npulogo

**数据库系统综合实验报告**

|  |  |
| --- | --- |
| 姓 名 | 吴奇 |
| 学 号 | 2019302817 |
| 班 级 | 10011905 |
| 任课教师 | 张利军 |

|  |
| --- |
|  |
| 2022年6月20日 |

Contents

[1. Lab1 Storage Model 3](#_Toc106967449)

[1.1. Exercise 1 Tuple and TupleDesc 5](#_Toc106967450)

[1.2. Exercise 2 Catalog 8](#_Toc106967451)

[1.3. Exercise 3 BufferPool 9](#_Toc106967452)

[1.4. Exercise 4 HeapPage 10](#_Toc106967453)

[1.5. Exercise 5 HeapFile 14](#_Toc106967454)

[1.6. Exercise 6 SeqScan 15](#_Toc106967455)

[2. Lab2 Operators 16](#_Toc106967456)

[2.1. Exercise 1 Filter and Join 17](#_Toc106967457)

[2.2. Exercise 2 Aggregates 22](#_Toc106967458)

[2.3. Exercise 3 Heap File Mutability 27](#_Toc106967459)

[2.4. Exercise 4 Insertion and Deletion 31](#_Toc106967460)

[2.5. Exercise 5 Page Eviction 33](#_Toc106967461)

[3. Lab3 Optimizer 34](#_Toc106967462)

[3.1. Exercise 1 and 2 Filter Selectivity 38](#_Toc106967463)

[3.2. Exercise 3 Join Cardinality 41](#_Toc106967464)

[3.3. Exercise 4 Join Ordering 43](#_Toc106967465)

[4. Lab4 Transaction 45](#_Toc106967466)

[4.1. Exercise 1 Granting Locks 47](#_Toc106967467)

[4.2. Exercise 2 Lock Lifetime 49](#_Toc106967468)

[4.3. Exercise 3 Implementing NO STEAL 51](#_Toc106967469)

[4.4. Exercise 4 Transactions 52](#_Toc106967470)

[4.5. Exercise 5 Deadlocks and Aborts 55](#_Toc106967471)

[5. Lab5 B+ Tree Index 56](#_Toc106967472)

[5.1. Exercise 1 Search 58](#_Toc106967473)

[5.2. Exercise 2 Insert 60](#_Toc106967474)

[5.3. Exercise 3 Delete 64](#_Toc106967475)

[6. Lab 6 Recovery and Rollback 69](#_Toc106967476)

[6.1. Exercise 1 Rollback 72](#_Toc106967477)

[6.2. Exercise 2 Recovery 74](#_Toc106967478)

[7. 总结 76](#_Toc106967479)

[7.1. SimpleDB 总体架构 76](#_Toc106967480)

[7.2. 实验收获 77](#_Toc106967481)

[7.3. 实验不足 78](#_Toc106967482)

[7.4. 建议 78](#_Toc106967483)

[8. 参考资料 79](#_Toc106967484)

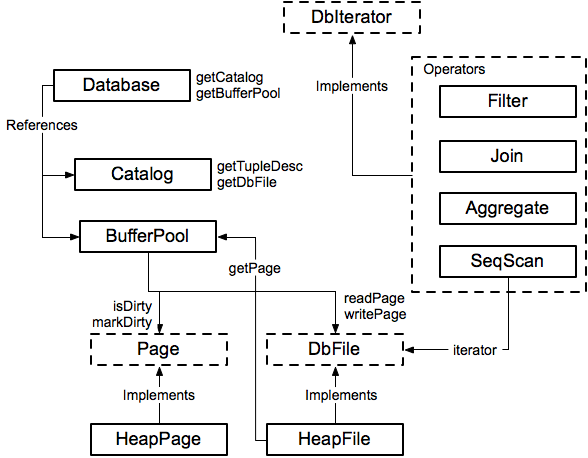
## 1. Lab1 Storage Model

#### **实验内容：**

lab1主要是实现了整个数据库的整体架构,主要有这几个部分：Tuple(元组)、TupleDesc(table的metadata)、catalog(该数据库并没有过度区分catalog和schema，可以看成是一个schema)、BufferPool(缓冲池)、HeapPage(数据页)、HeapFile(disk上的文件)、SeqScan(全表顺序扫描)。主要包括以下内容：

* Tuple：元组，数据库上把一个有n列的table称作n元组，一个Tuple有多个字段。通俗来讲，一条记录就是一个元组，在该实验中体现为一个Tuple类的实例。
* TupleDesc：TupleDesc用来表示一个元组的描述信息，更准确来说应该是一个表的描述信息；
* Catalog：Catalog是仅次于DataBase的抽象概念，一个DataBase可以有多个Catalog，一个Catalog有多个Schema，一个Schema有多张table。
* BufferPool：BufferPool的基本单位是Page，每次从磁盘中(这里表现为DbFile)读取数据页到BufferPool，在数据库上的crud操作都是在Buffer Pool的Page中进行的(所以有脏页、故障恢复等)。该数据库的BufferPool默认是缓冲50个Page，每个Page的默认大小是4096bytes即4kb。
* HeapPage与HeapFile：HeapPage是Page接口的实现，是以顺序逻辑组织数据的一张数据页(后续会用b+树实现来替代)，crud操作都是在Page上进行的。HeapPage既是BuffeerPool的基本单位也是HeapFile的基本单位。HeapFile是DbFile接口的实现，与磁盘中的文件交互，该数据中一张表的所有数据就是存到DbFile的File属性中，即一个磁盘的文件就是一张表的所有数据。
* SeqScan：全表顺序扫描的实现，相当于select \* from table。

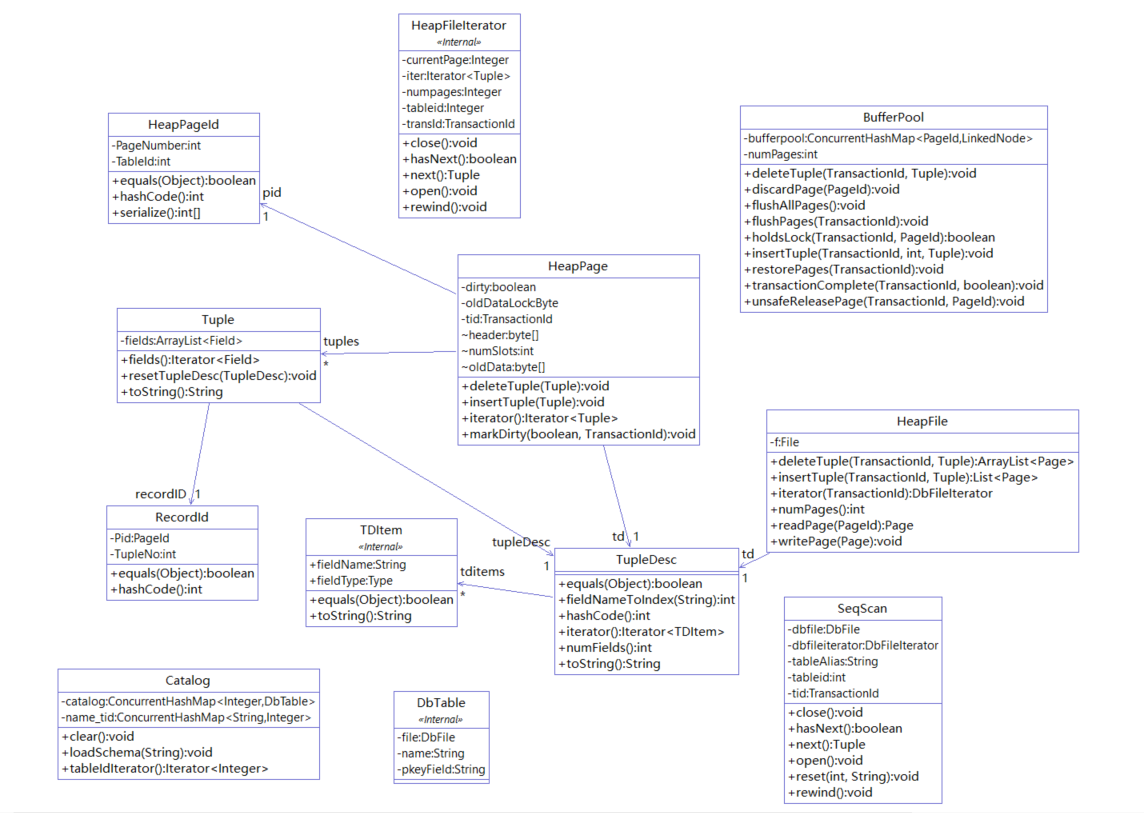
#### **实验类图：**



Module Implementation Diagram



各部分组成关系



lab1类图

### 1.1. Exercise 1 Tuple and TupleDesc

#### **本Exercise实验内容：**

实现了元组以及元组的属性行。元组，数据库上把一个有n列的table称作n元组，一个Tuple有多个字段。通俗来讲，一条记录就是一个元组，在该实验中体现为一个Tuple类的实例。一个Tuple由以下部分组成：a.TupleDesc：该元组的描述信息；b.fields：该记录各个字段的类型与值; c.RecordId：该记录在磁盘的位置。

#### **实现内容：**

**TupleDesc类**

用来表示一个元组的描述信息，更准确来说应该是一个表的描述信息，TupleDesc就是tuple的schema。

TupleDesc中提供了一个TDItem辅助类，类中有Type fieldType、String fieldName 两个参数。例如id为一个TDItem，filedName==序号，fieldType==int。 Type为枚举类型，实现了INT\_TYPE和STRING\_TYPE。且fieldName可以为空。可以创建数组或链表存储TDItem。

* public TupleDesc(Type[] typeAr, String[] fieldAr)：初始化方法
* public Iterator iterator（）：返回所有属性的迭代器
* public int numFields()：返回TupleDesc中属性的数量
* public String getFieldName(int i)：返回第i个属性的属性名
* public Type getFieldType(int i)：返回第i个属性的属性类型
* public int fieldNameToIndex(String name)：根据属性名返回它在tdItem中的序号
* public int getSize()：返回元组所占的字节大小
* public static TupleDesc merge(TupleDesc td1, TupleDesc td2)：合并两个tdItem
* public boolean equals(Object o)：euqals方法，只要两个TupleDesc的属性数量相等，且td1的第i个属性类型==td2的第i个属性类型，则两个TupleDesc的相等
* public String toString()：toString方法返回TupleDesc的所有属性名：“id,name,age”

**Tuple类**

Field接口，包含compare()、getType()、equals()、toString()方法

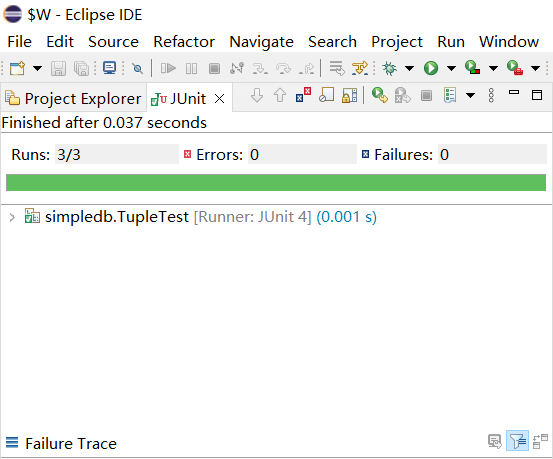
IntField和StringFiled类实现了该接口,public boolean compare(Predicate.Op op, Field val)根据查询逻辑op（equals、not\_equals、greater\_than、less\_than等）返回该字段与val字段比较的结果。

* public Tuple(TupleDesc td)：构造函数，TupleDesc 代表的是数据库中表头的一行的具体类型 包括了一张表中的数据类型和名称。然后TupleDesc.java中就有个数组 TDItem[ ]代表了这张表里面的表头，数据类型就是在里面定义的静态的内部类。
* public TupleDesc getTupleDesc()：获得Tuple对应的TupleDesc
* public void setRecordId(RecordId rid)：设置元组id
* public RecordId getRecordId()：获得元组id
* public void setField(int i, Field f)：为字段赋值
* public Field getField(int i)：获得指定字段
* public String toString()：toString()方法返回所有字段
* public Iterator fields()：返回字段的迭代器
* public void resetTupleDesc(TupleDesc td)：重置TupleDesc

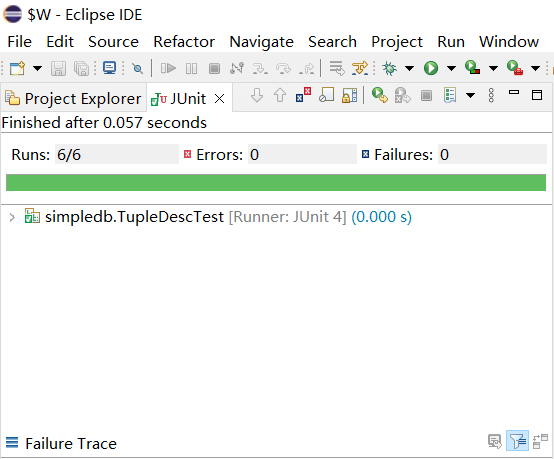
#### **问题：**

在实现TupleDesc class 时, 对于类属性中的属性链表，一开始编写程序时没有直接生成对应 对象，后续方法使用 TupleDesc时无法访问链表中的 IDItem 导致测例错误。

#### **测例：通过本Exercise全部测例**



TupleTest



TupleDescTest

### 1.2. Exercise 2 Catalog

#### **本Exercise实验内容：**

实现 Catalog类:

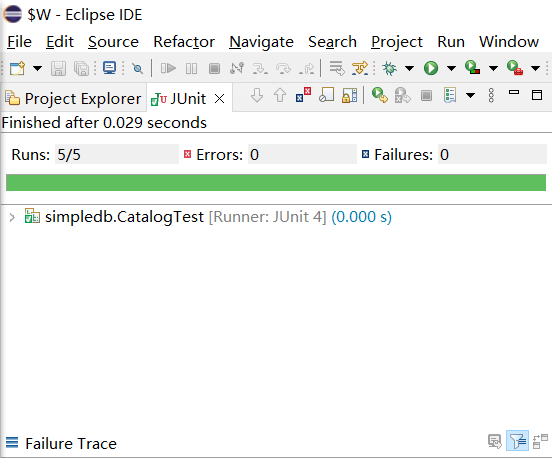
Catalog：目录。数据库包含很多张表，每张表有一个TupleDesc，以及这个TupleDesc规范下的很多个Tuple。Catalog管理 着数据库中的所有表。调用数据库的Catalog需要调用Database.getCatalog()方法。  
 DbFile：为数据库磁盘文件的接口。数据库中每张表对应着一个DbFile，DbFile储存着表中的所有信息。

#### **实现内容：**

**TupleDesc类**

* public Catalog()：构造方法，通过表名获取TableId（也就是ConcurrentHashMap<Integer,Table>( )的Integer）的函数、通过TableId获取TupleDesc的函数、通过TableId获取Dbfile的函数、通过TableId获取主键的函数以及原本提供的loadSchema函数。
* public void addTable(DbFile file, String name, String pkeyField)：向CataLog中添加表,主键可为空、name可为空，name为空时随机一个UUID作为其name。
* public int getTableId(String name)：通过表名获得表id
* public TupleDesc getTupleDesc(int tableid)：通过表id 获得表的TupleDesc
* public DbFile getDatabaseFile(int tableid)：通过表id 获得表的内容DbFile
* public String getPrimaryKey(int tableid)：通过表id 获得表的主键
* public Iterator tableIdIterator()：返回tableId的迭代器
* public String getTableName(int id)：通过表id 获得表名
* public void clear()：清空CataLog

#### **测例：通过本Exercise全部测例**



CatalogTest

### 1.3. Exercise 3 BufferPool

#### **本Exercise实验内容：**

实现BufferPool类: 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，只需要实现构造器和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。通过Database.getBufferPool()获得bufferPool.

