



西北工业大学
NORTHWESTERN POLYTECHNICAL UNIVERSITY

数据库系统综合实验报告

姓 名 冯雨森

学 号 2019301224

班 级 10011907

任课教师 张利军

2022 年 6 月 7 日

Contents

| | | |
|------|---|----|
| 1. | Lab1 Storage Model..... | 1 |
| 1.1. | Exercise 1 Tuple and TupleDesc | 2 |
| 1.2. | Exercise 2 Catalog | 6 |
| 1.3. | Exercise 3 BufferPool | 8 |
| 1.4. | Exercise 4 HeapPage..... | 9 |
| 1.5. | Exercise 5 HeapFile | 12 |
| 1.6. | Exercise 6 SeqScan | 13 |
| 2. | Lab2 Operators..... | 16 |
| 2.1. | Exercise 1 Filter and Join..... | 17 |
| 2.2. | Exercise 2 Aggregates | 21 |
| 2.3. | Exercise 3 Heap File Mutability | 24 |
| 2.4. | Exercise 4 Insertion and Deletion | 26 |
| 2.5. | Exercise 5 Page Eviction..... | 27 |
| 3. | Lab3 Optimizer | 29 |
| 3.1. | Exercise 1 and 2 Filter Selectivity | 31 |
| 3.2. | Exercise 3 Join Cardinality | 34 |
| 3.3. | Exercise 4 Join Ordering..... | 35 |
| 4. | Lab4 Transaction..... | 39 |
| 4.1. | Exercise 1 Granting Locks | 40 |
| 4.2. | Exercise 2 Lock Lifetime | 43 |
| 4.3. | Exercise 3 Implementing NO STEAL | 44 |
| 4.4. | Exercise 4 Transactions..... | 45 |
| 4.5. | Exercise 5 Deadlocks and Aborts..... | 46 |
| 5. | B+-Tree Index | 48 |
| 5.1. | Exercise 1 Search | 50 |
| 5.2. | Exercise 2 Insert..... | 53 |
| 5.3. | Exercise 3 Delete | 56 |
| 5.4. | Exercise 4 Merging pages | 57 |

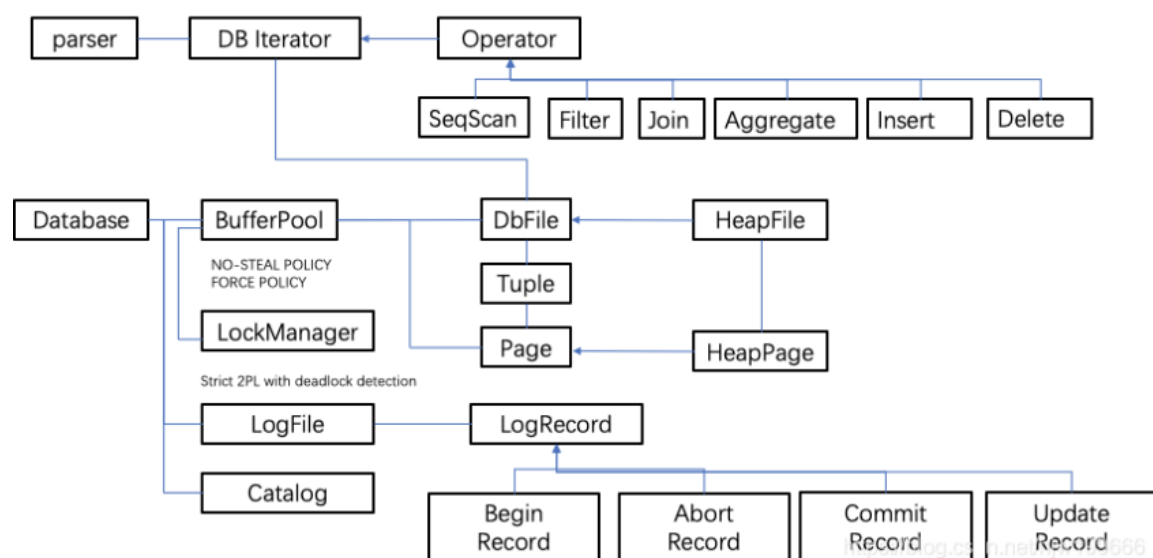
| | | |
|------|-----------------------------------|----|
| 6. | Lab 6 Recovery and Rollback | 60 |
| 6.1. | Exercise 1 Rollback | 60 |
| 6.2. | Exercise 2 Recovery..... | 61 |
| 7. | 总结..... | 62 |
| 7.1. | SimpleDB 总体架构..... | 62 |
| 7.2. | 实验收获..... | 63 |
| 7.3. | 实验不足..... | 63 |
| 7.4. | 建议..... | 64 |
| 8. | 参考资料..... | 64 |

