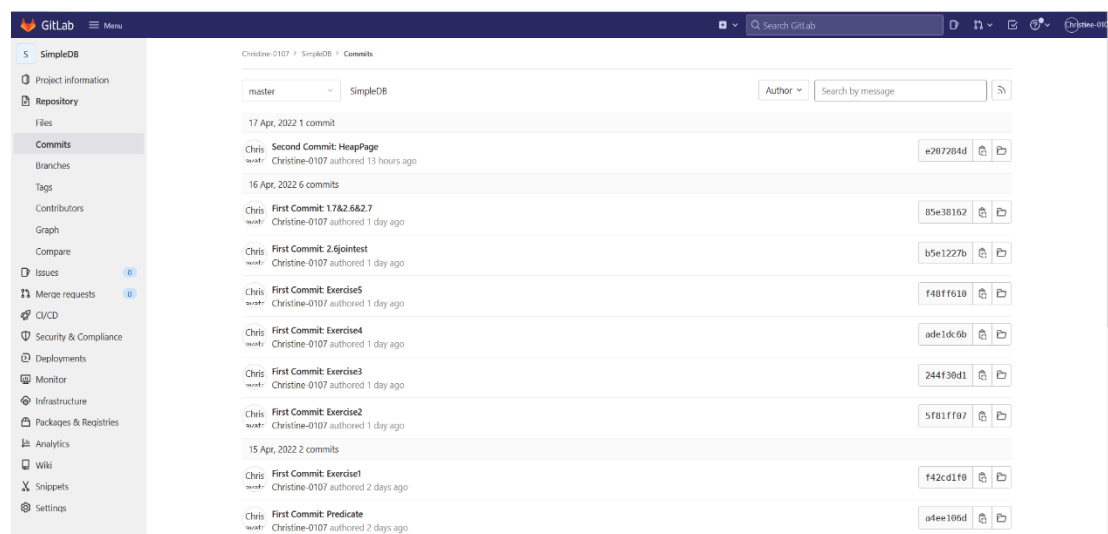
# SimpleDB Lab2 实验报告

2010239 李思凡

## 一、Lab2 概述

通过 Lab2，将实现一系列操作符从而完成对表中元组的插入和删除。在本次实验中，需要实现过滤符合条件的元组的类（Predicate、Filter）和能够连接符合条件的元组的类（JoinPredicate、Join）；实现聚合操作的类（IntegerAggregator、StringAggregator、Aggregator）；实现修改表的方法，从单个页面和一个表文件的角度完成插入删除元组后表的修改（HeapPage、HeapFile），还需要相应对缓冲池进行操作（BufferPool）；基于此完成实现插入和删除操作符完成操作（Insert、Delete）；最后需要完成当 BufferPool 中页面满后的页面置换。

## 二、Git Commit History



## 三、过滤和连接（Filter and Join）

### （一）Predicate 类

#### 1. 设计思路：

Predicate 类是 Filter 类的辅助类，主要用于筛选 tuple，将 tuple 中的字段和指定字段比较，比较逻辑包括 “=, >, <, >=, <=, LIKE, <>”，筛选出满足指定条件的元组。

（1）根据 Predicate 类的要求可知，需要添加属性，比较的字段字段号 field、

比较运算符 `op`、比较的内容 `operand`。

(2) 通过构造函数为属性赋初值，并为三个属性编写 `get()` 方法。

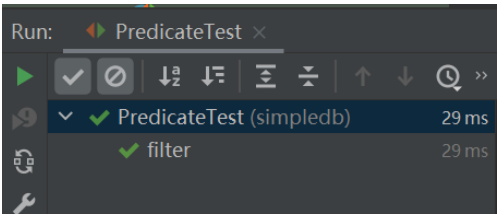
(3) 编写判断元组是否满足条件的 `filter()` 方法。

2. 重难点：

通过 `filter()` 方法完成比较。通过元组 `t` 的 `getField` 方法找到字段编号 `field` 的字段对应的内容。调用 `Field` 类中定义的 `compare(op,operand)` 方法，将 `t` 的该字段内容与 `operand` 用 `op` 比较，满足则返回真值。

3. 改动部分：未改动 API 和测试代码。

4. test 测试结果如下：



## (二) Filter 类

1. 设计思路：

`Filter` 类根据 `Predicate` 的判断结果，实现对于满足条件的元组的操作，实现 `Operator` 接口。实现过程如下图所示。`Filter` 类根据给出的一些元组和指定谓词 `Predicate`，过滤出满足条件的元组。