#### **实现内容：**

**BufferPool类**

* public BufferPool(int numPages)：构造函数
* public Page getPage(TransactionId tid, PageId pid, Permissions perm)：通过Database中的getCatalog方法获取catalog，然后再通过getDatabaseFile获取catalog中的具体的table的Dbfile，然后通过readPage方法获取具体某一页的page信息，再存储在pageStore中，pageStore为ConcurrentHashMap<Integer,Page>( )。

#### **问题：**

lab1只需要实现getpage这一个方法且后续还需要不断地重构，因此比较简单,暂时没有问题。

#### **测例：本Exercise无测例**

### 1.4. Exercise 4 HeapPage

#### **本Exercise实验内容：**

对象被排列成一组页面，每个页面由固定数量的字节组成，用于存储‎‎元组（由常量定义），包括一个标头。在 SimpleDB 中‎，数据库中的每个表都有一个对象（一个HeapFile对应一张表）。中的每个页面都排列成一组槽，每个‎‎槽可以容纳一个元组（SimpleDB 中给定表的元组大小相同）。除了这些槽之外，‎每个页面都有一个标头，该标头由一个位图组成，每个元组槽有一个位。如果与特定‎‎元组对应的位为1，则表示该元组有效;如果为 0，则元组无效。例如，已被删除或从未初始化过‎对象页是实现接口的类型。‎页‎‎存储在缓冲池中，但由‎‎HeapFile‎‎类读取和写入。‎

实现HeapPageId类、RecordId类、HeapPage类。HeapPage中包含多个slot和一个header，每个slot是留给一行的位置。header是每个tuple slot的bitmap。如果bitmap中对应的某个tuple的bit是1，则这个tuple是有效的，否则无效（被删除或者没被初始化）。

#### **实现内容：**

**HeapPageId类**

存储的是tableId和pagenumber的对应关系，实现了读取TableId、读取PageNumber、返回hashCode等函数。

* public HeapPageId(int tableId, int pgNo)：构造函数
* public int getTableId()：返回该pageID所在表的ID
* public int getPageNumber()：返回该pageId对应的序号
* public int hashCode()：返回该pageId的hashcode"表id+page序号"
* public boolean equals(Object o)：equals方法

**RecordId类**

存储的是pageId和tuplenumber的对应关系，实现了读取TupleNumber、读取PageId、返回hashCode等函数。

* public RecordId(PageId pid, int tupleno)：构造方法
* public int getTupleNumber()：返回元组的序号
* public PageId getPageId()：返回元组所属页的pageId
* public boolean equals(Object o)：equals方法
* public int hashCode()：hashcode方法，返回"tableId+pageNo+tupleNo"

**HeapPage类**

实现了从硬盘内读取数据将数据实例化成heapPage，其实就是将HeapFile实例化一个个的HeapPge。

* private int getNumTuples()：返回每个page中最多包含的tuple数
* 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：
* 其中，tuple size是页中单个tuple 的bytes大小。
* private int getHeaderSize()：返回page中的header的大小

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：

。

* public HeapPageId getId()：返回pageId
* public boolean isSlotUsed(int i)：判断第i个slot是否为空
* public int getNumEmptySlots()：返回page中空slot的数量
* public Iterator<Tuple> iterator()：返回HeapPage中所有元组的迭代器（不能包括空slot）

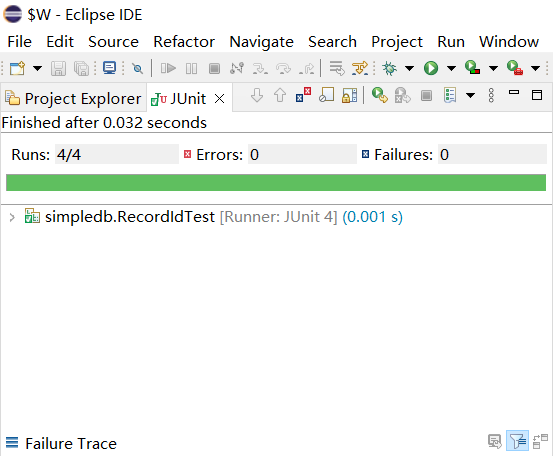
#### **问题：**

在处理空白位图时，由于一开始对位图的理解不够深刻，导致计算空slot等方法出现错误。

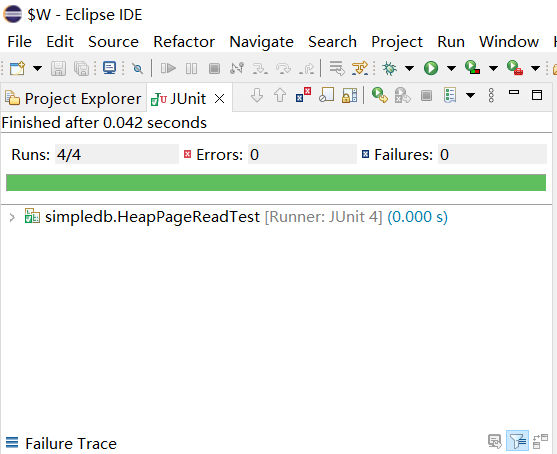
#### **测例：通过本Exercise全部测例**



HeapPageIdTest



RecordIDTest



HeapPageReadTest

### 1.5. Exercise 5 HeapFile

#### **本Exercise实验内容：**

要从磁盘读取页面，首先需要计算文件中的正确偏移量。提示：需要随机访问该文件，以便以任意偏移量读取和写入页面。从磁盘读取页面时，不应调用缓冲池方法。还需要实现HeapFile.iterator()方法，它应该遍历HeapFile中每个页面的元组。迭代器必须使用BufferPool.getPage()方法来访问“HeapFile”中的页面。此方法将页面加载到缓冲池中，并最终用于（后续的Lab）实现基于锁定的并发控制和恢复。不要在调用open()时将整个表加载到内存中——这将导致非常大的表出现内存不足错误。

实现HeapFile类，HeapFile对象包含一组“物理页”，每一个页大小固定，大小由BufferPool.DEFAULT\_PAGE\_SIZE定义，页内存储行数据。在SimpleDB中，数据库中每一个表对应一个HeapFile对象，HeapFile对象中的物理页的类型是HeapPage，物理页是存储在buffer pool中，通过HeapFile类读写。

#### **实现内容：**

**HeapFile类**

* public HeapFile(File f, TupleDesc td)：构造函数，并实例化了File、TupleDesc这两个成员变量，并提供了get方法。还提供了readPage函数，通过传入PageId返回Page。需要注意的是从文件中找到当前PageId对应的偏移量，通过seek函数移动指针，然后再用read方法进行读取。
* public File getFile()：返回表的内容
* public int getId()：返回标识次表的唯一id

为确保返回 id 的唯一性，需要对heapfile的绝对路径进行 hash。

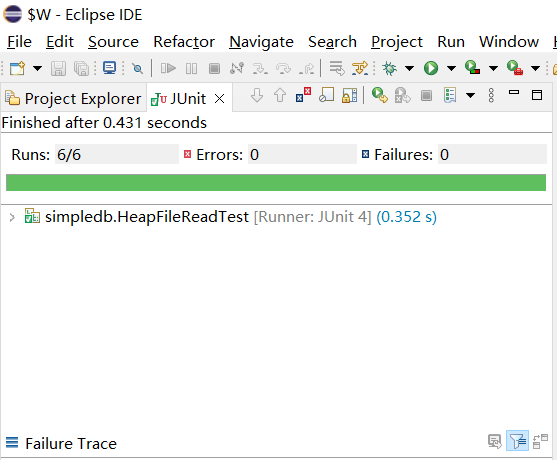
* public Page readPage(PageId pid)：读取page
* 在读取page时，readPage()方法仅会被BufferPool中的getPage()方法调用，而在其他位置需要获取page时，均要通过BufferPool调用。这也是BufferPool的意义所在。用RandomAccessFile去读文件，通过seek(offset)可以直接访问偏移量所对应的位置而不用从头进行查找，然后read(buf)，将偏移量后面buf.length长度的数据保存到buf中。
* public DbFileIterator iterator(TransactionId tid)：返回HeapFile中所有的heapPage 中的所有元组的迭代器

#### **问题：**

迭代器的实现：需要可以遍历所有的tuple，而tuple分布在多个page上，故需要在next的时候进行一定的处理。自己不熟悉java的iterator，写的时候逻辑混乱，最后整理逻辑后得到如下结果。

* 调用next的时候，都会调用hasNext去探探路，所以逻辑只需要放在hasNext中即可。
* 迭代器初始化的时候不指向元素，跟c++不太一样。
* 迭代器中的读取需要通过buffer，不能直接使用HeapFile读取，因为这样无法利用cache。

#### **测例：通过本Exercise全部测例**



HeapFileReadTest

### 1.6. Exercise 6 SeqScan

#### **本Exercise实验内容：**

数据库Operators(操作符)负责查询语句的实际执行。在SimpleDB中，Operators是基于迭代器实现的，每种iterator实现了一个DbIterator接口。SeqScan则为顺序扫描的功能，提供表内数据的迭代。实现SeqScan类，该类应该能够通过构造函数时传入的tableId扫描对应表中的所有元组，使用DbFile.iterator()访问元组。

#### **实现内容：**

**SeqScan类**

* public SeqScan(TransactionId tid, int tableid, String tableAlias)：构造函数,创建一次特定的transaction对特定table内的数据进行遍历，有三个参数，第一个参数是transaction对应的tid；第二个参数是tableId；第三个参数是表的别名，用于解析。
* public String getAlias()：返回表的别名
* public void reset(int tableid, String tableAlias)：查找新的表，重新对tableId、tableAlias赋值
* public TupleDesc getTupleDesc()：返回tupleDesc，在tupleDesc中的filedName前添加表的别名

#### **问题：**

之前写的getHeaderSize()函数计算字节数时直接调用库函数

337/8=42导致分配的是42bytes的数组， 之后自己实现 ceil（）函数功能才可以。

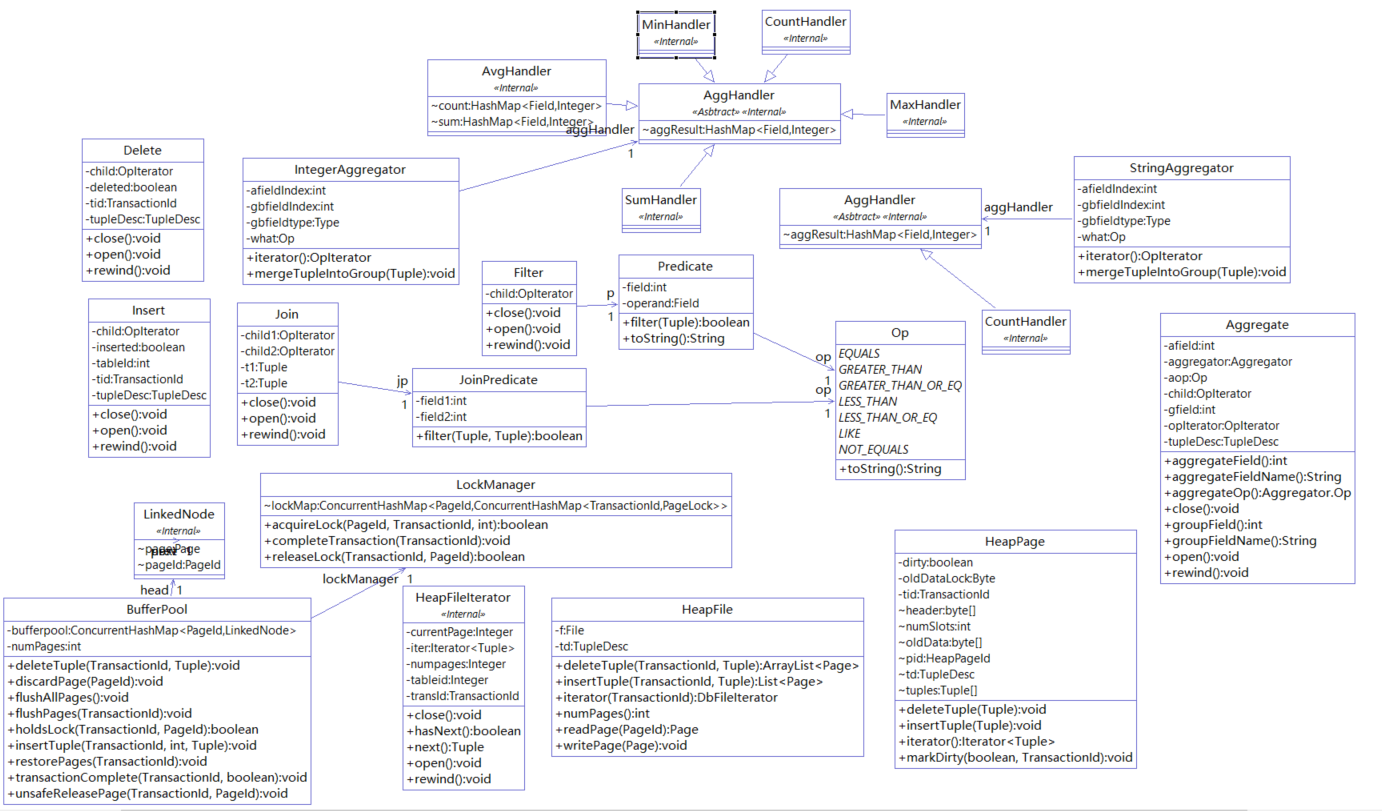
#### **测例：本Exercise无测例**

## 2. Lab2 Operators

#### **实验内容：**

* 实现过滤、连接运算符，这些类都是继承与OpIterator接口了，该实验提供了OrderBy的操作符实现，可以参考实现。最终的SQL语句解析出来都是要依靠这些运算符的；
* 实现聚合函数，由于该数据库只有int和string两种类型，int类型可实现的聚合函数有max,min,avg,count等，string类型只需要实现count；这些与分组查询一起使用，选择进行聚合操作时，可以选择是否进行分组查询。
* 对IntegerAggregator和StringAggregator的封装，查询计划是调用Aggregate，再去调用具体的聚合器，最后获得聚合结果。
* 实现插入、删除记录。包括从HeapPage、HeapFile、BufferPool中删除，这里需要把三个之间的调用逻辑搞清楚，代码会好写很多。
* 实现BufferPool的数据页淘汰策略。BufferPool的默认数据页容量为50页，进行插入数据页的操作时，如果数据页数量大于BufferPool的容量，需要选择某中页面淘汰策略去淘汰页面，我选择的是LRU算法来淘汰页面。

#### **实验类图：**



lab2类图

### 2.1. Exercise 1 Filter and Join

#### **本Exercise实验内容：**

实现Predicate、JoinPredicate、Filter、Join 类

#### **实现内容：**

**Predicate类**

Filter类的辅助类，用于筛选满足条件的tuple。将tuple中的字段与指定的字段进行比较，实现对单个tuple的过滤操作，比较逻辑有==、 >=、 <=、 >、 <、 !=、 LIKE(主要针对字符串)。

* public Predicate(int field, Op op, Field operand)：初始化方法
* public int getField()：返回fieldNo
* public Op getOp()：返回op
* public Field getOperand()：返回opField
* public boolean filter(Tuple t)：对元组t 进行比较

**Filter类**

Filter实现了Operator接口。根据Predicate的判读结果，得到满足条件的tuples。实现了where age > 18这样的操作。通过使用Predicate.java，在[构造函数](https://so.csdn.net/so/search?q=%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&spm=1001.2101.3001.7020)中实例化Predicate 和 OpIterator，通过fetchNext函数逐个读取OpIterator中的Tuple，然后让他们与Predicate 中的字段进行比较，如果为真则返回该Tuple。

* public Filter(Predicate p, OpIterator child)：初始化方法
* public Predicate getPredicate()：返回predicate
* public TupleDesc getTupleDesc()：返回待过滤元组的属性，用child.getTupledesc()即可
* public void open()：Filter是项目中的Operator类的子类， 需要执行super.open()
* public void close()：对child 和 super进行close
* protected Tuple fetchNext()：返回过滤后的tuple
* public OpIterator[] getChildren()：返回待过滤的tuples
* public void setChildren(OpIterator[] children)：重置待过滤的tuples

**JoinPredicate类**

实现连接的条件，和Predicate类似 ，是JoinPredicate的辅助类，对两个tuple中的 某一字段进行比较，但是实现表内字段的比较。

* public JoinPredicate(int field1, Predicate.Op op, int field2)：初始化方法
* public int getField1()：返回fieldNo1
* public int getField2()：返回fieldNo2
* public Predicate.Op getOperator()：返回op
* public boolean filter(Tuple t1, Tuple t2)：对t1中的fieldNo1字段 与t2 中的fieldNo2字段进行比较

**Join类**

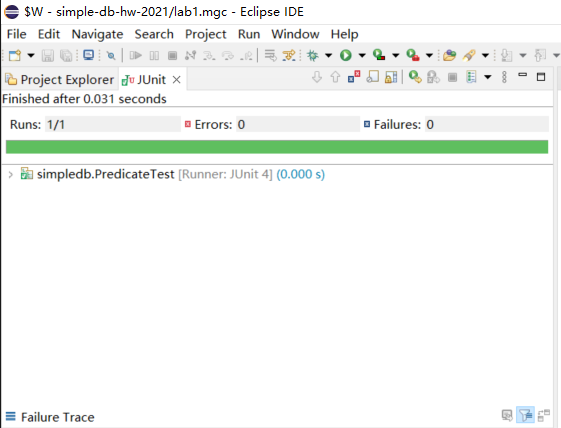
同样实现了Operator接口。实现连接操作（自然连接），给定两组tuples对这两组tuples中满足JoinPredicate操作的两个tuple进行连接。连接方式采用嵌套-循环连接。通过构造函数实例化JoinPredicate 和两个OpIterator 。实现一系类get方法和open、close等迭代器的函数。最后完成fetchNext函数找到两个迭代器中可以jion的字段进行join。

* public Join(JoinPredicate p, OpIterator child1, OpIterator child2)：初始化方法
* public JoinPredicate getJoinPredicate()：返回joinPredicate
* public String getJoinField1Name()：left tuple中进行比较的字段的名字
* public String getJoinField2Name()：right tuple中进行比较的字段的名字
* public TupleDesc getTupleDesc()：返回连接后的tuple的属性，TupleDesc中的merge操作
* protected Tuple fetchNext()：先获取child1中的一个tuple 赋值给left，left依次与child2中的tuples进行比较，与满足连接条件的 right 进行连接并返回连接后的tuple，遍历完child2之后，left = child1.next()， child2.rewind()