1. Lab1 Storage Model

SimpleDB 由以下部分组成：

- 代表字段、Tuple 和 Tuple 模式的类。
- 对 Tuple 应用谓词和条件的类。
- 一个或多个访问方法（例如，堆文件），将关系存储在磁盘上，并提供一种方法来遍历这些关系的 Tuple。
- 一个操作者类的集合（例如，选择、连接、插入、删除等），用于处理 Tuple。
- 一个缓冲池，用于缓存内存中的活动 Tuple 和页面，并处理并发控制和事务；以及。
- 一个 catalog，用于存储关于可用表和它们的模式的信息。

SimpleDB Architecture



Lab1 主要实现整个数据库基本的存储逻辑结构，也即一个粗略轮廓，具体包括 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)。

src/java/simplydb/storage/Tuple.java

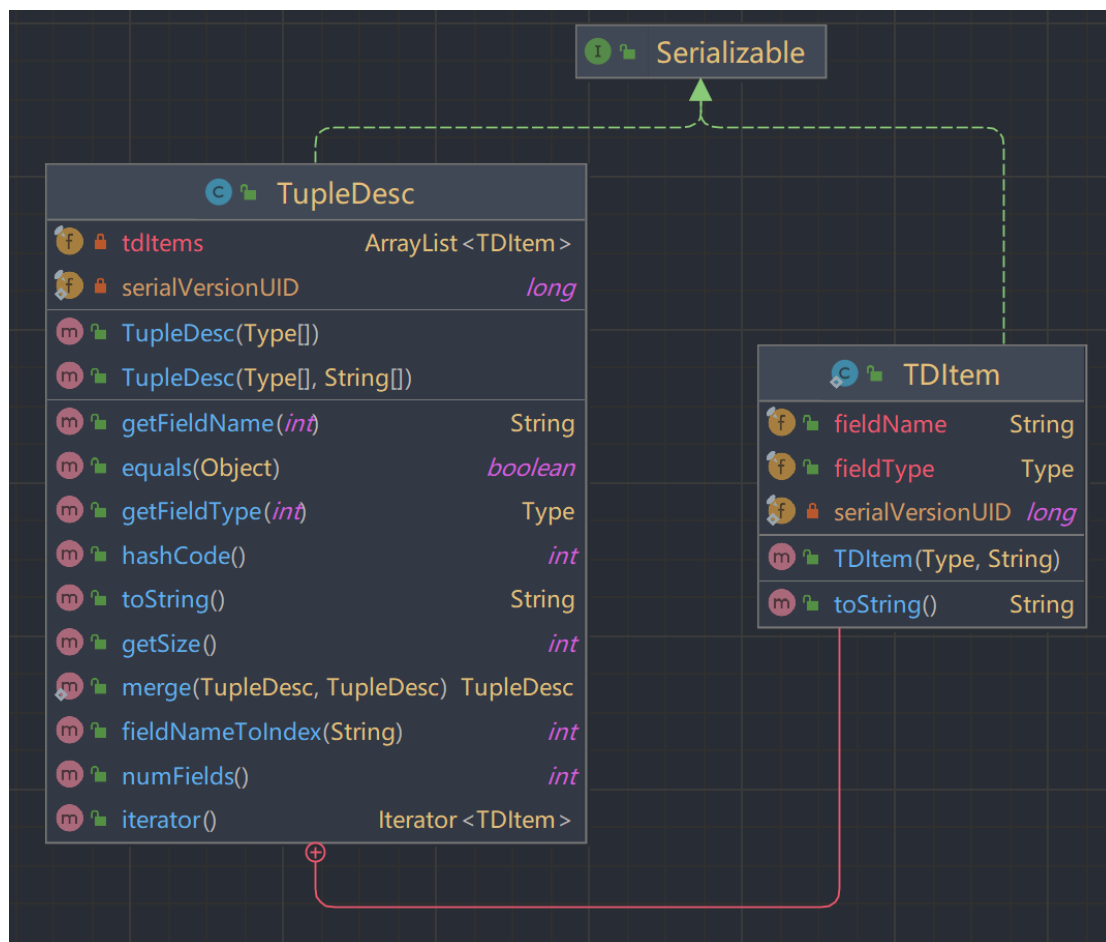
(2) 一个 Tuple 对象表示一行数据，一个 TupleDesc 对象表示数据的表头。