Predicate p:

field	op	operand
2	>	15

tuples:

编号	姓名	年龄
01	AA	15
02	BB	16
03	CC	17

输出:

02	BB	16
03	CC	17

(1) 根据 `Filter` 类的要求可知，需要添加属性，表示指定谓词的 `Predicate` 类对象 `p` 和待过滤的元组迭代器 `OpIterator` 对象 `child`。通过构造函数初始化。

(2) 通过 `get()` 方法返回谓词 `p`，通过调用 `child` 迭代器的 `getTupleDesc()` 方法返回这些元组遵循的元组模式。

(3) 编写操纵操作符的方法。打开本类中迭代器 `child`，同时由于 `Filter` 是某一个 `Operator` 类的子类，需要使用 `super` 打开其父类的迭代器。对于关闭迭代器同理，对 `super` 和 `child` 迭代器进行关闭。调用 `child` 的 `rewind()` 方法进行重置。

(4) 编写取过滤后下一个元组的方法 `fetchNext()`。

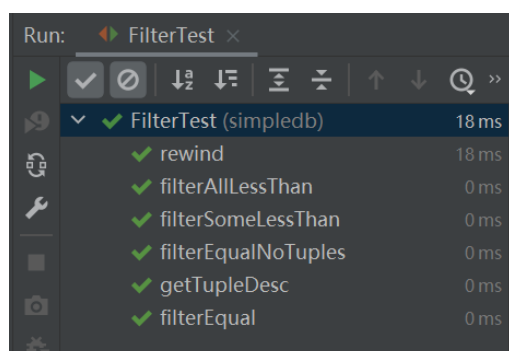
(5) 通过 `getChildren` 方法获取 `child`，即创建一个 `OpIterator` 数组，第一个元素为 `child`，返回数组。`setChildren` 方法即重置 `child`，令 `child` 等于给定的 `OpIterator` 数组的首元。

## 2.重难点:

通过 `fetchNext()` 方法取下一个满足条件的元组。利用 `child` 迭代器，循环进行当 `child` 迭代器中还有下一个元组时，取出下一个元组，调用谓词 `p` 的 `filter` 方法判断它是否满足条件，若满足则将其返回。

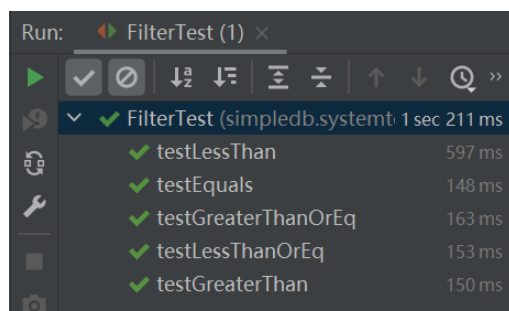
3.改动部分：未改动 API 和测试代码。

## 4.test 测试结果:



Run:	FilterTest	
✓	FilterTest (simplifiedb)	18 ms
✓	rewind	18 ms
✓	filterAllLessThan	0 ms
✓	filterSomeLessThan	0 ms
✓	filterEqualNoTuples	0 ms
✓	getTupleDesc	0 ms
✓	filterEqual	0 ms

## systemTest 测试结果:



Run:	FilterTest (1)	
✓	FilterTest (simplifiedb.systemt)	1 sec 211 ms
✓	testLessThan	597 ms
✓	testEquals	148 ms
✓	testGreaterThanOrEq	163 ms
✓	testLessThanOrEq	153 ms
✓	testGreaterThan	150 ms

## (三) JoinPredicate 类

### 1.设计思路:

`JoinPredicate` 类是 `Join` 类的辅助类，对两个 `tuple` 的某个字段进行比较。比较

的逻辑运算符与 `Predicate` 中相同。筛选出符合条件的两个元组。

(1) 根据 `JoinPredicate` 类的要求，需要设置属性，指明两个元组要进行比较的字段号 `field1` 和 `field2`，以及比较运算的操作符 `op`。