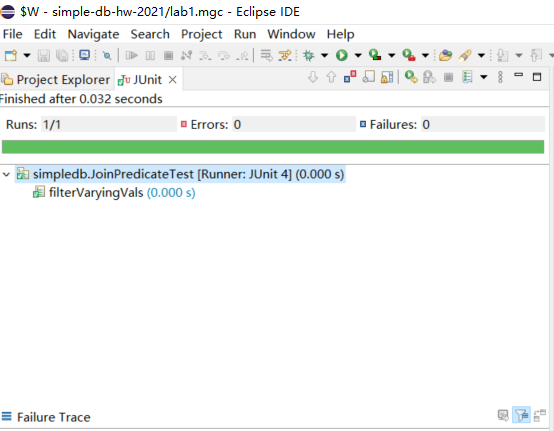
public OpIterator[] getChildren()：返回child1、child2

* public void setChildren(OpIterator[] children)
* public void open()
* public void close()
* public void rewind()

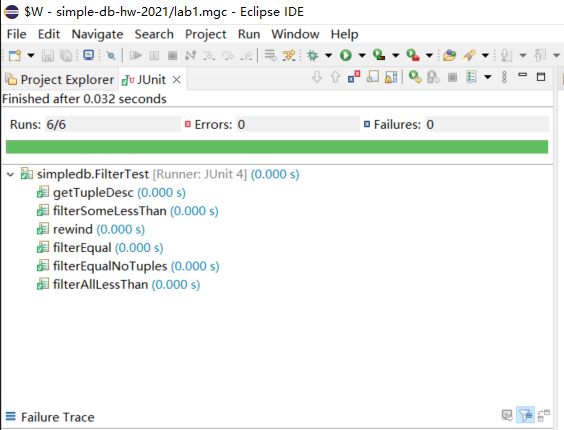
#### **测例：通过本Exercise全部测例**



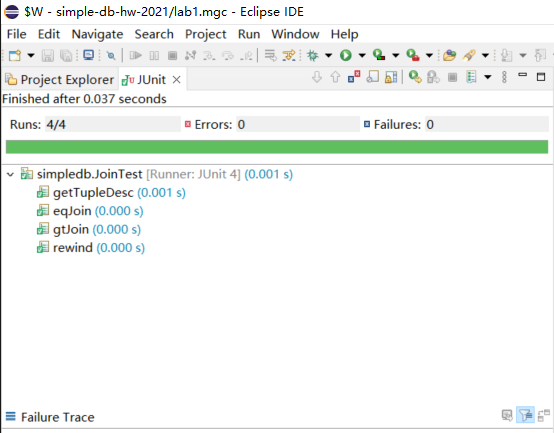
PredicateTest



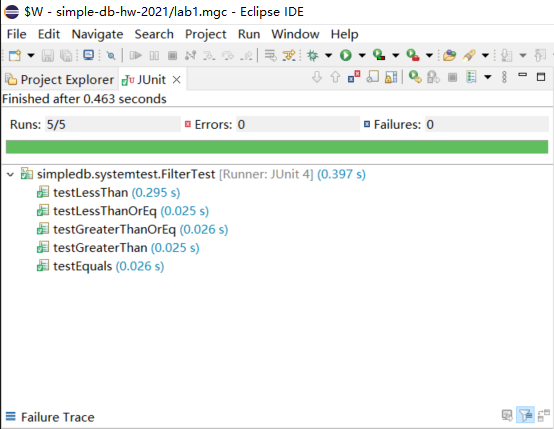
JoinPredicateTest



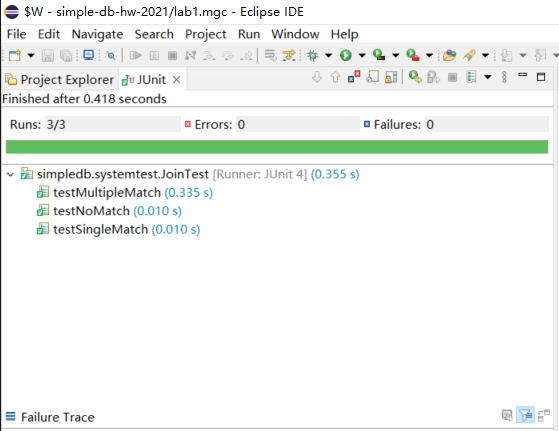
FilterTest



JoinTest



system tests FilterTest



system tests JoinTest

### 2.2. Exercise 2 Aggregates

#### **本Exercise实验内容：**

实现IntegerAggregator、StringAggregator、Aggregate类。exercise2中实现五中SQL聚合(COUNT,SUM,AVG,MIN,MAX)，同时支持分组。调用为了计算聚合，我们使用Aggregator接口，该接口将新tuple通过Aggregator.mergeTupleIntoGroup() 融进已存在的一个聚合。当所有的tuple都被合并后，可以得到聚合结果的迭代器。结果中的每个元组有分组字段、聚合字段(groupValue, aggregateValue)，当 group by 字段的值是Aggregator.NO\_GROUPING时，结果中的元组是单个字段(aggregateValue)。

***实现思路：***

不管是IntegerAggregator还是StringAggregator，他们的作用都是进行聚合运算（分组可选），所以他们的核心方法在于mergeTupleIntoGroup。IntegerAggregator.mergeTupleIntoGroup(Tuple tup)的实现思路是这样的：

* 根据构造器给定的aggregateField获取在tup中的聚合字段及其值；
* 根据构造器给定的groupField获取tup中的分组字段，如果无需分组，则为null；这里需要检查获取的分组类型是否正确；
* 根据构造器给定的aggregateOp进行分组聚合运算，对于MIN,MAX,COUNT,SUM，我们将结果保存在groupMap中，key是分组字段(如果无需分组则为null)，val是聚合结果；对于AVG，我们不能直接进行运算，因为整数的除法是不精确的，我们需要把所以字段值用个list保存起来，当需要获取聚合结果时，再进行计算返回。

#### **实现内容：**

**IntergerAggregator类**

* public IntegerAggregator(int gbfield, Type gbfieldtype, int afield, Op what):初始化方法，根据不同的聚合操作使用aggHandler对应的子类

public void mergeTupleIntoGroup(Tuple tup)：聚合操作的执行过程是：先读取一个tuple进行聚合操作，得到一个只聚合了一个tuple的聚合结果，之后每读取一个tuple就将其加入到聚合结果中重新进行聚合

* public OpIterator iterator()：返回聚合结果的迭代器。结果集中的每个元组都有(groupValue, aggregateValue)两个字段。当 group by 字段的值是Aggregator.NO\_GROUPING时，结果中的元组只有(aggregateValue)一个字段。返回结果的类型是OpIterator 可以用TupleIterator对结果进行封装

**StringAggerator类**

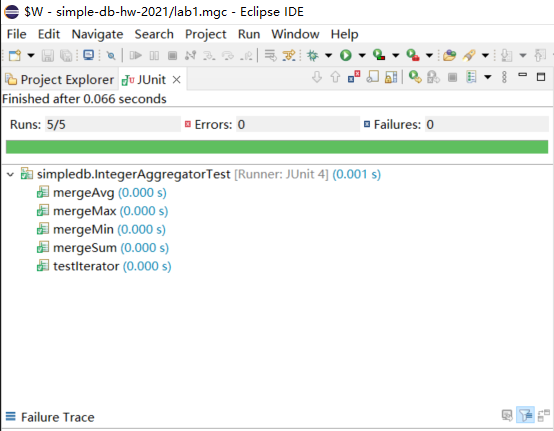
因为是对String类型进行聚合，所以只有count操作，还是老样子实例化的是groupby的字段位置、要聚合的数据的位置、要进行 的聚合操作、groupby的字段的数据类型，已经存储分组的groupMap。当不需要分组时直接返回总的数值，需要分组时则返回分组 值和每组的个数。

**Aggerate类**

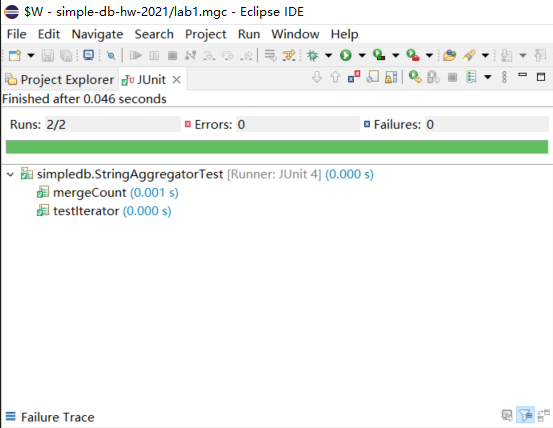
对IntegerAggregator.java 和 StringAggregator.java联合使用，通过输入一串的tuple，通过判断gfieldType来判断是用Interger聚合还是String聚合，创建完聚合对象后，将tuple一个一个的通过mergeTupleIntoGroup方法传进去，然后生成迭代器进行迭代。

* public Aggregate(OpIterator child, int afield, int gfield, Aggregator.Op aop)：初始化方法，在初始化方法中得到聚合结果的TupleDesc 方便后序使用
* public int groupField()：返回分组字段的序号，如果没有分组字段则返回Aggerator.NO\_GROUPING
* public String groupFieldName()：返回分组字段的字段名
* public int aggregateField()：返回待聚合字段的序号
* public String aggregateFieldName()：返回待聚合字段的字段名
* public Aggregator.Op aggregateOp()：返回aop
* public void open()：open时要根据aggFieldType得到响应的Aggerator，然后通过Aggerator进行聚合操作，用aggIterator保存聚合后的结果
* public void close()
* protected Tuple fetchNext()：返回aggIterator中的tuple
* public void rewind()
* public TupleDesc getTupleDesc()：返回聚合后的结果集的TupleDesc
* public OpIterator[] getChildren()
* public void setChildren(OpIterator[] children)

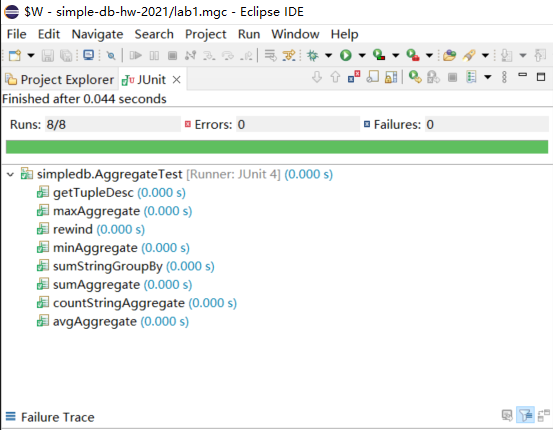
#### **测例：通过本Exercise全部测例**



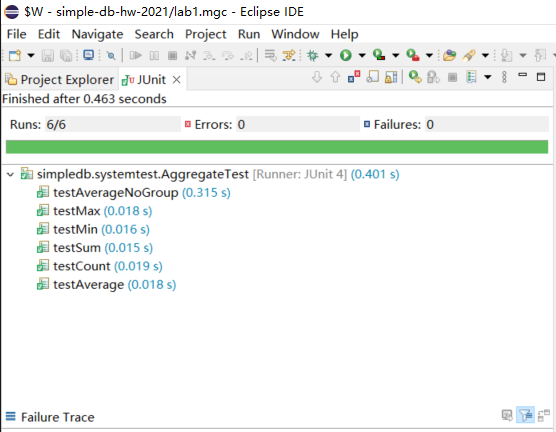
IntegerAggregatorTest



StringAggregatorTest



AggregateTest



system test AggregateTest

### 2.3. Exercise 3 Heap File Mutability

#### **本Exercise实验内容：**

实现HeapPage、HeapFile、BufferPool 中的insertTuple、deleteTuple。实现对表的添加和删除。先从独立物理页和文件的层次开始，这有两组主要的操作符，adding tuples和removing tuples。

* Removing tuples: 要移除一个Tuple，需要实现deleteTuple。Tuples包含了RecordIDs，能帮助找到tuple存储的物理页，所以大致思路就是找到tuple所属的物理页，正确地修改物理页的headers。
* Adding tuples: HeapFile.java中的insertTuple方法负责添加一个Tuple到heap file。为了增加一个新的tuple到HeapFile，需要找到页中一个空闲的slot。如果HeapFile中不存在这样的页，需要创建一个新页，并添加新页到磁盘上的物理文件中。需要保证tuple中的RecordID正确地更新。



调用关系

#### **实现内容：**

**HeapPage类**

* public void deleteTuple(Tuple t)：删除page中的tuple，同时修改该slot对应的bitmap（通过markSlotUsed方法），表示该slot已为空
* public void insertTuple(Tuple t)：返回一个受到修改的Page，遍历当前HeapFile所有在BufferPool中的HeapPage，查看是否有空位，如果有空位，就将Tuple插入，并返回插入的Page。如果遍历都没有空位，就新建一个HeapPage并将HeapPage写入磁盘中，然后再通过BufferPool进行访问并插入。最后再返回该HeapPage。
* public void markDirty(boolean dirty, TransactionId tid)：修改脏页标志位
* public TransactionId isDirty()：判断该page是否为dirty page如果是则返回产生该脏页的事务id
* private void markSlotUsed(int i, boolean value)：修改page中的header，value == true 在第i位添加, value == false 在第i位删除

**HeapFile类**

**插入删除元组思路：**

* 调用HeapFile的insertTuple；
* HeapFile的insertTuple遍历所有数据页（用BufferPool.getPage()获取，getPage会先从BufferPool再从磁盘获取），然后判断数据页是否有空slot，有的话调用对应有空slot的page的insertTuple方法去插入页面；如果遍历完所有数据页，没有找到空的slot，这时应该在磁盘中创建一个空的数据页，再调用HeapPage的insertTuple方法进行插入；
* 插入的页面保存到list中并返回，表明这是脏页，后续会用到。在BufferPool中插入和删除元组；
* public void writePage(Page page)：将page写入磁盘的操作。
* public List<Page> insertTuple(TransactionId tid, Tuple t)：将tuple插入到HeapFile中的page中，如果HeapFile中的page都已经满了，则在HeapFile中创建一个新的page
* public ArrayList<Page> deleteTuple(TransactionId tid, Tuple t)：将HeapFile中某一page上的某一tuple 从page上删除

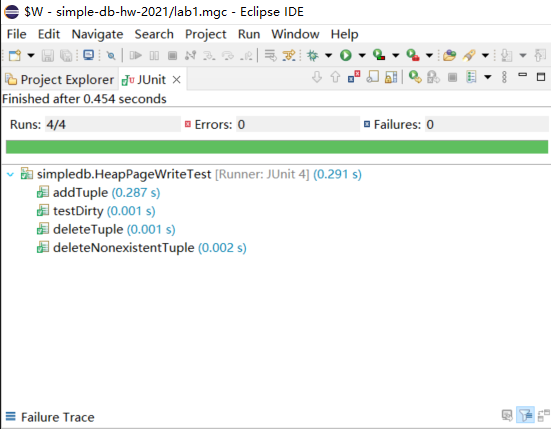
**BufferPool类**

* public void insertTuple(TransactionId tid, int tableId, Tuple t)：将tuple插入到指定的table（HeapFile）中，调用HeapFile中的insertTuple()方法，将返回的结果保存到BufferPool中
* public void deleteTuple(TransactionId tid, Tuple t)：调用HeapFile中的deleteTuple方法删除指定table中的tuple

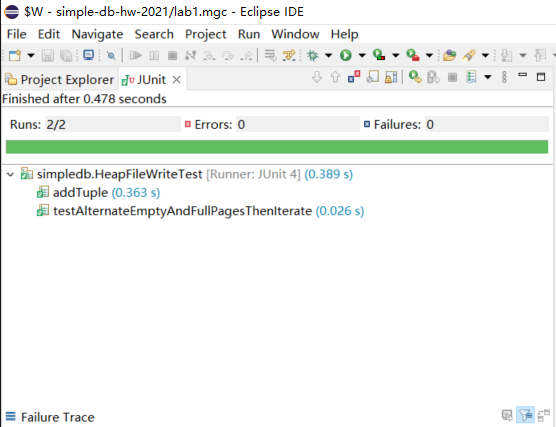
#### **问题：**

* 在做BufferPool的插入tuple时，由于没有刷新BufferPool的page，导致出错。测试用例一开始插入10个空页，然后分别往10个页面插入一条记录，然后用迭代器取数据并统计条数。由于迭代器都是先从BufferPool取page，再返回tuple，而BufferPool中的page是有的，只是都是老数据，前面插入tuple都是写入到page后再写到文件中，导致读到的是脏页的数据。正确的做法是把修改完成后的page写到文件后也将脏页写入缓存中。

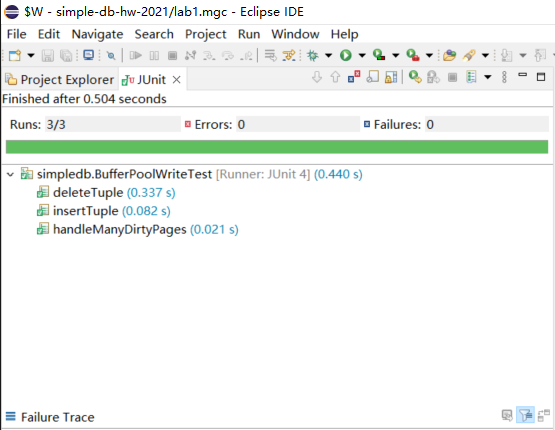
#### **测例：通过本Exercise全部测例**



HeapPageWriteTest



HeapFileWriteTest



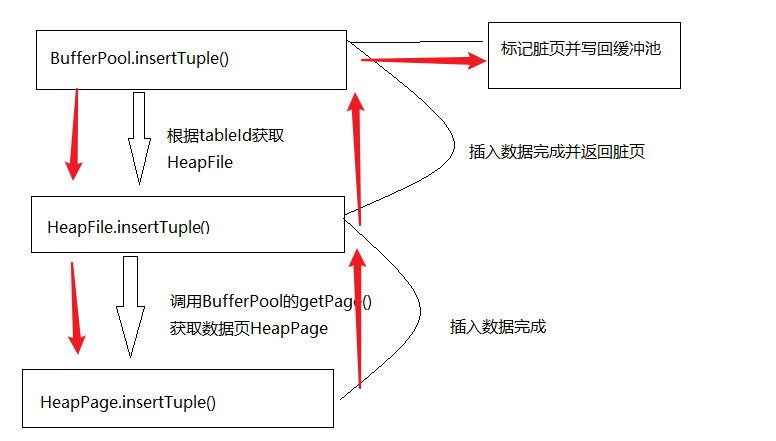
BufferPoolWriteTest

### 2.4. Exercise 4 Insertion and Deletion

#### **本Exercise实验内容：**

实现Insert类、Delete类，对exercise3进行封装即可。实现Insert 和 Delete操作符，最高的操作符是Insert和Delete，用于修改磁盘上的页数据，这些操作符返回受影响的tuples个数。