TupleDesc 表示表头，其中有一个 TDIItem 队列，而 TDIItem 表示一个属性，其中有 fieldType 和 fieldName 两个字段。例如学生的姓名属性，其中 fieldName 就是姓名，因为姓名是字符串而 fieldType 就是字符串。fieldType 是一个 Type 对象，点进去后发现分为 INT_TYPE 和 STRING_TYPE 类型。

Field 是一个接口，实现了 StringField 和 IntField。

(3)

TupleDesc 类



参数：

```
private final <ArrayList>TDIItem[] tdlItems;
```

```
Type fieldType
```

```
String fieldName
```

TupleDesc 中提供了一个 TDIItem 辅助类，类中有 Type fieldType、String

fieldName 两个参数。例如 id 为一个 TDIItem, 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 TDIItem。我创建了一个 ArrayList, 便于直接返回 iterator。

方法:

public TupleDesc(Type[] typeAr, String[] fieldAr): 初始化方法, 将 typeAr 和 FieldAr 数组中的对应元素通过一遍循环加入 tdItems 中即可。

public TupleDesc(Type[] typeAr): 初始化构造方法, 将 field 设为 “anonymous”, 同上操作即可。

public Iterator<TDItem> iterator (): 返回所有属性的迭代器, 直接返回 tdItems.iterator()即可。

public int numFields(): 返回 TupleDesc 中属性的数量, 即 tdItems 的大小, 直接返回 tdItems.size()。

public String getFieldName(int i): 返回第 i 个属性的属性名, tdItems 第 i 个元素的 fieldName。

public Type getFieldType(int i): 返回第 i 个属性的属性类型, tdItems 第 i 个元素的 fieldType。

public int fieldNameToIndex(String name): 根据属性名返回它在 tdItem 中的序号,即在 tdItems 中查找 fieldName 为 name 的属性元素, 返回序号。

public int getSize(): 返回元组所占的字节大小, 计算 tdItems 中每个元素 fieldType 的大小之和。

public static TupleDesc merge(TupleDesc td1, TupleDesc td2): 合并两个 tdItem, 构造两个 ArrayList: types 和 name, 分别放置两个 td 中的类型和名字, 在调用 TupleDesc 的构造方法 1。

public boolean equals(Object o): equals 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。可以使用 IDEA 生成 equals 模板, 或采用学习到的 equals 方法, 先判断是否是相同的类, 然后比较两个 TupleDesc 每个属性的类型。

public String toString(): toString 方法返回 TupleDesc 的所有属性名:
“id,name,age”。

Tuple 类

参数:

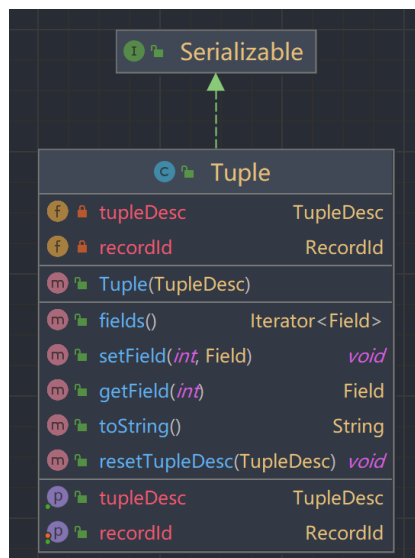
private TupleDesc tupleDesc: 元组对应的属性

private RecordId recordId: 元组的 id

private final Field[] fields: 用于存储 Tuple 中的所有字段

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

IntField 和 StringField 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



方法:

public Tuple(TupleDesc td): 构造函数, 直接赋值即可, 构造对应大小的 fields。

public TupleDesc getTupleDesc(): 获得 Tuple 对应的 TupleDesc, 直接返回 tupleDesc。

public void setRecordId(RecordId rid): 设置元组 id, 修改 recordId 为 rid。

public RecordId getRecordId(): 获得元组 id, 返回 recordId。

public void setField(int i, Field f): 为字段赋值, 直接 fields.add(i, f)