(2) 通过构造函数初始化，并通过 `get` 方法获取属性值。

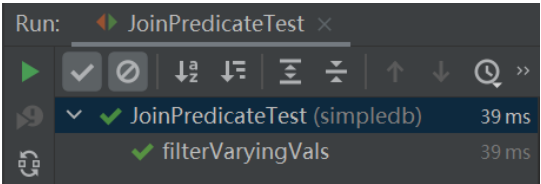
(3) 编写 `filter` 方法判断两个字段是否满足要求。

2.重难点：

通过 `filter()`方法完成比较。通过元组 `t1` 的 `getField` 方法找到字段编号 `field1` 的字段对应的内容。调用 `Field` 类中定义的 `compare(op,operand)`方法,其中 `operand` 为 `t2` 的 `field2` 字段对应的内容。满足条件则返回真值。

3.改动部分：未改动 API 和测试代码。

4.test 测试结果：



(四) Join 类

1.设计思路：

`Join` 类根据 `JoinPredicate` 的判断结果，实现对于满足条件的两个元组的连接操作，实现 `Operator` 接口。实现过程如下图所示。依次遍历 `tuple1` 和 `tuple2` 组成的元组对，返回满足 `JoinPredicate` 的元组对。

JoinPredicate p:

field1	op	field2
1	=	0

tuples1:

编号	姓名	年龄
01	AA	15
02	BB	16
03	CC	17

tuples2:

姓名	电话
AA	1234
BB	2345
CC	3456

Join输出:

01	AA	15	AA	1234
02	BB	16	BB	2345
03	CC	17	CC	3456

(1) 根据 `Join` 类的要求，需要添加属性，两个元组对应的迭代器 `child1` 和

child2, 指定谓词 JoinPredicate 的对象 p。通过构造函数初始化。

(2) 通过 get 方法获取谓词 p。接着要获取两个被连接字段的名称。先通过调用 child 的 getTupleDesc 方法获取元组模式 td, 再调用 td 的 getFieldName 方法, 传入谓词中的字段号, 即可获取两个字段的名称。

(3) 获取连接后的元组模式, 可以调用 TupleDesc 类中的静态方法 merge(), 传入 child1 和 child2 对应的元组模式, 获得合并后的元组模式。

(4) 打开、关闭、重置操作符的方法与 Field 类中类似。

(5) 编写取连接后下一个元组的方法 fetchNext()。

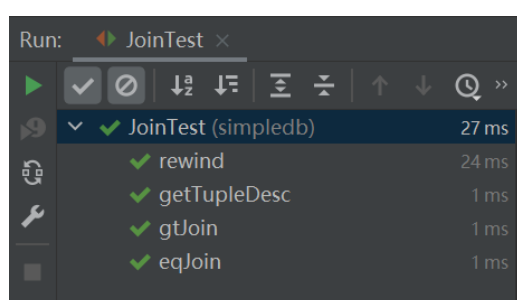
(6) getChildren 和 setChildren 方法与 Filter 类中类似, 但需要对两个 child 迭代器进行操作。

## 2.重难点:

通过 fetchNext()方法取下一个满足条件的元组。进行两层循环, 外层遍历第一个元组迭代器, 内层循环遍历第二个元组迭代器。即将两组元组一一组成元组对, 判断是否满足给定的谓词指出的关系。如果满足, 则将两者合并, 为新得到的元组设置元组的 RecordId, 为两块字段分别调用原两个元组的 setField 设置字段。注意, 当内层循环执行完一遍后, 通过调用 child2.rewind()方法, 回到第二个迭代器的头部。最终将得到的第一个新元组返回。

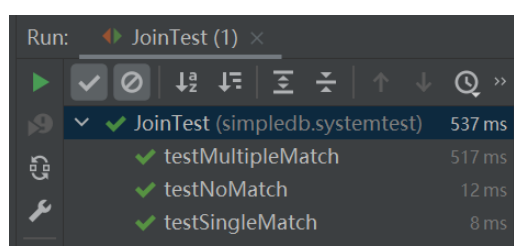
3.改动部分: 未改动 API 和测试代码。

## 4.test 测试结果:



Run: JoinTest x	
JoinTest (simplifiedb)	27 ms
rewind	24 ms
getTupleDesc	1 ms
gtJoin	1 ms
eqJoin	1 ms

## SystemTest 测试结果:



Run: JoinTest (1) x	
JoinTest (simplifiedb.systemtest)	537 ms
testMultipleMatch	517 ms
testNoMatch	12 ms
testSingleMatch	8 ms