* Insert: 这个操作符添加 child operator读取的tuples 到 tableid 代表的表中，需要用到BufferPool.insertTuple()方法实现。
* Delete:这个操作符删除 child operator读取的tuples 到 tableid 代表的表中，需要用到BufferPool.deleteTuple()方法实现。



整体思路

#### **实现内容：**

**Insert类**

* public Insert(TransactionId t, OpIterator child, int tableId)：初始化方法，把从child operator中读取到的tuples添加到tableId对应的表中。
* protected Tuple fetchNext()：

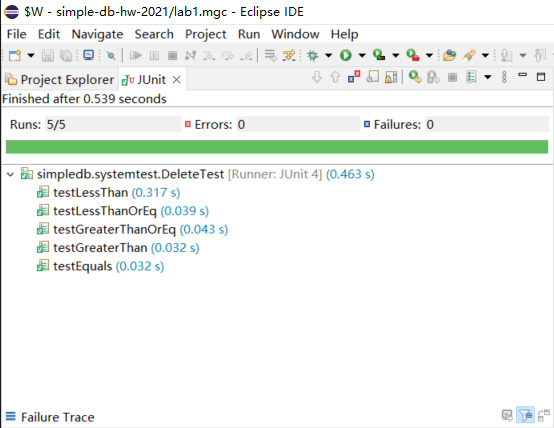
执行插入操作，返回包含插入了多少tuple的一个tuple。利用OpIterator类型的迭代器child找到一组要添加的记录，insert需要经过BufferPool，所以使用Database.getBufferPool().insertTuple(this.tid, this.tableId, t)方法进行添加。

* public TupleDesc getTupleDesc()：返回tupleDesc

#### **问题：**

在写Insert操作符的时候，fetchNext实现思路少考虑了部分情况。因为要考虑到数据源一条记录都没有的情况；如果第一次插入，但是数据源是没有记录的，这时应该返回affectRows为0的tuple，而不应该是null，所有需要一个Boolean变量call来表示这是否是第一次调用。

#### **测例：通过本Exercise全部测例**



system test DeleteTest

### 2.5. Exercise 5 Page Eviction

#### **本Exercise实验内容：**

实现BufferPool中的页面置换策略。该exercise中实现了BufferPool中的页面置换策略，采取的页面置换策略是LRU（最近最少使用）。Lab1中用的是HashMap<PageId,Page> bufferPool保存的pageId与Page的映射关系，加入页面置换策略以后修改了HashMap的映射结构，改用了HashMap<PageId,LinkedNode> bufferPool，LinkedNode是自定义的链表节点，节点内保存了pageID和page，以及前后两个节点prev、next。用LinkedNode构建一个链表，每当BufferPool中的page被访问时，将该pageId对应的LinkedNode移动到链表的头部，当有page需要放置到BufferPool中，且BufferPool的容量已经满时，则将最近最少使用的page淘汰，即链表的最后一个节点，然后将该page放置到BufferPool中。

#### **实现内容：**

**BufferPool类**

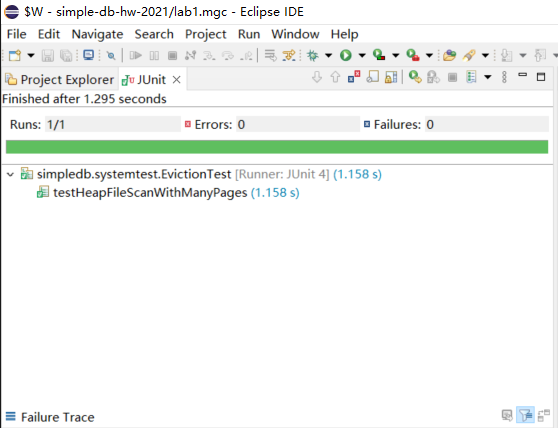
* public Page getPage(TransactionId tid, PageId pid, Permissions perm)：当从BufferPool中读取page时，若BufferPool中有要读取的page则将其对应的LinkedNode移动到头部，若没有则去硬盘中读取，并将其生成LinkedNode放在链表的头部，如果BufferPool已满，则调用evictPage()置换BufferPool中的页面
* private synchronized void evictPage()：页面置换操作
* private synchronized void flushPage(PageId pid)：如果移除的page是dirty page则将其写回磁盘
* public synchronized void discardPage(PageId pid)：将pid所对应的page从映射结构中移除

#### **问题：**

迭代器的实现：需要可以遍历所有的tuple，而tuple分布在多个page上，故需要在next的时候进行一定的处理。自己不熟悉java的iterator，写的时候逻辑混乱，最后整理逻辑后得到如下结果。

* 调用next的时候，都会调用hasNext去探探路，所以逻辑只需要放在hasNext中即可。
* 迭代器初始化的时候不指向元素，跟c++不太一样。
* 迭代器中的读取需要通过buffer，不能直接使用HeapFile读取，因为这样无法利用cache。

#### **测例：通过本Exercise全部测例**



system test EvictionTest

## 3. Lab3 Optimizer

#### **实验内容：**

**基于代价的优化器：**

CBO: Cost-Based Optimization也即“基于代价的优化器”，该优化器通过根据优化规则对关系表达式进行转换，生成多个执行计划，然后CBO会通过根据统计信息(Statistics)和代价模型(Cost Model)计算各种可能“执行计划”的“代价”，即COST，从中选用COST最低的执行方案，作为实际运行方案。

CBO依赖数据库对象的统计信息，统计信息的准确与否会影响CBO做出最优的选择。

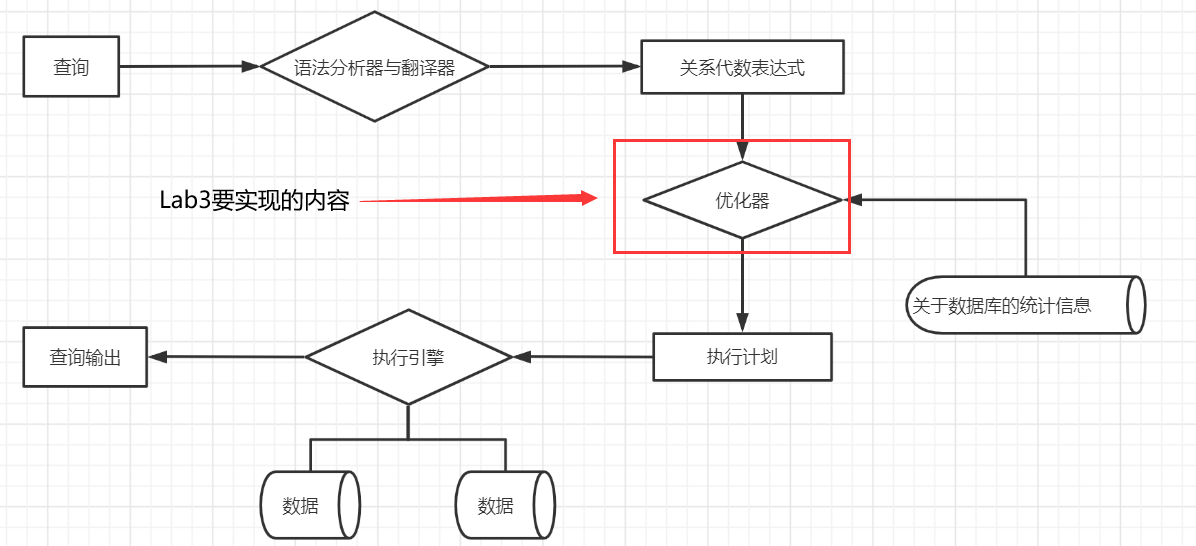
CBO的主要流程（本Lab的主要内容）：

* 利用关于表的统计数据，来估计不同查询计划的cost。

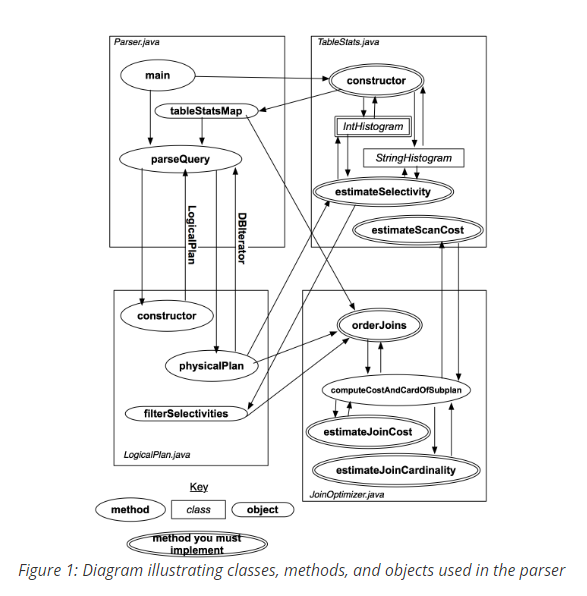
（cost与join、selection的基数、filter的选择率和join的谓词有关。）

* 使用数据来对join和select进行排序，并选择最佳的实现方式；

lab3实现的是基于代价的查询优化器，可以使用表的统计数据去估计不同查询计划的代价。通常，查询计划的代价与中间进行连接和选择产生的记录数的基数，以及过滤和连接的选择性有关。通过这些统计信息，我们可以选择最佳的连接和选择顺序，从多个查询方案中选择一个最佳的计划去执行。



优化器位置



优化器整体结构

TableStats类：对指定表中的数据进行统计，对表中的每一字段构建直方图

（IntHistogram、StringHistogram就是exercise1中要完成的，它能根据给定的字段和选择谓词，计算出选择率，即estimateSelectivity。exercise2要根据给定的tableId完成对TableStats的初始化）

Parser类：查询解析器，当有查询输入时，调用parseQuery方法对查询进行解析。

LogicalPlan类：实例代表解析后的查询，调用physicalPlan方法返回给Parser类一个最优的查询计划。

JoinOptimizer类：选择出最优的查询计划，orderJoin方法根据不同的连接顺序所产生的代价，选择出连接代价最小的查询计划。（exercise3、4要实现的类）。

**查询优化器的执行流程：**

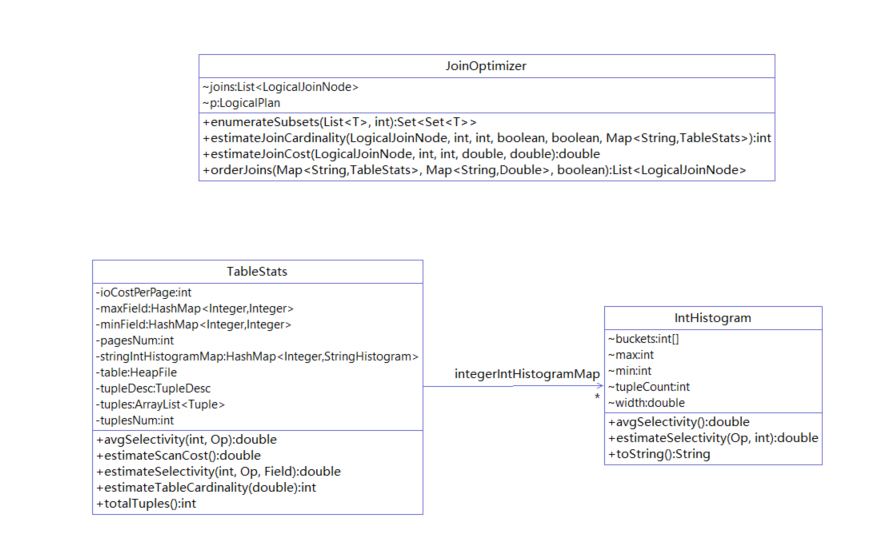
* Parser.Java在初始化时会收集并构造所有表格的统计信息，并存到statsMap中。当有查询请求发送到Parser中时，会调用parseQuery方法去处理.
* parseQuery方法会把解析器解析后的结果去构造出一个LogicalPlan实例，然后调用LogicalPlan实例的physicalPlan方法去执行，然后返回的是结果记录的迭代器，也就是我们在lab2中做的东西都会在physicalPlan中会被调用。

**成本估计：**

本Lab中，只关注表的连接成本和访问成本。不去考虑访问方法（Lab中只使用了一种访问方法，表扫描）和额外操作符（如聚合操作符）的成本。

* 扫描成本：对于本实验，您可以忽略缓存的影响（例如，假设对表的每次访问都会产生扫描的全部成本）。因此：
* 连接成本：
* 其中，ntups(t1)是表t1中的tuples数。

#### **实验类图：**

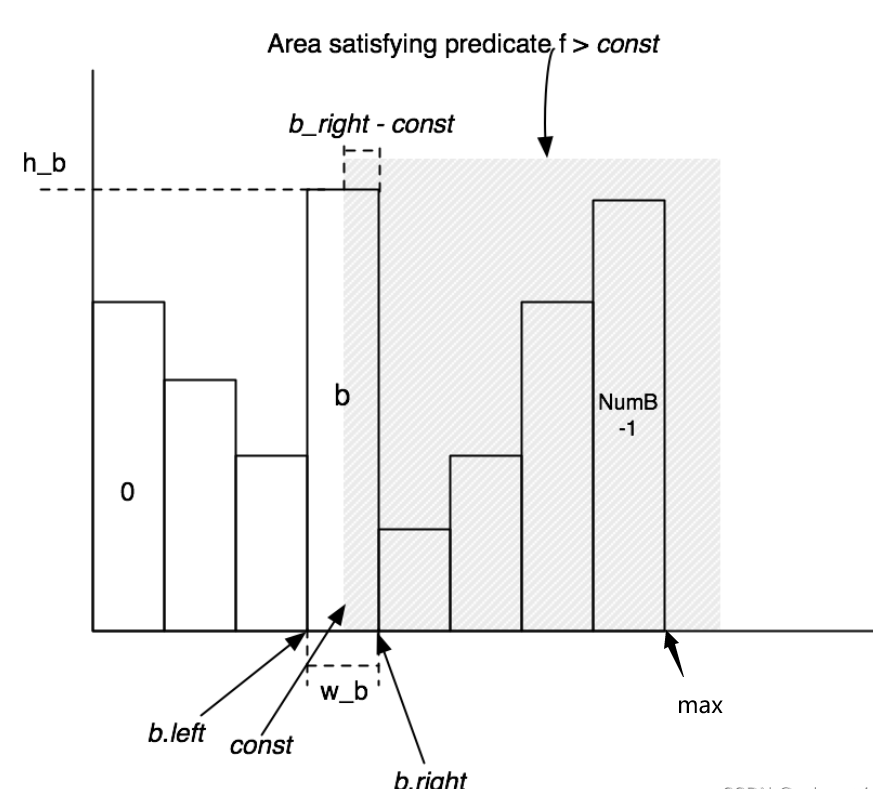


lab3类图

### 3.1. Exercise 1 and 2 Filter Selectivity

#### **本Exercise1实验内容：**

实现IntHistogram。估计查询计划的代价首先要获得数据的统计数据，本lab采用直方图对数据进行统计。一个直方图代表着一个字段的统计信息，直方图将字段的值分为多个相同的区间，并统计落于每个区间的记录数。每个区间的记录数是一个bucket，bucket的宽度是区间的大小，bucket的高度是落于该区间的记录的数量。



示例

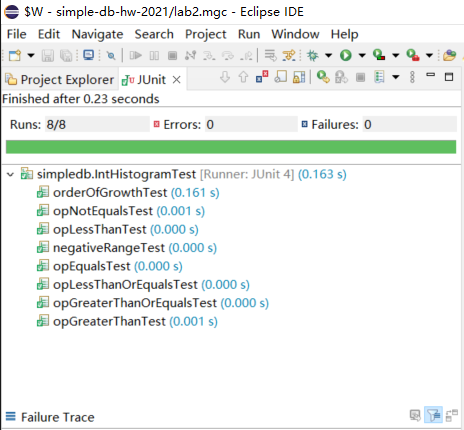
#### **实现内容：**

**IntHistogram类**

实现的是数值类型的直方图选择性估计，那么在构造函数中就需要实例化 最大值、最小值（为了算每个桶有多宽）以及桶的个数。

* IntHistogram(int buckets, int min, int max)：构造函数，初始化桶的数量、最大值、最小值
* int getIndex(int value)：根据value获得桶的序号
* void addValue(int v)：向直方图中添加数据
* double estimateSelectivity(Predicate.Op op, int v)：返回指定判断条件下（谓词+值）的选择率。