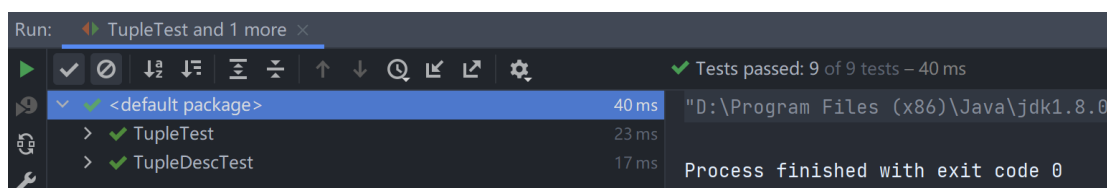
public Field getField(int i): 获得指定字段, 返回 fields.get(i)

`public String toString():` `toString()`方法返回所有字段，将 `fields` 中每个字段通过 `StringBuffer` 转换为字符串返回。

`public Iterator<Field> fields():` 返回字段的迭代器，直接返回 `fields.iterator()`。

`public void resetTupleDesc(TupleDesc td):` 重置 `TupleDesc`，将 `tupleDesc` 设置为 `td`。

(4) 测试截图



1.2. Exercise 2 Catalog

(1) 实现下面的方法:

`src/java/simpliedb/common/Catalog.java`

在这一点上，你的代码应该通过 `CatalogTest` 的单元测试。

(2) `catalog` 类描述的是数据库实例。包含了数据库现有的表信息以及表的 `schema` 信息。现在需要实现添加新表的功能，以及从特定的表中提取信息。提取信息时通过表对应的 `TupleDesc` 对象决定操作的字段类型和数量。

在整个 `SimpleDb` 中, `Catalog` 是全局唯一的，可以通过方法 `Database.getCatalog()` 获得，`global buffer pool` 可以通过方法 `Database.getBufferPool()` 获得。

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。

(3) `Catalog` 类

参数:

`private final ConcurrentHashMap<Integer,Table> catalog:` 表 `id` 与表的映射，使用 `ConcurrentHashMap` 线程安全，考虑多线程。

`private final ConcurrentHashMap<String,Integer> hashMap:` 表名字与表 `id` 的映射，

方便后续通过表名获取表 id

Catalog 提供的辅助类 Table，包含参数 tableName、primartKey（表中的主键）、DbFile dbFile（用于存储表的内容），DbFile 中提供了 getId()方法，可以获取此 Dbfile 对应表的 tableid。这个 id 并不是顺序生成，后续 exercise 中通过 file.getAbsolutePath().hashCode()生成的

方法：

public Catalog(): 构造方法，上述两个参数构造新的 HashmA

ublic void addTable(DbFile file, String name, String pkeyField): 向 CataLog 中添加表，在两个 HashMap 同时添加 Table。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，在 StringTables 获得

public TupleDesc getTupleDesc(int tableid): 通过表 id 获得表的 TupleDesc，在 tables 获得表，然后得到表的 TupleDesc

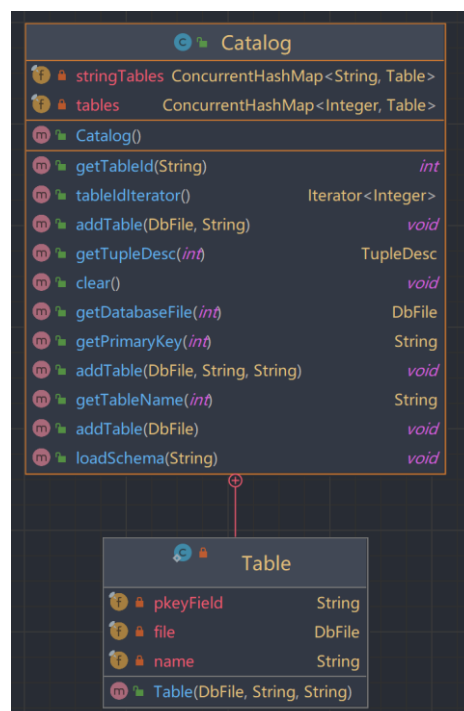
public DbFile getDatabaseFile(int tableid): 通过表 id 获得表的内容 DbFile，同上