### 三、聚合操作 (Aggregates)

需要实现五种聚合操作 (COUNT、SUM、MIN、MAX、AVG)，并支持分组聚合。每次将新的元组加入到所属分组的聚合中，并重新计算聚合值。当所有元组都操作完毕后，返回按组分类后的新元组，模式为 (groupValue, aggregateValue)。如果某元组没有分组，则返回元组模式为 (aggregateValue)。

#### (一) IntegerAggregator 类

##### 1. 设计思路：

IntegerAggregator 类用于对整型字段进行分组聚合操作，因此五种聚合操作都可以执行。

(1) 根据 IntegerAggregator 类的目的，可知需要实现对整型元组的聚合操作。可以为聚合操作的实现编写类 AggHandler，五种聚合操作分别继承聚合操作基类，完成相应的聚合操作编写。实现操作的函数 handle() 需要传入参数合并字段 gbField 和聚合操作字段 aggField。设置一个 <Field, Integer> 的 HashMap aggResult 来保存聚合操作得到的结果。

(2) 为 IntegerAggregator 类添加属性，需要添加分类字段的字段号 gbFieldIndex，分类字段的字段类型 gbFieldType，聚合操作字段的字段号 aggFieldIndex，以及声明聚合操作类 AggHandler 的对象 aggHandler。在构造函数中，为三个属性赋值，并判断传入的聚合操作符的类型，根据类型将 aggHandler 初始化相应的聚合操作子类。例如，若传入的操作符为 MAX，则 aggHandler=new MaxHandler()。

(3) 函数 mergeTupleIntoGroup 中，即要根据所给聚合操作符执行聚合操作。即调用类 AggHandler 中的函数 handle()。通过所给元组 t 的 getField 方法和字段号属性就可以获得聚合操作字段 aggField 和分组字段 gbField。注意，当分组字段号为-1 即不存在分组时，gbField 应为空值。将这两个字段传入 handle 方法中即可通过相应类的 handle 方法实现分组聚合操作。

(4) 编写迭代器，将获得的分组聚合结果组成的一系列元组存到迭代器中。

##### 2. 重难点：

(1) 编写不同的聚合操作子类。其中，MIN、MAX、SUM、COUNT 的编写过程类似。以 SUM 为例，通过调用传入的 aggField 的 getValue() 方法获取需要聚

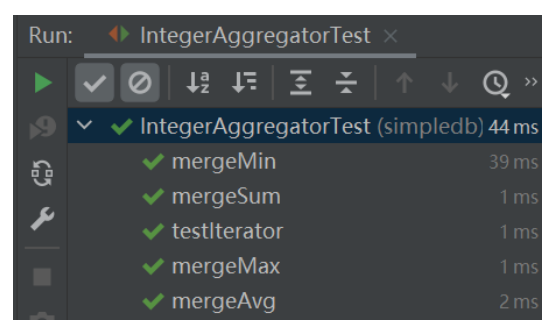
合操作的字段值；判断保存结果的 HashMap 中是否存在 gbField，若存在则在结果的值上加上新值，若不存在则添加新的键值对。

对于 AVG 操作，要相对复杂。可以再设置两个 HashMap，分别存<aggField, sum>和<aggField, count>，用于计算平均值。判断两个 HashMap 中是否存在 gbField，若存在则在和上加上新值、数量上加一，若不存在则添加新的键值对。最终将相应 gbField 对应的 sum 和 count 做除法，得到 avg 存入结果的 HashMap 中。

（2）编写迭代器，依次获取所有得到的元组。可以利用 TupleIterator 类，因此需要得到最终元组的模式 tupleDesc 和元组组成的列表 tuples。根据是否有分组属性，将最终的元组模式分为两种。若无分组，则元组模式为只有一个 INT\_TYPE 类型、名为“aggregateValue”的字段，将 aggResult 中键为 null 对应的值添加到元组的字段中，将所有元组添加到元组列表 tuples 中。若有分组，元组的模式为两个字段，类型分别为（gbFieldType, INT\_TYPE），名称分别为（“groupByValue”，“aggregateValue”），将 aggResult 中键为 gbField 对应的值添加到元组的字段中，将所有元组添加到元组列表 tuples 中。最终返回调用 TupleIterator 的结果。