#### **测例：通过本Exercise1全部测例**



IntHistogramTest

#### **本Exercise2实验内容：**

实现TableStats类，根据给定的tableid，扫描出所有记录，并对每一个字段建立一个直方图。

* 扫描全表，确定每个字段的最大最小值；并且根据最大最小值、表的桶数，构造出每个字段的直方图。
* 再次扫描全表，并在扫描过程中填充数据。

#### **实现内容：**

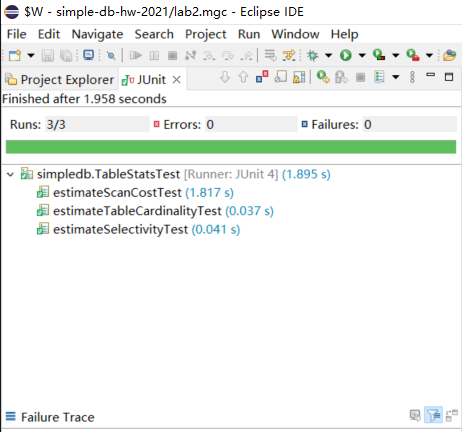
**TableStats类**

* TableStats(int tableid, int ioCostPerPage)：给定tableId构建其各个字段的直方图。一次扫描，第一次扫描统计各个字段的最大值和最小值，同时将表中的记录保存到链表tuples中，然后遍历tuples中的记录生成各个字段的直方图
* double estimateScanCost()：估计扫描成本
* int estimateTableCardinality(double selectivityFactor)：返回给定选择率下的基数。
* double avgSelectivity(int field, Predicate.Op op)：返回指定字段的平均选择率。
* double estimateSelectivity(int field, Predicate.Op op, Field constant)：返回给定字段的选择率，，返回扫描页面的花费时间，返回遍历整个表内Tuple的时间，返回未知字段类型的被选中可能性，返回已知字段类型的被选中可能性 以及 返回总的Tuple个数。

#### **问题：**

混淆了JoinNode和Table，以为代码里面想以JoinNode来进行排列。

#### **测例：通过本Exercise2全部测例**



TableStatsTest

### 3.2. Exercise 3 Join Cardinality

#### **本Exercise实验内容：**

实现JoinOptimizer类的estimateJoinCost、estimateJoinCardinality方法。实现对连接（也就是数据库中的关键词 join）进行连接花费的估计。

#### **实现内容：**

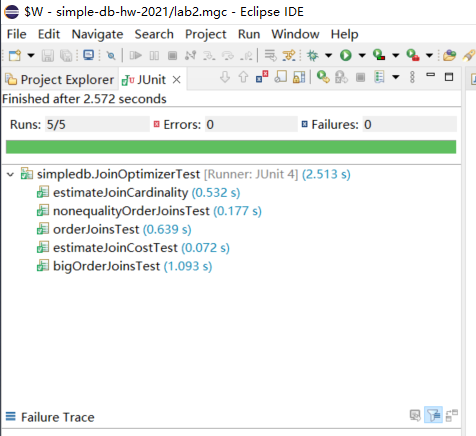
**JoinOptimizer类**

* public double estimateJoinCost(LogicalJoinNode j, int card1, int card2, double cost1, double cost2) ：估计连接成本，card1是左表的基数，cost1是扫描左表的成本，card2是右表的基数，cost2是扫描右表的成本。

成本估计：

* public static int estimateTableJoinCardinality()：估算两个表连接后的基数。
  + 对于等值连接：当一个属性是primary key，把non-primary key属性的记录数取做连接后的基数。因为主键是唯一的，也就是说非主键的每一条记录最多能连接一个主键的记录数。当两个属性都是primary key时，取字段中记录数较小的做基数。
  + 对于没有primary key的等值连接，很难估计连接结果的基数，可能是两表记录数的乘积，也可能 是0。本Lab中采用一种简单的估计方式，即连接后的结果的基数是是两表中较大的基数。
  + 对于范围扫描：基数也很难估计。本Lab采用两表基数乘积 \* 0.3 作为范围扫描的基数估计。

#### **测例：通过本Exercise全部测例**



JoinOptimizerTest

### 3.3. Exercise 4 Join Ordering

#### **本Exercise实验内容：**

完成JoinOptimizer类,在多表连接的情况下，根据连接的代价，选择出代价最小的连接顺序。采用一种用于连接次序优化的动态规划算法。核心是：分割成两个子集，然后选出两个子集中各自最佳的连接顺序，最后两个子集的最优结果去确定连接顺序。为了实现这个算法，exercise4给我们提供了两个很好用的方法：

* enumerateSubsets(List v, int size)：分割子集，给定集合和分割大小，返回指定大小子集的所有可能
* computeCostAndCardOfSubplan：计算子计划的代价

**算法伪代码如下：**

produe FindBestPlan(S)  
 if(bestplan[S].cost ≠ ∞ ) //bestplan[S]已经计算好了  
 return bestplan[S]  
 if(S中只包含一个关系)  
 根据访问S的最佳方式设置bestplan[S].plan和bestplab[S].cost  
 else  
 for each S 的非空子集S1，且S1≠S  
 P1 = FindBestPlan(S1)  
 P2 = FindBestPlan(S-S1)  
 A = 连接P1和P2的结果的算法 //嵌套循环连接  
 plan = 使用A对P1和P2进行连接的结果  
 cost = P1.cost + P2.cost + A的代价  
 if cost < bestplan[S].cost  
 bestplan[S].cost = cost  
 bestplan[S].plan = plan  
return bestplan[S]

#### **实现内容：**

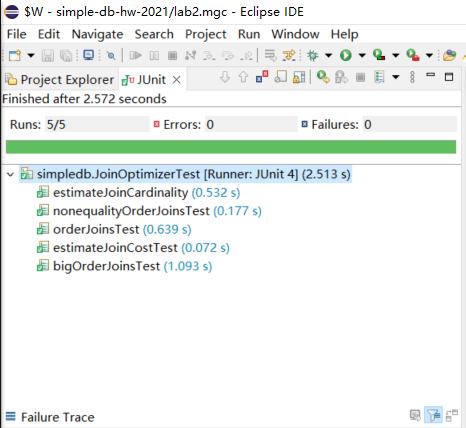
**JoinOptimizer类**

* public List orderJoins(Map<String, TableStats> stats,Map<String, Double> filterSelectivities, boolean explain)：给定各个表的统计数据，与各个表的选择率，返回joins的最优连接顺序。

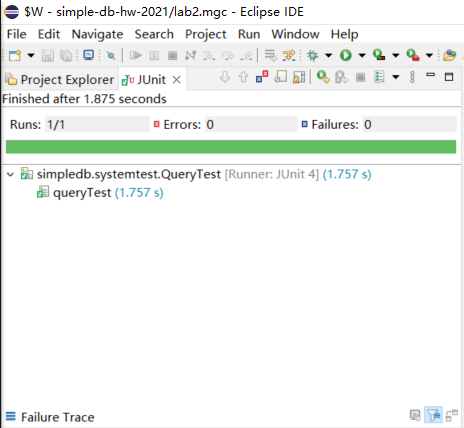
思路：

* 函数返回值是这个 List 实际上这是通过PlanCache类的getOrder方法进行排序后返回的List集合，PlanCache是一个辅助类，可以用来存储最好的方式来排列一组给定的连接。
* 然后根据joinOptimizer类中join的大小，定义循环的次数，通过enumerateSubsets函数枚举所有join的集合，然后针对每个子集中的LogicalJoinNode在computeCostAndCardOfSubplan方法中计算，然后找到花费最小的进行存储，并保存在planCache中。

#### **测例：通过本Exercise全部测例**



JoinOptimizerTest



system test QueryTest

## 4. Lab4 Transaction

#### **实验内容：**

lab4要做的是让SimpleDB支持事务，所以实验前需要对事务的基本概念有了解，并知道ACID的特点。lab4是基于严格两阶段封锁协议去实现原子性和隔离性的，所以开始前也需要了解两阶段封锁协议是如何实现事务的。对于一致性和持久性，这里假设暂时不会发送断电等异常，所以暂时不需要崩溃恢复，不需要undo log从，后面lab6会有专门的崩溃恢复的解决方案。

1. 两阶段封锁协议：

首先是封锁协议：我们将要求在系统中的每一个事务遵从封锁协议，封锁协议的一组规则规定事务何时可以对数据项们进行加锁、解锁。

对于两阶段封锁协议：两阶段封锁协议要求每个事务分两个节点提出加锁和解锁申请：

* 增长阶段：事务可以获得锁，但不能释放锁；
* 缩减阶段：事务可以释放锁，但不能获得新锁。

最初，事务处于增长阶段，事务根据需要获得锁。一旦该事务释放了锁，它就进入了缩减阶段，并且不能再发出加锁请求。

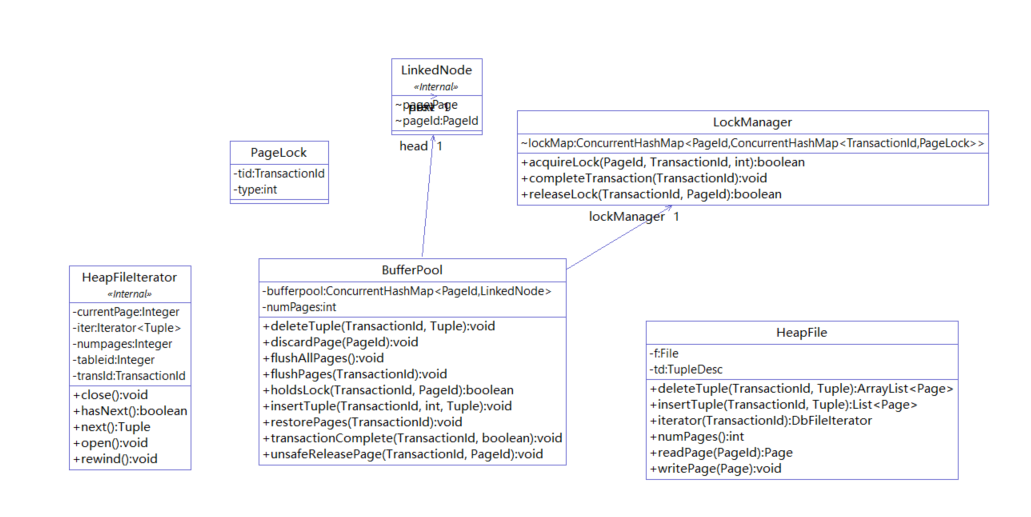
严格两阶段封锁协议不仅要求封锁是两阶段，还要求事务持有的所有排他锁必须在事务提交后方可释放。这个要求保证未提交事务所写的任何数据在该事务提交之前均已排他方式加锁，防止了其他事务读这些数据。

1. 强两阶段封锁协议。它要求事务提交之前不释放任何锁。在该条件下，事务可以按其提交的顺序串行化。
2. 锁转换：在两阶段封锁协议中，我们允许进行锁转换。我们用升级表示从共享到排他的转换，用降级表示从排他到共享的转换。锁升级只能发送在增长阶段，锁降级只能发生在缩减阶段。

具体包括以下内容：

* exercise1、2 ：实现一个page级别，遵从二段锁协议的锁管理器。即在访问任何page之前，事务应该获取该page上适当类型的锁，并且在事务提交之前不应该释放任何锁。
* exercise3 ：完善BufferPool中的evictPage()方法，避免数据丢失，当需要置换的页面为脏页时，要跳过脏页，置换掉不是脏页的page。
* exercise4 ：实现事务的功能，当事务提交时，将事务涉及的脏页写回磁盘，然后释放锁。当事务回滚时，清理该事务涉及到的脏页，重新从磁盘中读取清理的page。
* exercise5：实现死锁防范功能，当发生死锁时，抛出AbortException。

#### **实验类图：**



lab4类图

### 4.1. Exercise 1 Granting Locks

#### **本Exercise实验内容：**

在BufferPool中编写获取和释放锁的方法 ,构造一个page级别的锁管理器，在BufferPool中构造两个类，PageLock类和LockManager类。当事务通过BufferPool访问page时，需要通过LockManager申请锁。

#### **实现内容：**

**PageLock类**

private class PageLock{//构造函数  
 public static final int SHARE = 0;  
 public static final int EXCLUSIVE = 1;  
 private TransactionId tid;  
 private int type;  
  
 public PageLock(TransactionId tid, int type){  
 this.tid = tid;  
 this.type = type;  
 }  
  
 public TransactionId getTid() {  
 return tid;  
 }  
  
 public int getType() {  
 return type;  
 }  
  
 public void setType(int type) {  
 this.type = type;  
 }  
 }

**LockManager类**

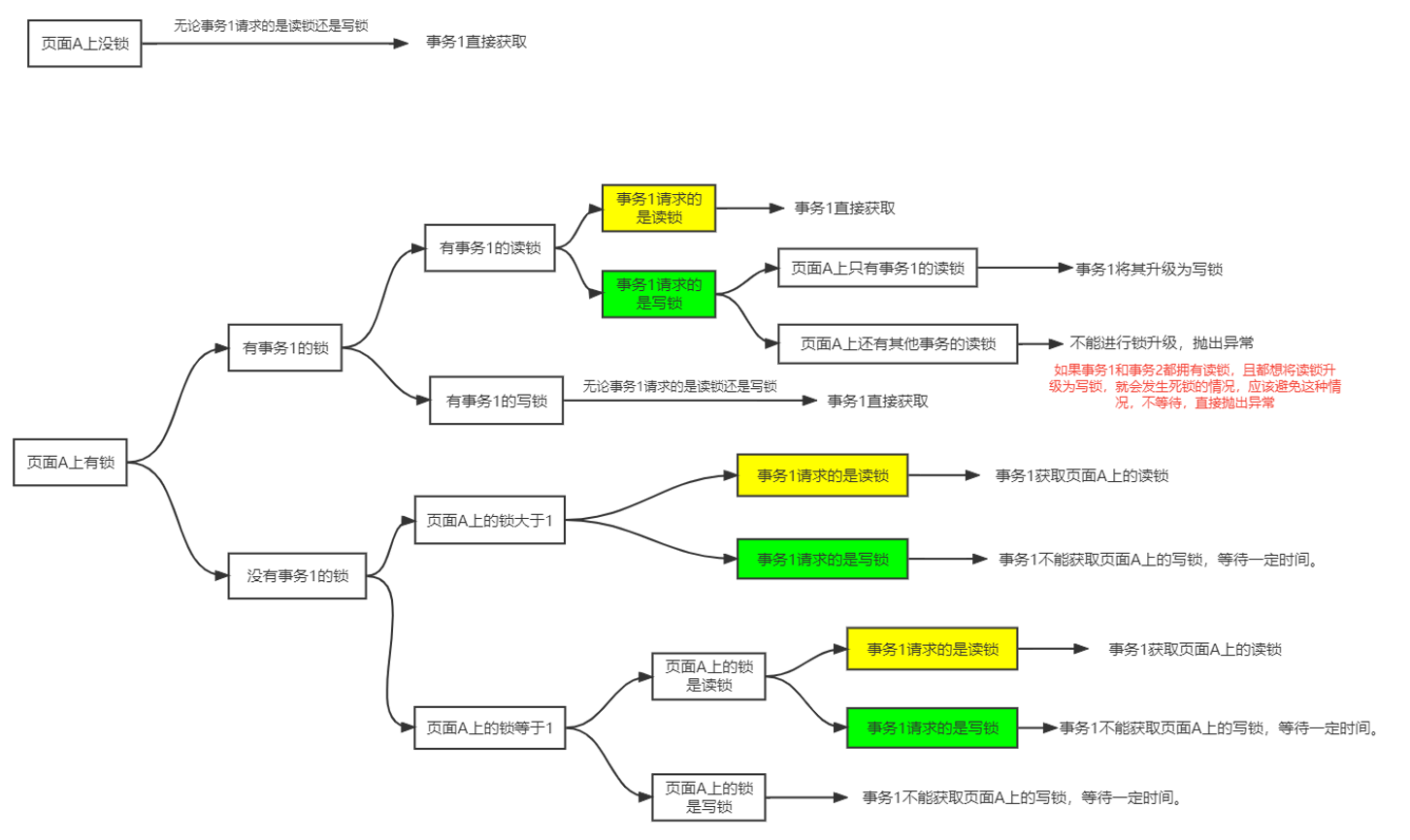
* public synchronized boolean acquireLock(PageId pageId, TransactionId tid, int requiredType)：事务向LockManager请求获取page上锁的方法。

**思路如下：**

1. 首先，我们获取锁时会获得当前事务的ID，要上锁的页面ID，锁的类型。
2. 我们需要在我们的锁表（lockMap）中查找看看是否，该页面已经上锁，如果没有上锁，根据事务ID创建传进来的锁，并存入锁表中，并返回True。如果该页面已经有锁则进入第三步。
3. 获取该页面的所有锁，找到等于当前事务的锁，看他是否满足如下情况：

* 是否跟当前锁的类型一致，如果一致就可以获取锁。
* 如果不一致，如果存在的锁是排他锁，那说明请求的锁是共享锁，也是可以直接获取的。这里涉及到一个知识点，\*\*同一事务可以不断对某个数据对象加锁，不需要等锁的释放。\*\*因为同一事务中所有操作都是串行化的，所以不会产生影响。
* 如果不一致，如果存在的锁是共享锁，那说明请求的锁是排他锁，这里就涉及到一个锁升级的概念，如果当同一事务中只有共享锁，但是即将需要上排他锁时，此时可直接将共享锁升级为排他锁，这里的思想跟意向锁很想，同时官方文档中也提到了。