public String getPrimaryKey(int tableid): 通过表 id 获得表的主键，同上

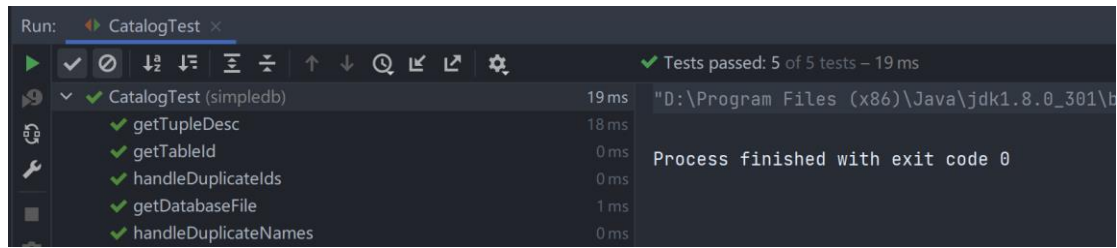
public Iterator<Integer> tableIdIterator(): 返回 tableId 的迭代器，返回 tables.keySet().iterator()

public String getTableName(int id): 通过表 id 获得表名，同上

public void clear(): 清空 CataLog，清空 tables。



(4) 测试截图:



1.3. Exercise 3 BufferPool

(1) 实现文件中的 `getPage()` 方法:

`src/java/simplydb/storage/BufferPool.java`

我们没有为 `BufferPool` 提供单元测试。你实现的功能将在下面 `HeapFile` 的实现中得到测试。你应该使用 `DbFile.readPage` 方法来访问 `DbFile` 的页面。

(2) 用 `ConcurrentHashMap` (`HashMap` 存在并发问题, 不过本地测试没有体现) 来存 `page`, 其中 `key` 和 `value` 分别是 `PageId` 和 `Page`。

`getPage()` 首先根据 `pid` 判断是否已经缓存, 如果没有缓存再调用

`Database.getCatalog()` 方法去 `Catalog` 中加载。加载后依旧缓存一下。

`DbFile` 接口实现了和磁盘读写之间的读写 `page`, 插入删除 `tuple` 等功能。

一个 `DbFile` 中存了一个 `table`, 数据库中的 `Table` 和 `DbFile` 是对应。

`HeapFile` 是 `DbFile` 的一种实现。除此之外还有 `B-trees` 实现, 此处仅需由 `heap file` 提供即可。

一个 `HeapFile` 中存了一个 `Page` 集合, `HeapPage` 是 `Page` 的一种实现, 其中每一个 `page` 都存有固定数量(`BufferPool.DEFAULT_PAGE_SIZE`)的 `tuple`。

`HeapFile` 中的每一页(`page`)都有一组槽(`slot`), 每一个 `slot` 中“嵌入”一个 `tuple`。

此外, 每一个 `page` 除了 `slot` 之外还有 `head` 部分, `head` 部分用来存 `page` 中的 `slot` 是否被使用。

数据库中的 `block` 对应此处的 `HeapFile`, 而在内存中数据以 `Page` 为单位。

(3)

`BufferPool` 类

参数:

```
private static final int DEFAULT_PAGE_SIZE = 4096
```

private static int pageSize = DEFAULT_PAGE_SIZE: 默认的 page 大小

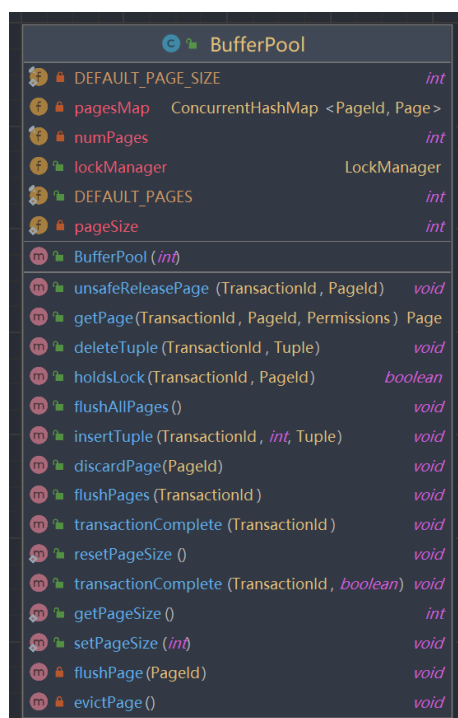
private final int numPages: bufferPool 能够读取的 page 数量

private ConcurrentHashMap<PageId, Page> pages: pageId 到 page 的映射

方法:

public BufferPool(int numPages): 构造函数

public Page getPage(TransactionId tid, PageId pid, Permissions perm): 从缓冲池中获取 pid 对应的 page, 如果缓冲池中沒有就去硬盘中搜索并保存到缓冲池中, 如果缓冲池满了则抛出异常