3.改动部分：未改动 API 和测试代码。

4.test 测试结果：



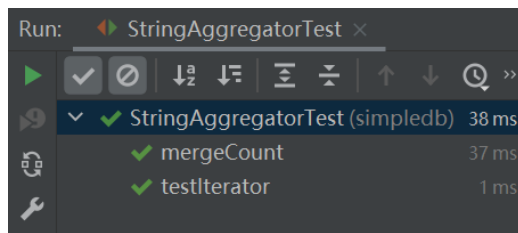
## （二）StringAggregator 类

### 1.设计思路：

StringAggregator 类用于对字符串类型字段进行分组聚合操作，因此只能执行 COUNT 聚合操作。

StringAggregator 类的实现与 Integer 几乎完全相同。只保留 COUNT 操作的子类。

## 2.test 测试结果:



### (三) Aggregate 类

#### 1.设计思路:

Aggregate 类实现了对 IntegerAggregator 和 StringAggregator 类的封装。应给出需要进行聚合操作的一些列元组，分组字段、聚合字段以及聚合操作符。

(1) 添加属性，添加 OpIterator 类型的 child 迭代器用于遍历所有元组。分组字段号 gbFieldIndex 和聚合字段号 aggFieldIndex，操作符 aop。

(2) 在构造函数中赋值，并通过 child 获得的元组模式、gbFieldIndex 和 aggFieldIndex 创建出结果元组对应的元组模式 tupleDesc。通过 get 方法获取分组字段号、聚合字段号、聚合操作符和元组模式。

(3) 为获取分组字段名称和聚合字段名称，需先判断是否有分组字段。若无分组字段，分组字段名为 null，聚合字段名为 tupleDesc 第一个字段的名称；若有分组字段，分组字段名为 tupleDesc 第一个字段的名称，聚合字段名为 tupleDesc 第二个字段的名称。

(4) 对 Aggregate 操作符进行打开、关闭、重置操作。打开时除了打开元组迭代器 child 和父类操作符，主要需要根据元组的 aggFieldType 选择调用 IntegerAggregator 还是 StringAggregator，并开启相应操作符的迭代器 aggIterator。

(5) 调用 fetchNext()方法获取下一个元组，判断 aggIterator 中是否有下一个元素，若有则返回，若无则返回空值。

(6) getChildren 和 setChildren 方法与 Filter 类中相同。

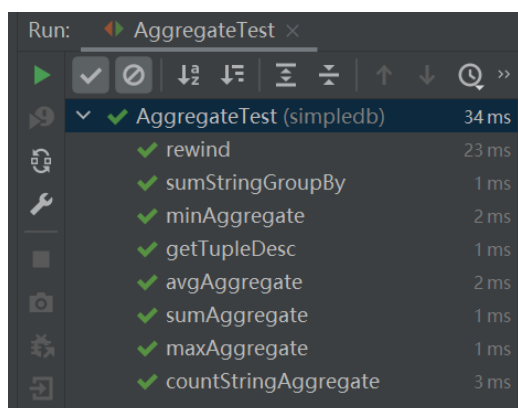
#### 2.重难点:

本类编写的难点是要确定对聚合字段类型判断并调用相应操作符是在 open 步骤进行的。根据聚合字段类型通过不同的构造函数构造操作符，并判断当 child 迭代器中还有下一个元素时，调用 mergeTupleIntoGroup 进行分组聚合操作。

3.改动部分：未改动 API 和测试代码。

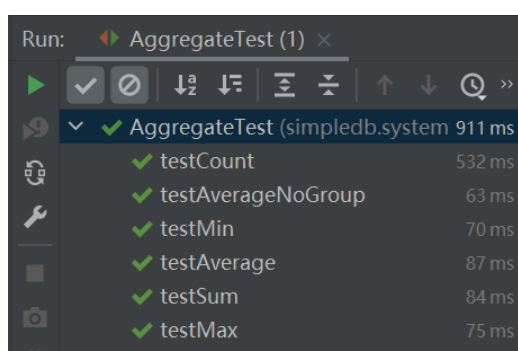


#### 4.test 测试结果:



Run:	AggregateTest (simplifiedb)	34 ms
✓	rewind	23 ms
✓	sumStringGroupBy	1 ms
✓	minAggregate	2 ms
✓	getTupleDesc	1 ms
✓	avgAggregate	2 ms
✓	sumAggregate	1 ms
✓	maxAggregate	1 ms
✓	countStringAggregate	3 ms