1. 如果当前事物在当前页面没有上过锁，那就看看该页面第一个锁是不是排他锁，如果是，按照封锁协议，其他事物都是不能加锁的，所以return False。
2. 同时如果当前页面上的锁都是共享锁，如果需要加的也是共享锁也就可以直接加上 return True，但是如果需要加上排他锁，根据共享锁的性质，排他锁将不被允许加上。



**申请思路导图**

* public synchronized boolean isholdLock(TransactionId tid, PageId pid)：判断指定事务是否持有某一page上的锁，判断当前事务是否在该页面有锁，那就直接遍历就好了，然后注意一下空值。
* public synchronized boolean releaseLock(TransactionId tid, PageId pid)：释放指定事务在某一page上的所有锁

基本思路：

* 如果锁表中没有当前页面没有锁，就直接返回 True。
* 如果该页面有当前事务的锁，那就挨个remove掉，并且最后如果这个页面没锁了就在锁表中也直接remove掉。当前事务不在当前页面有锁则返回False。
* public synchronized void completeTransaction(TransactionId tid)：事务执行完成，释放该事务在所有page上的锁

#### **问题：**

开始本Exercise需要提前梳理清楚各种可能情况以及对于处理，开始实验时部分情况考虑不周导致错误。

#### **测例：本Exercise无测例**

### 4.2. Exercise 2 Lock Lifetime

#### **本Exercise实验内容：**

主要是让我们检查一下前面申请锁的时候，使用getPage方法获取数据页时申请锁，传入的锁类型是否正确。

#### **实现内容：**

**HeapFile类**

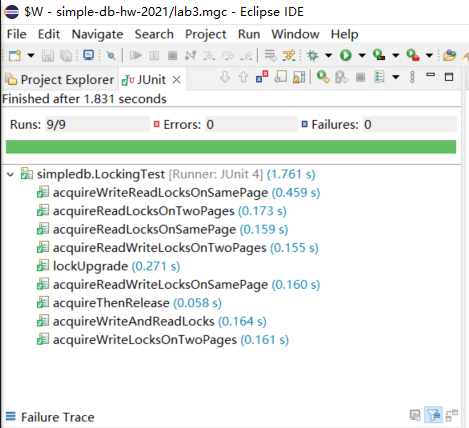
* public List insertTuple(TransactionId tid, Tuple t) : 完善HeapFile中的InsertTuple方法

**主要思路：**

* 如果不能就会超时，然后回溯（回溯是在getPage中进行）。
* 如果能上锁就判断page在不在当前缓冲区中。
* 如果不在就从磁盘里读出来（同时如果Buffer pool满了的话，要采用驱逐策略将页面驱逐），将页面放到pageQueue的队尾。
* 如果在就从缓冲区读出来，并把当前页送到pageQueue的队尾。传回来的page在insertTuple函数中判断有没有空值，如果没有进行下一个页面（并且把上了的锁清除），如果有就将tuple插入。如果遍历完都没能插入tuple，则创建新页插入table中。

*Note: 当我们要插入一个元组时，会去找到一个有空的slot的page，但是当我们获取的page没有空的slot了，我们应该立即释放在这个page的锁，即使这样会不符合严格二阶段封锁协议，但后续我们不会再使用到这个page了，所以并没有影响，这样能够让其它事务可以访问该page*

#### **测例：通过本Exercise全部测例**



LockingTest

### 4.3. Exercise 3 Implementing NO STEAL

#### **本Exercise实验内容：**

实现NO STEAL策略, 事务对page的修改只有在commit之后才会写入到磁盘，但是在之前的Lab中实现页面置换策略时，当置换掉的页面是dirty page时，也会将更改写回到磁盘。这是不允许的，所以需要完善evictPage()方法，当需要置换的page是dirty page时，需要跳过此page，去置换下一个非dirty的page。当BufferPool中缓存的page都是dirty page时，抛出异常。

#### **实现内容：**

**BufferPool类**

* public void deleteTuple(Tuple t)：删除page中的tuple，同时修改该slot对应的bitmap（通过markSlotUsed方法），表示该slot已为空

**测例：本Exercise无测例**

### 4.4. Exercise 4 Transactions

#### **本Exercise实验内容：**

实现事务：

在SimpleDB中，每个事务都会有一个Transaction对象，我们用TransactionId来唯一标识一个事务，TransactionId在Transaction对象创建时自动获取。事务开始前，会创建一个Transaction对象，后续的操作会通过传递TransactionId对象去进行，加锁时根据加锁页面、锁的类型、加锁的事务id去进行加锁。当事务完成时，调用transactionComplete去完成最后的处理。transactionComplete会根据成功还是失败去分别处理，如果成功，会将事务id对应的脏页写到磁盘中，如果失败，会将事务id对应的脏页淘汰出bufferpool或者从磁盘中获取原来的数据页。脏页处理完成后，会释放事务id在所有数据页中加的锁。

实现BufferPool中的transactionComplete方法。transactionComplete() 有两个版本，一个接受额外的布尔参数（当布尔参数为true时，进行提交。false时进行回滚），另一个不接受。没有附加参数的版本应该总是提交，因此可以简单地通过调用来实现 transactionComplete(tid, true)。当进行回滚时，从BufferPool中清除掉该事务造成的脏页，并将原始版本重新读到BufferPool中。

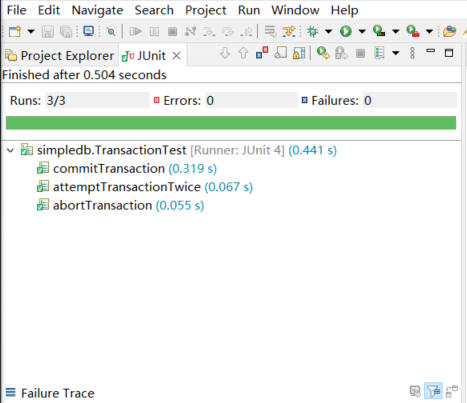
#### **实现内容：**

**BufferPool类**

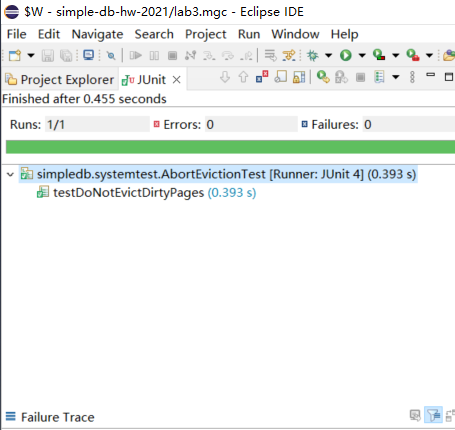
* public void transactionComplete(TransactionId tid)；
* public void transactionComplete(TransactionId tid, boolean commit) ；

public void transactionComplete(TransactionId tid){  
 // some code goes here  
 // not necessary for lab1|lab2  
 transactionComplete(tid,true);  
 }  
  
 public void transactionComplete(TransactionId tid, boolean commit) {  
 // some code goes here  
 // not necessary for lab1|lab2  
 if(commit){  
 try {  
 flushPages(tid);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 else {  
 restorePages(tid);  
 }  
 lockManager.completeTransaction(tid);  
 }  
  
 public synchronized void restorePages(TransactionId tid){  
 for(LinkedNode node : bufferPool.values()){  
 PageId pageId = node.getPageId();  
 Page page = node.getPage();  
 if(tid.equals(page.isDirty())){  
 int tableId = pageId.getTableId();  
 DbFile table = Database.getCatalog().getDatabaseFile(tableId);  
 Page pageFromDisk = table.readPage(pageId);  
  
 node.setPage(pageFromDisk);  
 bufferPool.put(pageId,node);  
 moveToHead(node);  
 }  
 }  
 }

#### **测例：通过本Exercise全部测例**



TransactionTest



system test AbortEvictionTest

### 4.5. Exercise 5 Deadlocks and Aborts

#### **本Exercise实验内容：**

发生死锁：

* 如果两个事务t0,t1，两个数据页p0,p1，t0有了p1的写锁然后申请p0的写锁，t1有了p0的写锁然后申请p1的写锁，这个时候会发生死锁；
* 如果多个事务t0,t1,t2,t3同时对数据页p0都加了读锁，然后每个事务都要申请写锁，这种情况下只能每一个事务都不可能进行锁升级，所以需要有其中三个事务进行中断或者提前释放读锁，由于我们实现的是严格两阶段封锁协议，这里只能中断事务让其中一个事务先执行完。

解决死锁：

* 超时。对每个事务设置一个获取锁的超时时间，如果在超时时间内获取不到锁，我们就认为可能发生了死锁，将该事务进行中断。
* 循环等待图检测。我们可以建立事务等待关系的等待图，当等待图出现了环时，说明有死锁发生，在加锁前就进行死锁检测，如果本次加锁请求会导致死锁，就终止该事务。

本实验采用方案一。

#### **实现内容：**

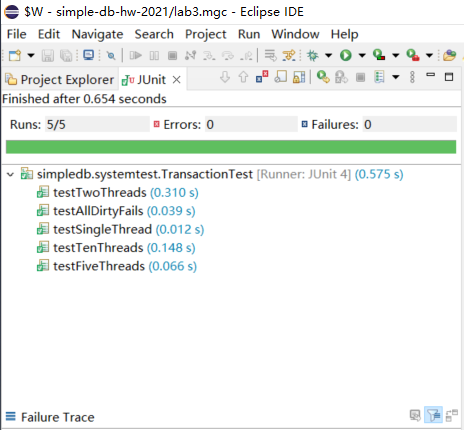
**BufferPool类**

* public synchronized Page getPage(TransactionId tid, PageId pid, Permissions perm):当事务通过BufferPool的getPage()方法获取page时，先确定该事务需要获取锁的类型。通过LockManager获取响应类型的锁，成功获取则正常读取页面。获取失败时，执行acquireLock()方法的线程会进行超时等待，等待超时或者被其他线程唤醒后会再次尝试获取锁，等待时间超过500ms时，对该事务进行回滚并抛出异常，避免死锁的发生。

#### **问题：**

多线程调试的问题，前面lab2挖的坑在这里没发现，导致调了好久好久，整个过程不断重复加锁的思路，然后不断完善日志信息，调代码，有时候一些情况需要多试几遍才会出现，这也导致调代码的困难。

#### **测例：通过本Exercise全部测例**



system test TransactionTest

## 5. Lab5 B+ Tree Index

#### **实验内容：**

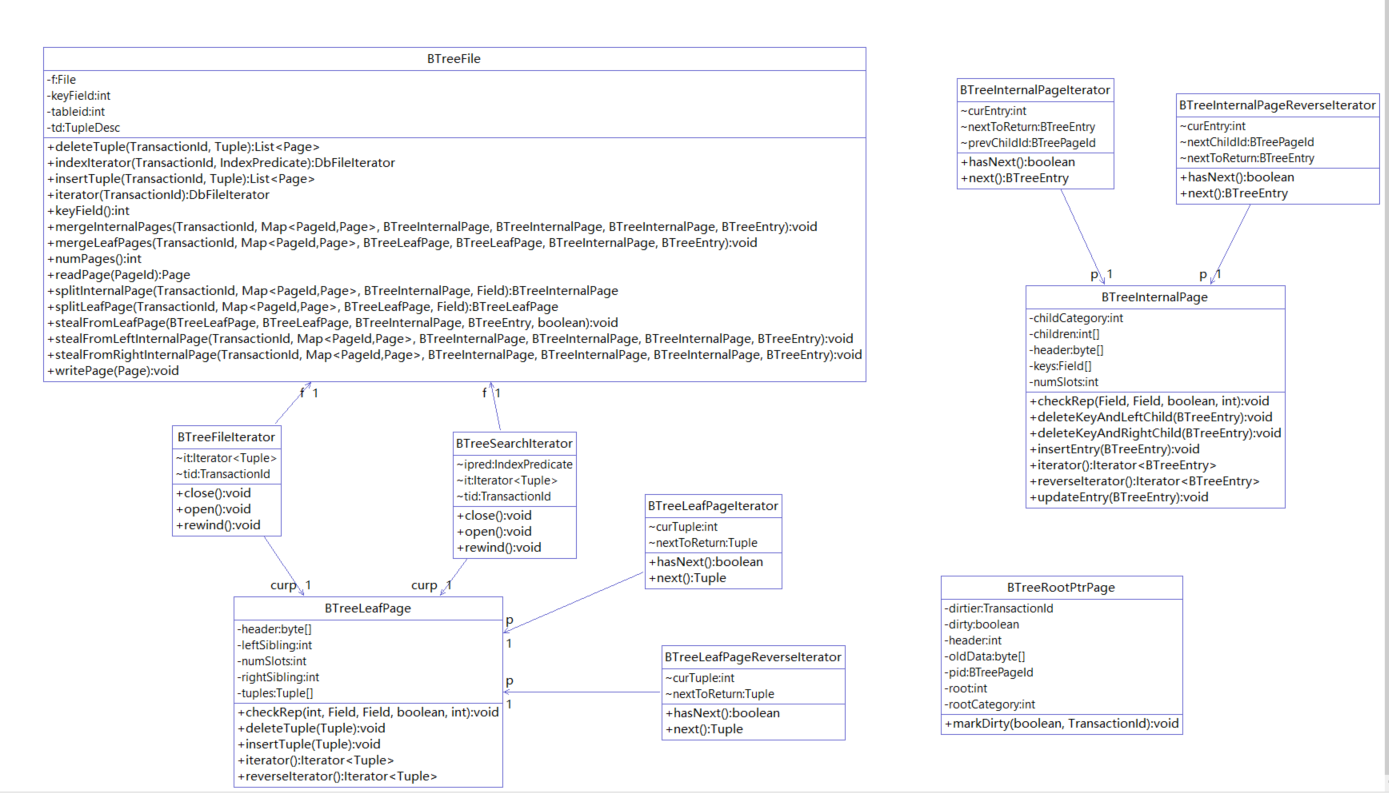
B+树的页面节点类型主要有四种：

1. 根节点页面：一个B+树的根节点，在SimpleDB中实现为BTreeRootPtrPage.java;
2. 内部节点页面：除去根节点和叶子节点外的节点，在SimpleDB中实现为BTreeInternalPage，每个BTreeInternalPage由一个一个的entry组成；
3. 叶子节点页面：存储tuple的叶子节点，在SimpleDB中实现为BTreeLeafPage；
4. 头部节点页面：用于记录整个B+树中的一个页面的使用情况，在SimpleDB中实现为BTreeHeaderPage。

lab5主要是实现B+树索引，主要有查询、插入、删除等功能，查询主要根据B+树的特性去递归查找即可，插入要考虑节点的分裂（节点tuples满的时候），删除要考虑节点内元素的重新分配（当一个页面比较空，相邻页面比较满的时候），兄弟节点的合并（当相邻两个页面的元素都比较空的时候)。

* exercise1 ：实现B+树的搜索，根据给定的key查找适当的页节点。
* exercise2 ：实现内部节点、页节点的拆分，当页面中key的数量大于n-1时，对页面进行拆分。
* exercise3 ：实现节点的重新分配，当删除key后如果页面中key的数量小于m/2 时，从其兄弟节点“窃取”一个key
* exercise4：实现节点的合并，当删除key后如果页面中key的数量小于m/2 时，且兄弟节点也只有m/2个key，则将两个节点合并。

#### **实验类图：**



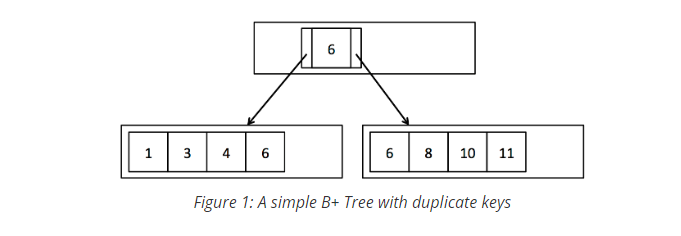
lab5类图

### 5.1. Exercise 1 Search

#### **本Exercise实验内容：**

给定一个field和一个page，要从这个page往下递归找到tuple在的叶子节点。主要思路如下：

* 获取数据页类型；
* 判断该数据页是否为叶子节点，如果是则递归结束，将该页面返回；
* 如果不是则说明该页面是内部节点，将页面进行类型转换；
* 获取内部节点的迭代器；
* 对内部节点的entry进行迭代，这里要主要field是空的处理，如果是空直接找到最左的叶子页面即可；
* 找到第一个大于（或等于）filed的entry，然后递归其左孩子；
* 如果到了最后一个页面，则递归其右孩子；