(4) 暂时没有单元测试

1.4. Exercise 4 HeapPage

(1) Implement the skeleton methods in:

src/java/simplydb/storage/HeapPageId.java

src/java/simplydb/storage/RecordId.java

src/java/simplydb/storage/HeapPage.java

尽管你不会在实验 1 中直接使用它们, 但我们要求你在 HeapPage 中实现 getNumEmptySlots()和 isSlotUsed()。这些都需要在页头中推送一些位。你可能

会发现查看 HeapPage 或 src/simplydb/HeapFileEncoder.java 中提供的其他方法对理解页面的布局有帮助。你还需要在页面中的 tuple 上实现一个迭代器，这可能涉及一个辅助类或数据结构。在这一点上，你的代码应该通过 HeapPageIdTest、RecordIdTest 和 HeapPageReadTest 的单元测试。

(2) 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）。

(3)

HeapPageId 类

参数：

private final int tableId: page 所在表的 id

private final int pageNumber: page 的序号

heapPageId 由 page 所在的表 id 和 page 的序号组成

方法：

public HeapPageId(int tableId, int pgNo): 构造函数，参数赋值即可

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

参数：

private PageId pageId: 元组 id 所在页的 pageId

private int tupleNumber: 元组的序号

元组 Id 由 pageId 和元组序号构成

方法：

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

参数:

private final HeapPageId pid: pageId

private final TupleDesc td: page 对应的属性行

private final byte[] header: slot 的 bitmap, 用于判断 slot 是否被占用

private final Tuple[] tuples: page 中的元组

private final int numSlots: page 中 slot 的数量

方法:

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: tuples per page = floor((page size * 8) / (tuple size * 8 + 1))。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:

headerBytes = ceiling(tupsPerPage/8)。

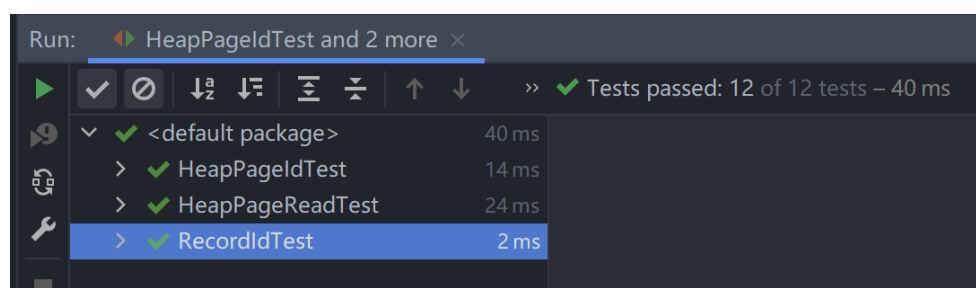
public HeapPageId getId(): 返回 pageId

public boolean isSlotUsed(int i): 判断第 i 个 slot 是否为空, header 储存方式为 byte 数组, 每个 byte 包含 8 个 bit, bitmap 中, low bits 代表了先填入的 slots 状态。因此, 第一个 headerByte 的最小 bit 代表了第一个 slot 是否使用, 第二小的 bit 代表了第二个 slot 是否使用。

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)

(6) 测试截图



1.5. Exercise 5 HeapFile

(1) Implement the skeleton methods in:

`src/java/simpliedb/storage/HeapFile.java`

要从磁盘上读取一个页面，你首先需要计算出文件中的正确偏移量。提示：你需要对文件进行随机访问，以便在任意的偏移量上读写页面。当从磁盘上读取一个页面时，你不应该调用 `BufferPool` 方法。你还需要实现 `HeapFile.iterator()` 方法，它应该遍历 `HeapFile` 中每个页面的 `tuple`。迭代器必须使用 `BufferPool.getPage()` 方法来访问 `HeapFile` 中的页面。这个方法将页面加载到缓冲池中，最终将被用于（在后面的实验中）实现基于锁的并发控制和恢复。不要在 `open()` 调用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的错误。