#### SystemTest 测试结果:



Run:	AggregateTest (1)	911 ms
✓	testCount	532 ms
✓	testAverageNoGroup	63 ms
✓	testMin	70 ms
✓	testAverage	87 ms
✓	testSum	84 ms
✓	testMax	75 ms

## 四、HeapFile 变化 (HeapFile Mutability)

要实现对元组的插入和删除, 需要对 HeapPage 和 HeapFile 进行操作, 同时操作后还要修改缓冲池 BufferPool。删除元组时需要通过元组的 RecordId 定位该元组被存储的页, 修改该页头 header 中标识该元组是否为空的 bit; 插入元组时需要找到有空闲 slot 的页, 将元组放到该 slot 中, 若没有则需新添加一页。

**重难点:** 需要注意, 由于在 SimpleDB 中对磁盘物理页的访问都是通过 BufferPool 进行, 因此实际的插入元组 (删除同理) 操作过程如下:

- (1) BufferPool 调用 HeapFile 的 insertTuple 方法。
- (2) HeapFile 调用 BufferPool 的 getPage 方法, 若 BufferPool 中有要找的页 (或含空 slot 的页) 则直接返回, 若没有则 BufferPool 调用 HeapFile 的 readPage 方法, 从磁盘读取相应 pageId 对应的页。
- (3) 找到页后调用 HeapPage 的 insertTuple 方法插入, 并将插入后的页返回给 BufferPool。此时会导致 BufferPool 中保存的页和磁盘上实际页内容不一致, 因此将该页标记为 dirty。

(4) 判断 `BufferPool` 中页是否满，不满则将返回的页放入，满了则置换旧页到磁盘，置换时若该页被标记为 `dirty`，则调用 `HeapPage` 的 `writePage` 方法将写回磁盘。

### (一) `HeapPage` 类

#### 1. 设计思路：

(1) 由于要标记脏页，所以添加属性 `boolean` 类型的 `dirty` 和产生脏页的事务 ID `TransactionId` 类型的 `dirtyId`。在 `markDirty()` 方法中为这两个属性赋值。使用 `isDirty()` 方法判断是否为脏页，若 `dirty` 为真，则返回 `dirtyId`。

(2) 插入或删除元组时需要通过 `markSlotUsed` 方法更新某些 `slot` 的值。通过元组编号 `i` 经过位运算定位到 `header` 中相应 `bit`。若传入的 `value` 为真值，则将该位更新为 1，反之更新为 0。

(3) 通过 `deleteTuple` 和 `insertTuple` 函数删除或插入元组。

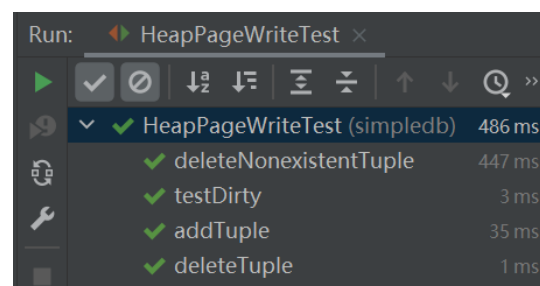
#### 2. 重难点：

(1) 删除元组时，首先根据传入的元组 `t` 获取其 `RecordId`。遍历该页所有 `slot`，若 `t` 的 `RecordId` 与该页第 `i` 个元组的相同，且第 `i` 个元组的 `slot` 被占用，则将该 `slot` 标记为 0、该元组标记为空，返回。若两个判断条件不能同时满足，继续寻找 `slot`，若所有 `slot` 均不满足，抛出没有匹配的元组异常。

(2) 插入元组时，首先判断本页是否有空 `slot`，若无则抛出该页已满异常。再判断所传元组 `t` 的模式是否和该页所存元组的模式匹配，不匹配抛出异常。接着遍历所有 `slot`，寻找第一个未被占用的 `slot`，将其标记为占用，将该 `slot` 对应的元组设置为 `t`，并为 `t` 设置 `RecordId`。

3. 改动部分：未改动 API 和测试代码。

#### 4. test 测试结果：



### (二) `HeapFile` 类

#### 1. 设计思路：

(1) 根据上述对插入(删除)元组时类和函数调用过程分析,完善 HeapFile 类中的 insertTuple 和 deleteTuple 类。

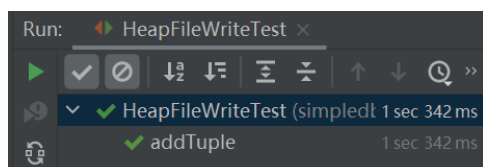
## 2.重难点:

(1) 插入元组的函数返回被修改过的页组成的列表。设置一个返回列表 pageList。遍历该 HeapFile 中已存在的所有页,如果当前遍历的页本身就在 HeapFile 中,则通过调用 BufferPool 的 getPage 方法获取该页;否则就新添加一页。直至找到具有空 slot 的一页 heapPage,调用 heapPage 的 insertTuple 方法插入元组,并将改动的页添加到页列表中。最终返回页列表。

(2) 删除元组的函数也返回被修改页组成的列表。设置一个返回列表 pageList。通过传入的 t 确定它的 RecordId,进而确定 pageId。如果该 pageId 所属的 tableId 与本 HeapFile 类的 id 相同,则通过 BufferPool 的 getPage 方法获取该页 heapPage,调用它的 deleteTuple 方法删除元组,将改动的页加到页列表中。

3.改动部分:未改动 API 和测试代码。

## 4.test 测试结果:

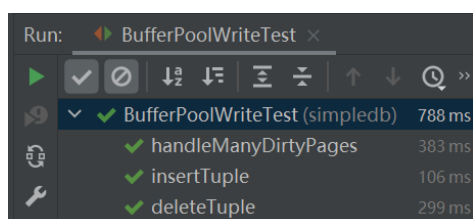


## (三) BufferPool 类

### 1.设计思路:

(1) BufferPool 的 insertTuple 和 deleteTuple 主要为了实现对 BufferPool 中插入元组页或删除元组页的更新,并判断是否需要向磁盘写入。通过 tableId 或元组 t 获得其所属的 HeapFile 类对象 file,调用 file 的 insertTuple 或 deleteTuple 获得一个被修改的 pageList。对该 pageList 中的每一个页都调用 markDirty 方法标记为脏页,判断此时 BufferPool 中页数是否超量,若超则调用 evict 方法进行页的置换,否则将该页放入 BufferPool 中。

### 2.test 测试结果:



## 五、插入和删除（Insertion and Deletion）

要实现插入和删除的操作符，即为对上一个练习三个与插入删除相关的类的方法进行封装。

### （一）Insert 类

#### 1.设计思路：

将元组迭代器中取到的元组通过调用 BufferPool 中的 insertTuple 方法插入到指定表中，最终返回受到影响的 tuple 的个数，以表的形式返回。

（1）根据 Insert 类的要求，需要添加属性，表示插入的事务 Id tid，要插入的一系列元组，通过迭代器 child 给出，以及被插入的表的 tableId。还可以设置返回元组的模式 tupleDesc，记录被影响元组的个数的 count 和标记每个元组是否已经被插入的 isInserted。

（2）通过构造函数赋值。未开启状态下 count=-1，isInserted=false。tupleDesc 中只有一个 INT 类型字段，表示受影响的元组个数，通过 get 方法获取 tupleDesc。

（3）开启操作符时将 count 值设为 0，打开迭代器 child 和其父类操作符；关闭时将属性值恢复为关闭状态；重置时将属性值恢复到刚开启状态。

（4）通过 fetchNext 方法完成 child 中元组的插入，并返回被影响的数量。

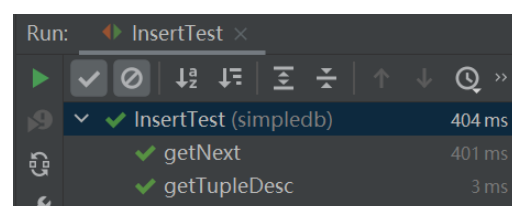
（5）getChildren 和 setChildren 方法与 Filter 类中类似。

#### 2.重难点：

编写 fetchNext()方法。首先判断是否已经被插入，若已被插入返回 null，否则将标记值 isInserted 设置为真。若 child 迭代器有下一个元素，则通过调用 BufferPool 中的 insertTuple 方法插入下一个元素，并使 count 值加一，循环执行知道 child 迭代器中无下一个元素。设置一个模式为 tupleDesc 的元组，将 count 值作为字段内容添加，返回该元组。