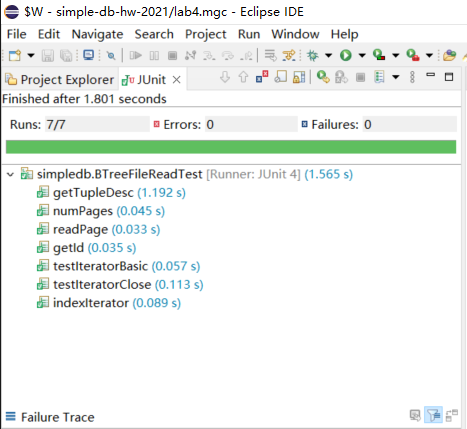
B+树查找

#### **实现内容：**

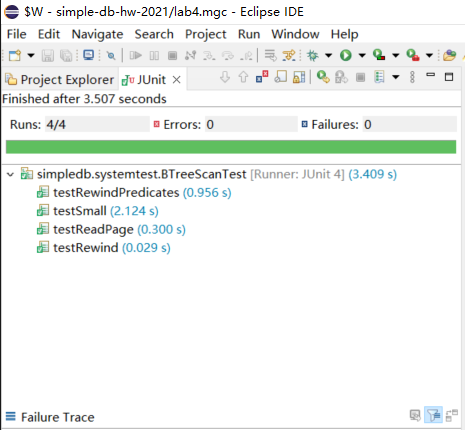
**BTreeFile类**

* private BTreeLeafPage findLeafPage(TransactionId tid, Map<PageId, Page> dirtypages, BTreePageId pid, Permissions perm,  
  Field f)

#### **测例：通过本Exercise全部测例**



BTreeFileReadTest



system tests BTreeScanTest

### 5.2. Exercise 2 Insert

#### **本Exercise实验内容：**

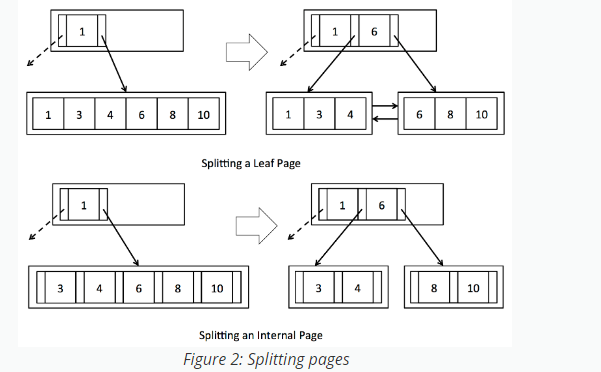
实现splitLeafPage()和splitInternalPage()方法。

分裂叶子节点的思路：

* 新建一个leaf page，作为新的页面；
* 将满页面的元组复制到新页面，边复制边删除；
* 检查之前的满页面是否有右兄弟，有的话需要更新指针，这里有点像在双向链表中插入一个结点，一开始没有考虑到，后面测试用例过不了重新整理思路才发现要更新这个指针；
* 更新脏页；
* 更新兄弟指针；
* 找出父节点并创建entry进行插入，最后更新脏页；
* 根据field找出要插入的页面并返回；

分裂内部节点的思路：

* 新建一个internal page，作为新的页面；
* 将满页面的entry复制到新页面，边复制边删除；
* 将中间节点挤出去;这里与leaf page不同，要注意，其实看图示就能发现不同之处了；
* 更新脏页；
* 更新左右孩子指针；
* 更新左右叶面的孩子指针，因为前面有大量的entry插入和移除；
* 根据中间节点获取父节点，将midEntry插入到父节点中，并更新脏页和指针；
* 根据field找出要插入的页面并返回



分裂示意图

#### **实现内容：**

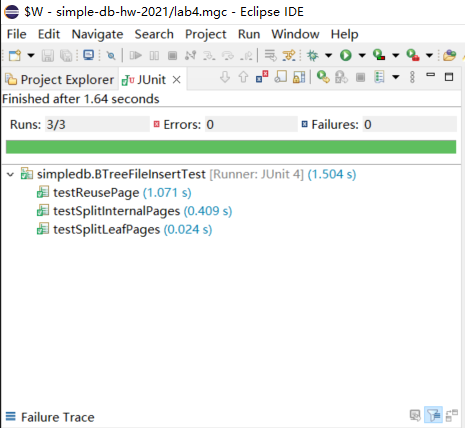
**BTreeFile类**

* public BTreeLeafPage splitLeafPage(TransactionId tid, Map<PageId, Page> dirtypages, BTreeLeafPage page, Field field) : 分裂叶子节点，思路如下：
  + 通过getEmptyPage创建一个newRightPage
  + 将当前page中一半的tuple插入到newRightPage中。插入时应该先从page中删除tuple，然后再插入到newRightPage。（newRightPage插入tuple后会给其赋值新的recordId，page删除tuple时根据其recordId进行查找然后删除，而page无法定位到被赋值了新recordId的tuple，则无法将其删除）。
  + 如果当前page有右兄弟oldRightPage，将oldRightPage左兄弟的指针指向newRightPage，将newRightPage的右兄弟指针指向oldRightPage。并将oldRightPage添加到dirtypages中。
  + 将page的右兄弟指针指向newRightPage，newRightPage的左兄弟指针指向page。将page、newRightPage添加到dirtypages中。
  + 获取指向该page的内部节点，在其中添加一个指向page和newRightPage的新entry。将父entry所在的page添加到dirtypages中
  + 更新page、newRightPage的父指针。
  + 返回field所在的页（page或newRightPage）
* public BTreeInternalPage splitInternalPage(TransactionId tid, Map<PageId, Page> dirtypages, BTreeInternalPage page, Field field) 分裂内部节点，思路如下：
  + 通过getEmptyPage创建一个newRightPage。
  + 将当前page中一半的entry插入到newRightPage中。同样，先从page中删除entry，再将其插入到newRightPage中。
  + 分配完entry后，选出page中最大的entry，将其从page中删除，并将该entry的左孩子指针指向page，右孩子指针指向newRightPage，获取父节点parent，将该entry添加到父节点中（实现将中间的key“挤到”父节点中）。
  + 将父节点parent、page、newRightPage添加到dirtypages中，并更新它们孩子节点的父指针。
  + 返回field所在的页（page或newRightPage）

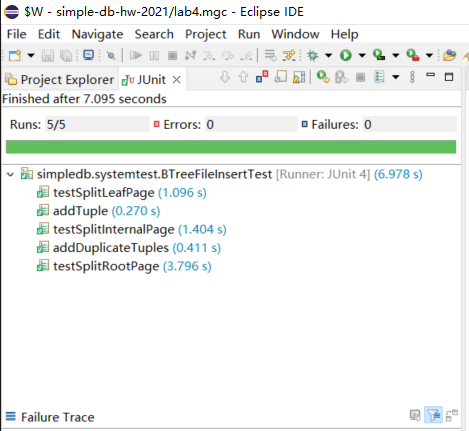
#### **问题：**

SplitInternalPage和SplitLeafPage之间最大的区别就是分裂叶子节点时，划分元素是复制上去的，而分裂内部节点时，划分元素是挤上去的（这里还要注意划分元素和导致分裂的插入元素一般都是不同的）。

#### **测例：通过本Exercise全部测例**



BTreeFileInsertTest



system test BTreeFileInsertTest

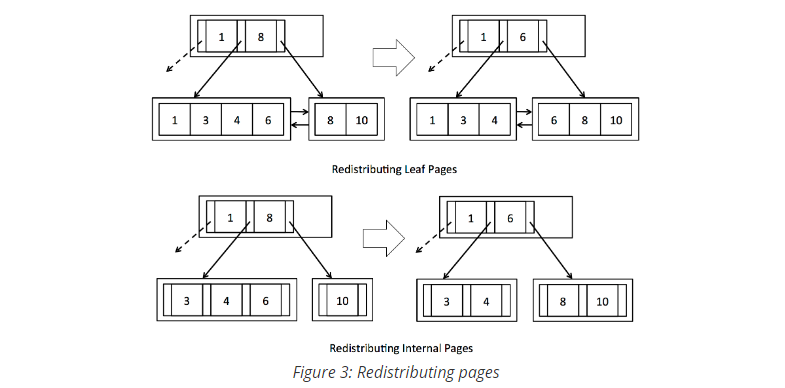
### 5.3. Exercise 3 Delete

#### **本Exercise实验内容：**

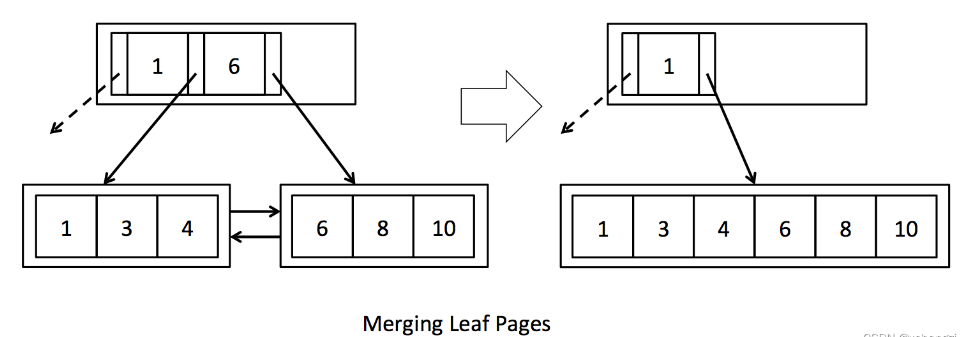
删除的话有两种情况，一种是兄弟页面比较满，自己因为删除一些tuple或者entry比较空，这时可以从兄弟页面拿一些元素过来，这样兄弟页面可以不用那么早去分裂页面，自己也可以达到元素比较多，这个对应于exercise3要做的东西；另一种情况是两个页面都是比较空的时候，这个时候需要考虑将两个页面合并成一个，以达到节省空间的目的，这对应于exercise4要做的东西。

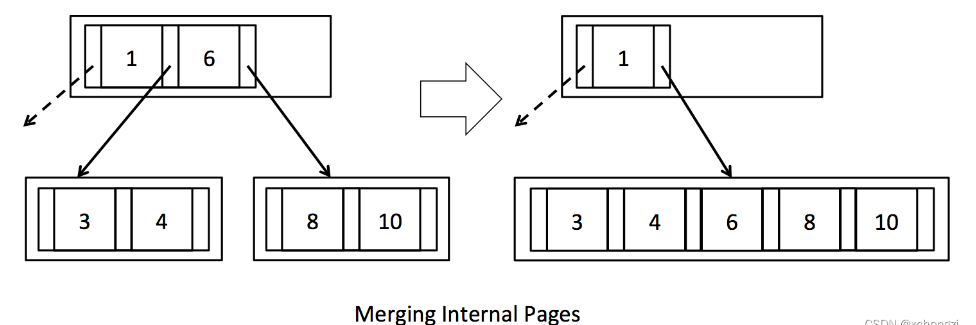
exercise3是页面元素的重新分配，internal page的entry，leaf page的tuple，两种页面的分配是有区别的，重新分配tuple的话parent是需要包含tuple的value的，而重新分配entry是不需要的，因为真正的数据是在leaf page中，这个只是作为一个索引，当然你加上也是没问题的，但是这样做是会更方便的，至少图示就是这样做的。

重新分配的话可以考虑两种策略：第一种是两个页面的元组总数加起来然后平均，第二种是一个页面留下总容量的1/2，剩下的都分配给比较空的页面，这里我采用的是前者；



调用关系





#### **实现内容：**

**BTreeFile类**

* public void stealFromLeafPage(BTreeLeafPage page, BTreeLeafPage sibling,BTreeInternalPage parent, BTreeEntry entry, boolean isRightSibling) : 实现叶节点从兄弟节点“窃取”元素的功能。主要思路如下：
  + 根据传入的参数isRightSibling确定是从左兄弟中“窃取”，还是从右兄弟中“窃取”。
  + 根据兄弟节点中tuple的数量，确定“窃取的数量”。
  + 参数entry是父节点中指向page和其兄弟节点的entry，将entry的key更改为page和其兄弟节点key的中间值。

* public void stealFromLeftInternalPage(TransactionId tid, Map<PageId, Page> dirtypages,BTreeInternalPage page, BTreeInternalPage leftSibling, BTreeInternalPage parent,BTreeEntry parentEntry) ： 实现内部节点从兄弟节点“窃取”元素的功能。主要思路如下：
  + 根据page及其左兄弟中key的数量，确定从其做兄弟中“窃取”几个key。
  + 因为内部节点与其父节点中的key值没有重复，迁移key的时候也需要将父节点中的key移动到page中。
  + 将page左兄弟节点中的key平均分配。
  + 分配之后，将page左兄弟节点中最大的key“挤到”父节点中。
  + 更新更新page与其左兄弟的父指针。

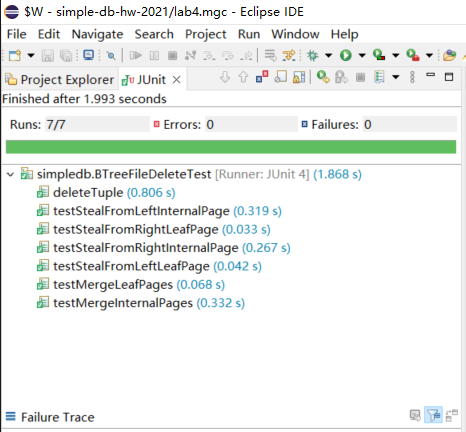
* public void stealFromRightInternalPage(TransactionId tid, Map<PageId, Page> dirtypages,BTreeInternalPage page, BTreeInternalPage leftSibling, BTreeInternalPage parent,BTreeEntry parentEntry) ： 与stealFromLeftInternalPage类似
* public void mergeLeafPages(TransactionId tid, Map<PageId, Page> dirtypages,  
   BTreeLeafPage leftPage, BTreeLeafPage rightPage, BTreeInternalPage parent, BTreeEntry parentEntry) : 合并叶子节点
  + 将rightPage中的所有tuple添加到leftPage中。
  + 判断rightPage是否有右兄弟，如果没有leftPage的右兄弟为空，如果有leftPage的右兄弟指向rightPage的右兄弟。
  + 调用setEmptyPage方法将rightPage在header标记为空。
  + 调用deleteParentEntry方法，从父级中删除左右孩子指针指向leftPage和rightPage的entry。
  + 将leftPage与parent添加到dirtypages中

* public void mergeInternalPages(TransactionId tid, Map<PageId, Page> dirtypages,BTreeInternalPage l: 合并内部节点
  + 先将父节点中的指向leftPage和rightPage的entry添加到leftPage中
  + 将rightPage中的entry添加到leftPage中
  + 更新leftPage孩子节点的指针（将原本父节点指向rightPage的孩子节点的父节点更新为leftPage）
  + 调用setEmptyPage方法将rightPage在header标记为空。
  + 调用deleteParentEntry方法，从父级中删除左右孩子指针指向leftPage和rightPage的entry。
  + 将leftPage与parent添加到dirtypages中

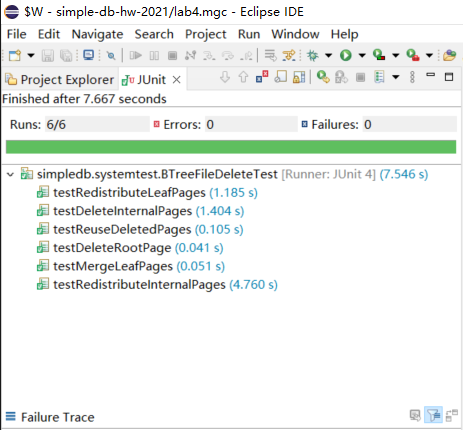
#### **问题：**

注意边界条件容易出现错误。

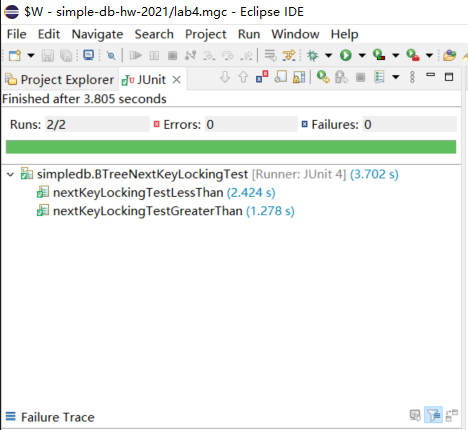
#### **测例：通过本Exercise全部测例**



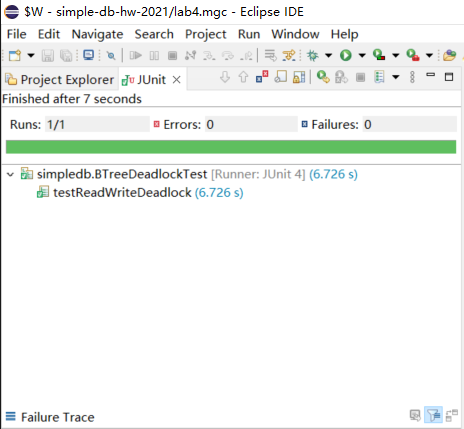
BTreeFileDeleteTest



system test BTreeFileDeleteTest



BTreeNextKeyLockingTest



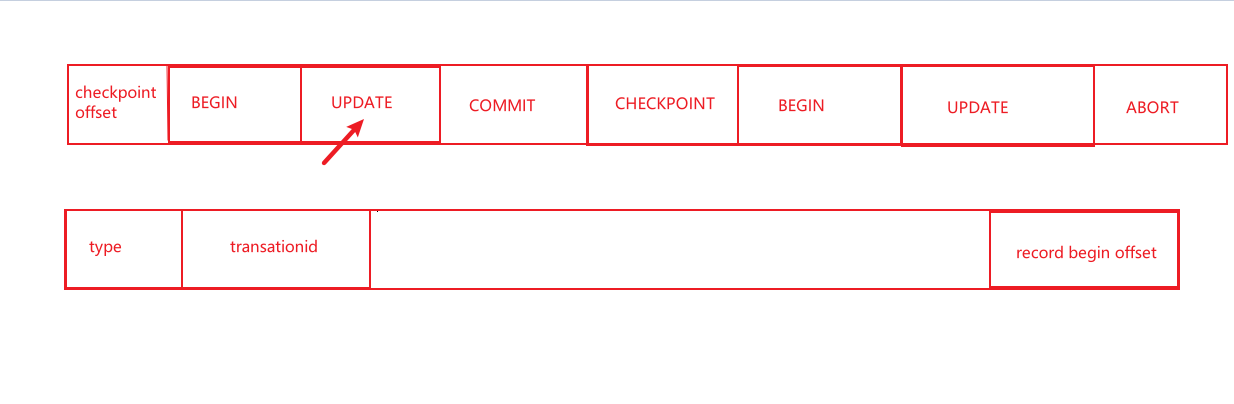
BTreeDeadlockTest

## 6. Lab 6 Recovery and Rollback

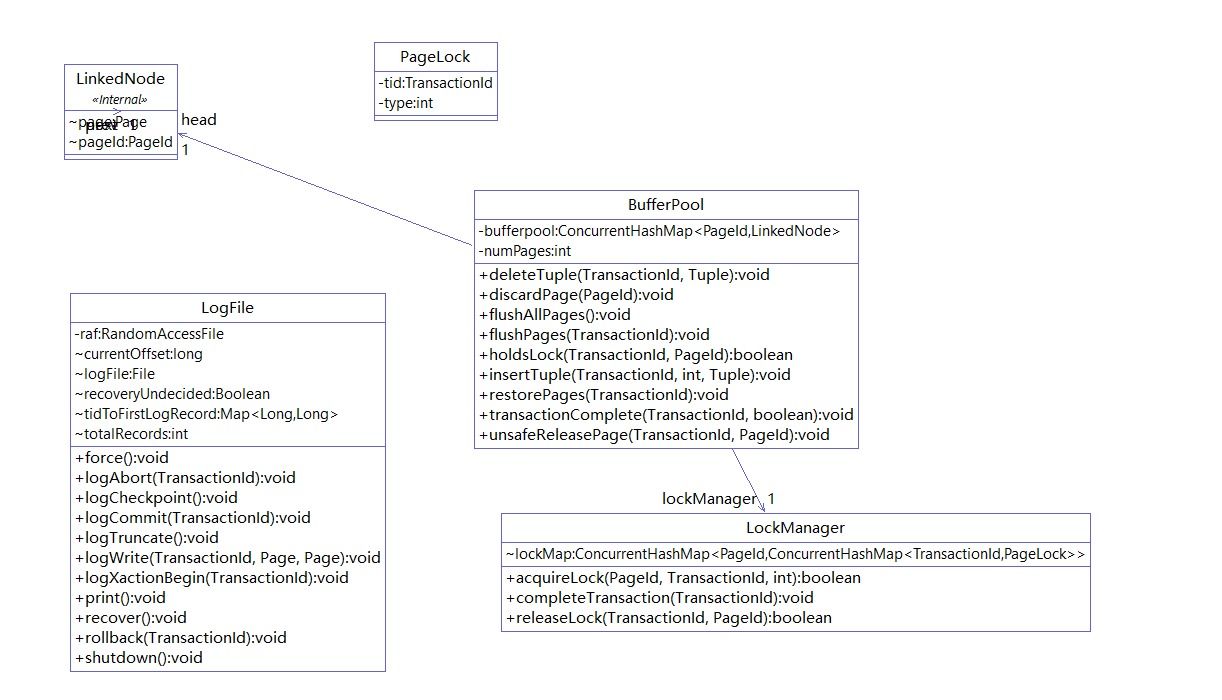
#### **实验内容：**

lab6要实现的是simpledb的日志系统，以支持回滚和崩溃恢复；

**注意点如下：**

* steal/no-force策略  
  lab6要实现的是simpledb的日志系统，以支持回滚和崩溃恢复；在lab4事务中，我们并没有考虑事务执行过程中，如果机器故障或者停电了数据丢失的问题，bufferpool采用的是no-steal/force的策略，而这个实验我们实现的是steal/no-force策略，两种策略的区别如下:
  + steal/no-steal: 是否允许一个uncommitted的事务将修改更新到磁盘，如果是steal策略，那么此时磁盘上就可能包含uncommitted的数据，因此系统需要记录undo log，以防事务abort时进行回滚（roll-back）。如果是no steal策略，就表示磁盘上不会存在uncommitted数据，因此无需回滚操作，也就无需记录undo log。
  + force/no-force:force策略表示事务在committed之后必须将所有更新立刻持久化到磁盘，这样会导致磁盘发生很多小的写操作（更可能是随机写）。no-force表示事务在committed之后可以不立即持久化到磁盘， 这样可以缓存很多的更新批量持久化到磁盘，这样可以降低磁盘操作次数（提升顺序写），但是如果committed之后发生crash，那么此时已经committed的事务数据将会丢失（因为还没有持久化到磁盘），因此系统需要记录redo log，在系统重启时候进行前滚（roll-forward）操作。
* 为了支持steal/no-force策略，即我们可以将未提交事务的数据更新到磁盘，也不必在事务提交时就一定将修改的数据刷入磁盘，我们需要用日志来记录一些修改的行为。在simpledb中，日志不区分redo log和undo log，格式较为简单，也不会记录事务执行过程中对记录的具体修改行为。
  + 对于redo log，为确保事务的持久性，redo log需要事务操作的变化，simpledb中用UPDATE格式的日志来保存数据的变化，在每次将数据页写入磁盘前需要用logWrite方法来记录变化。
  + 对于undo log，我们采用的是对heappage中保存一份旧数据。数据页一开始的旧数据是空的，那什么时候会对旧数据进行更新呢？答案是事务提交时，当事务提交时，就意味着这个修改已经是持久化到磁盘了，新的事务修改后就数据页的数据就是脏数据了，而在新事务回滚时，由于我们采用的是steal策略，脏页可能已经在页面淘汰时被写入磁盘中了，那么该如何进行恢复呢？答案是before-image，即oldData，通过上一次成功事务的数据，我们可以恢复到事务开始前的样子，这样，就可以实现了事务的回滚了。
* simpledb的日志记录一共有5种：ABORT, COMMIT， UPDATE, BEGIN, and CHECKPOINT，分别记录事务失败、事务提交、写入磁盘前的脏页、事务开始、检测点，这些格式的日志都记录在同一个日志文件中。
  + 日志通用格式：
* 
* img
  + 各种日志格式区别：
    - 对于ABORT, COMMIT, and BEGIN这三种，中间的content是空的；
    - 对于UPDATE格式的记录，有两部分组成，即before image和after image，分别记录修改前和修改后的日志；事务提交失败回滚我们会用到before image，事务提交成功但数据由于故障丢失数据我们会用到after image；
    - 对于CHECKPOINT 记录，主要记录在checkpoint点活跃的事务数，以及每个事务的的事务id和第一条日志记录的偏移量；
  + checkpoint可以说是整个日志文件的核心，在崩溃恢复时很有用；在崩溃恢复时，我们会读取到checkpoint所在的位置，在checkpoint之前的修改已经是刷入磁盘的，除非磁盘坏了否则就是永久不会丢失的；对于checkpoint之后的日志，我们只保证修改持久化到日志，但未保证将日志记录的内容持久化到磁盘，因此崩溃恢复时，我们需要从checkpoint开始往后读，然后根据日志记录进行恢复。

#### **实验类图：**



lab6类图

### 6.1. Exercise 1 Rollback

#### **本Exercise实验内容：**

rollback是undo log做的事，即提供上一个版本的快照，在回滚时将上一个版本的数据写回磁盘。

#### **实现内容：**

**LogFile类**

* public void rollback(TransactionId tid)

**基本思路：**

* 根据tidToFirstLogRecord获取该事务第一条记录的位置；
* 移动到日志开始的地方；
* 根据日志格式进行读取日志记录，读到update格式的记录时根据事务id判断是否为要修改的日志，如果是，写before image

**BufferPool类**

* private synchronized void flushPage(PageId pid) throws IOException //flush 之前需要日志
* public synchronized void flushPages(TransactionId tid) throws IOException //flush pages之前需要获取脏页

#### **问题：**

在BufferPool类中修改 flushPages方式记录脏页数据时，获取对应脏页方法使用错误。

本应该获得pid对应bufferpool中的page，使用

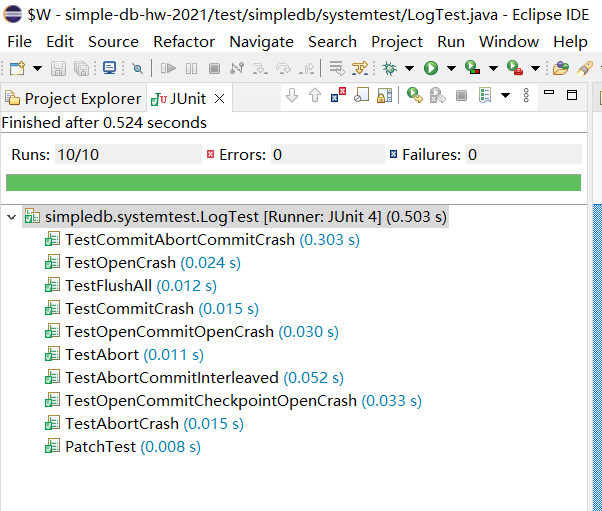
Page p = bufferpool.get(pid).page;

获取对应页。但是我之前使用了方法

DbFile dbFile = Database.getCatalog().getDatabaseFile(pid.getTableId());  
Page page = dbFile.readPage(pid);

获取对应pid在磁盘上的page，发生了错误。

#### **测例：通过本Exercise需要的测例**



system test LogTest

### 6.2. Exercise 2 Recovery

#### **本Exercise实验内容：**

崩溃恢复是redo log要做的事，在因故障数据丢失时，有部分数据是还未写入数据库的，这个时候可以利用到undo log。从日志文件中，我们可以获取到checkpoint所在位置，然后对checkpoint后面的日志记录进行读取并进行恢复数据。

#### **实现内容：**

**LogFile类**

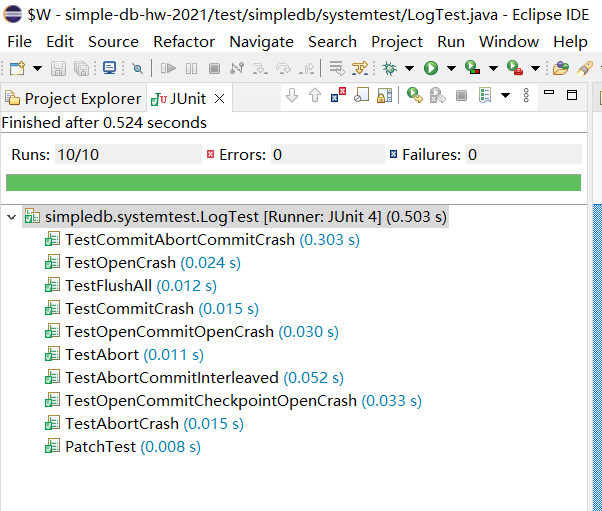
* public void recover() throws IOException

基本思路：

• 对于未提交的事务：使用before-image对其进行恢复；

• 对于已提交的事务：使用after-image对其进行恢复；

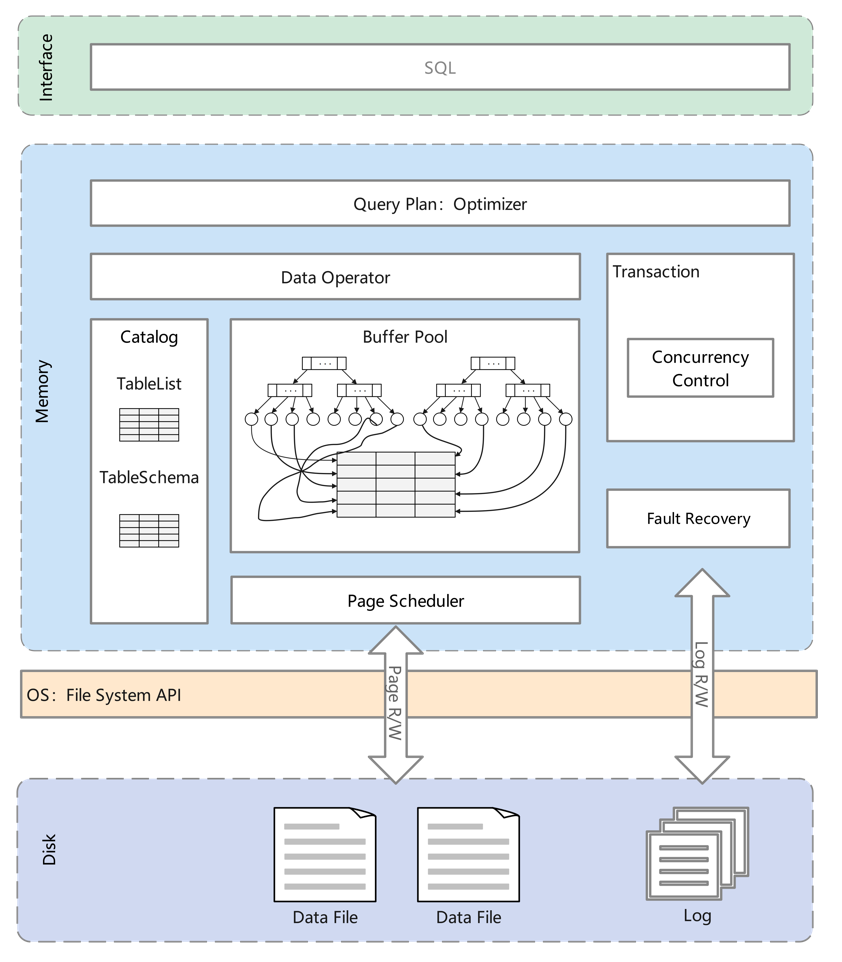
#### **测例：通过本Exercise全部测例**



system test LogTest

## 7. 总结

### 7.1. SimpleDB 总体架构



SimpleDB整体架构



SimpleDB 功能架构

SimpleDB通过循序渐进的实验构建，从最开始数据库基础架构开始，到查询，事务，锁，索引，再到恢复。最终完成了一个具备现代关系型数据库基本功能的教学数据库管理系统，具备了数据库管理系统的所有基本功能，同时在查询等方面做了尝试性的优化，可以说这是一个miniMySQL 数据库。

### 7.2. 实验收获

通次本次课程，获得收获主要为以下几个方面：

* 获得了对于现代关系型数据库的初步认识和其底层原理的基本了解，为后续在数据库方向发展打下了一定基础；
* 增强了自身的java编程的能力，同时通过学习实验中给出的框架代码，进一步学习了java开发的封装、继承等编程思想以及优秀的java代码风格；
* 通过对于本实验中较为复杂的数据库系统的设计，进一步加强了对于大型复杂系统软件设计过程的认识，增强了自身系统设计能力；
* 通过本次开发 SimpleDB 的构建过程，进一步体会了解到了复杂软件系统的逐步构建、由易入难等设计方法和贯穿开发过程的单元 测试、系统测试也加深了我们对于软件工程测试的理解。

### 7.3. 实验不足

由于时间原因，该项目中一些具有探索挑战性的项目没有尝试，只是完成了所有基本的练习。包括以下方面仍然可以继续优化：

* lab 2的join 的实现可以采用双重嵌套循化，还可以hash join merge join；
* lab 3 的优化还有更加合理的优化；
* lab 4 只是实现了page level的读写锁，还可以是tuple粒度的锁，严格2pl的锁并发度并不是很高，可以使用mvcc的方式;
* lab4 死锁判断只是用了超时的机制，可以使用死锁图检测的方式;
* lab 5 索引的并发，蟹形协议，可以参考左转螃蟹协议及其各种优化;
* lab 6 recover 采用的方式是把整个page 的before 和 after 都保存，这样会导致内存中的数据很大，相当于double capacity，所以还要看看例如innodb这种的是怎么处理的日志问题。

### 7.4. 建议

**课程优点：**

* 优秀的实验教程。本次实验课程采用的MIT6.380实验教程具有良好的测试机制、便于上手的代码框架、保姆级的教程、循序渐进的实验构建，对于我们把握数据库管理系统的整体架构，获得数据库管理系统的初步认知具有比较好的效果。
* 梯度的打分机制。本次实验课中老师设置的梯度打分即要求前四个实验必须完成，后两个实验选择完成可以使用不同层次学生的需求。既能让基础薄弱的学生了解数据库管理系统的基本原理，又能让想拔高的同学增进对DBMS的理解，取得不错的效果。

**建议：**

* 增加本实验课学时。本实验课学时相对于采用的实验教程来说偏短，需要学生在课下投入巨量时间才能完成所有实验，获得更好的分数。
* 增加阶段性检查计划。实验课过程中缺乏阶段性检查督促学生按时完成实验，需要进一步调整检查计划。
* 增加课堂讲解实验环节，使学生更快入手实验。本次课程中老师和助教发挥的作用不明显，同学们更多的是依靠自学、上网查资料等方式进一步学习。老师如果能够在每次上课前讲解一部分实验内容，可以起到很好的引导作用。

## 8. 参考资料

[1] 数据库系统概念 机械工业出版社

[2] 数据库系统概论与原理 清华大学出版社

[3] 数据库课程开源心得笔记 链接： <https://zhenghe.gitbook.io/open-courses/>