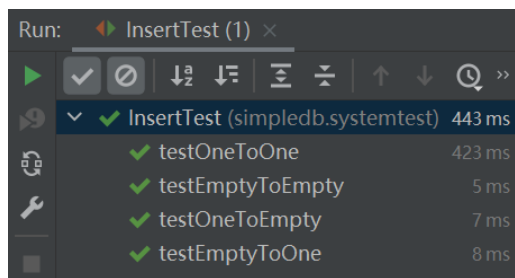
3.改动部分：未改动 API 和测试代码。

#### 4.test 测试结果：



Run:	InsertTest	
✓	InsertTest (simplifiedb)	404 ms
✓	getNext	401 ms
✓	getTupleDesc	3 ms

SystemTest 测试结果如下：



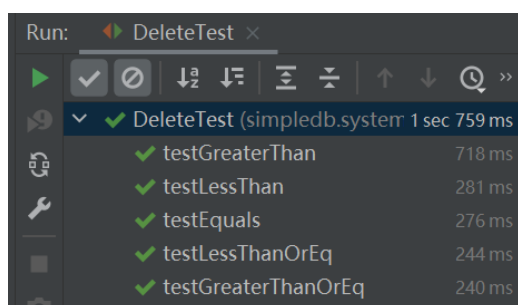
Run: InsertTest (1) x

✓	InsertTest (simplifiedb.systemtest)	443 ms
✓	testOneToOne	423 ms
✓	testEmptyToEmpty	5 ms
✓	testOneToEmpty	7 ms
✓	testEmptyToOne	8 ms

## （二）Delete 类

Delete 类的设计思路与 Insert 类几乎完全相同，只需改为调用 BufferPool 中的 deleteTuple 函数。

SystemTest 测试结果如下：



Run: DeleteTest x

✓	DeleteTest (simplifiedb.system 1 sec)	759 ms
✓	testGreaterThan	718 ms
✓	testLessThan	281 ms
✓	testEquals	276 ms
✓	testLessThanOrEq	244 ms
✓	testGreaterThanOrEq	240 ms

## 六、页面驱逐（Page Eviction）

当 BufferPool 满了之后需要进行页面置换。当置换页为脏页时需要调用 HeapFile 中的 writePage 方法将脏页写入磁盘中。

### （一）HeapFile 类

完成向磁盘中写入一页的操作，定位某页时需要知道该页在该文件中的编号和每一页的大小。通过参数 page 获取它的 pageId，进而获取它的编号 pgNo。若编号大于该文件最大页数则抛出异常。否则通过写文件流 f 定位到该页，将 getPageData 方法获得的数据写入磁盘页。

### （二）BufferPool 类

#### 1.设计思路：

在置换操作时，若该页为脏页要将其重新写入磁盘，可以设置 flushPage()方法；之后将该页从缓冲池上清除，可以设置 discardPage()方法。

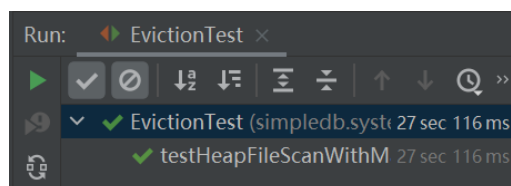
（1）在 flushPage 方法中，通过给出的 PageId 获取该页 page，判断它是否为脏页，若为脏页则调用 HeapFile 中的 writePage 方法写入该页，并将该页的脏页

标记删除。

(2) `discardPage` 方法中，只需将给出的 `PageId` 和其对应的 `page` 组成的键值对从 `buffer` 中移除。

(3) `evictPage` 方法中获取 `HashMap buffer` 中的第一个键值，调用 `flushPage` 判断它是否为脏页并执行相应操作，再调用 `discardPage` 将其从缓冲池中删去。

2.SystemTest 测试结果：



## 七、连接查询示例

设置两个表的数据，分别为：

1	1,1,1	1	1,1,1
2	2,2,2	2	2,2,2
3	3,3,3	3	4,4,4
4	4,4,4	4	5,5,5
5	5,5,5	5	6,6,6
6	6,6,6		

按照示例编写 `jointest` 文件，运行结果为：

```
D:\SimpleDB\SimpleDB>java -classpath dist/simplifiedb.jar simplifiedb.jointest
2      2      2      2      2      2
4      4      4      4      4      4
5      5      5      5      5      5
6      6      6      6      6      6
```

满足预期。

## 八、查询解析器

创建数据文件，并将其转成表。在命令行按照要求操作，得到预期结果。

```
d. f1    d. f2
-----
1         10
2         20
3         30
4         40
5         50
5         50

6 rows.
Transaction committed
```