在这一点上，你的代码应该通过 `HeapFileReadTest` 的单元测试。

(2) 从磁盘上读取一个页面。大致思路如下：首先确定偏移值（页数乘单页大小），然后初始化一张空 `page`，最后根据偏移值将空 `page` 填满。最后跑通 `HeapFileReadTest` 这个类。`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。

(3)

`HeapFile` 类

参数：

`private final File file`: 表中的内容

`private final TupleDesc tupleDesc`: 表的属性行

方法：

`public HeapFile(File f, TupleDesc td)`: 构造函数

`public File getFile()`: 返回表的内容

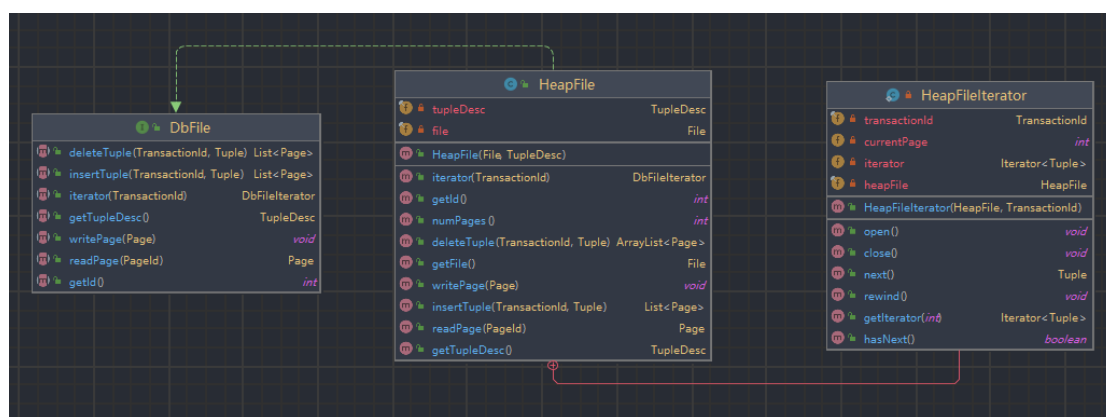
`public int getId()`: 返回标识此表的唯一 `id`，`file.getAbsolutePath().hashCode()`，应该确保每个表都有一个唯一的 `id`，对于特定的 `HeapFile` 文件返回相同的值。文档建议使用 `heapfile` 文件的绝对文件名进行 `hash`

`public TupleDesc getTupleDesc():` 返回表的属性行

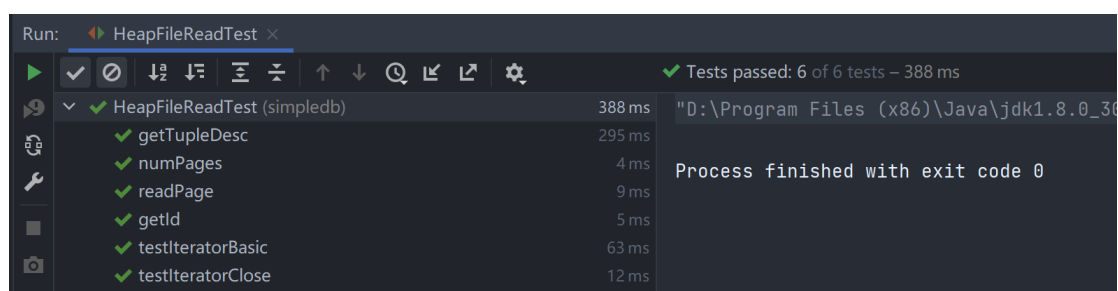
`public int numPages():` 返回表中的 page 数目

`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。`HeapFile` 的迭代挺难，`HeapFileIterator` 相当于是对整个表中所有的元组进行了迭代操作，需要复写接口的 `next`, `hasNext`, `open` 三个方法，基本思路是对 `File` 中的每个 `Page` 进行元组迭代，需要自己维护 `currentPage` 和游标。



(4) 测试截图



1.6. Exercise 6 SeqScan

(1) Implement the skeleton methods in:

`src/java/simplifiedb/execution/SeqScan.java`

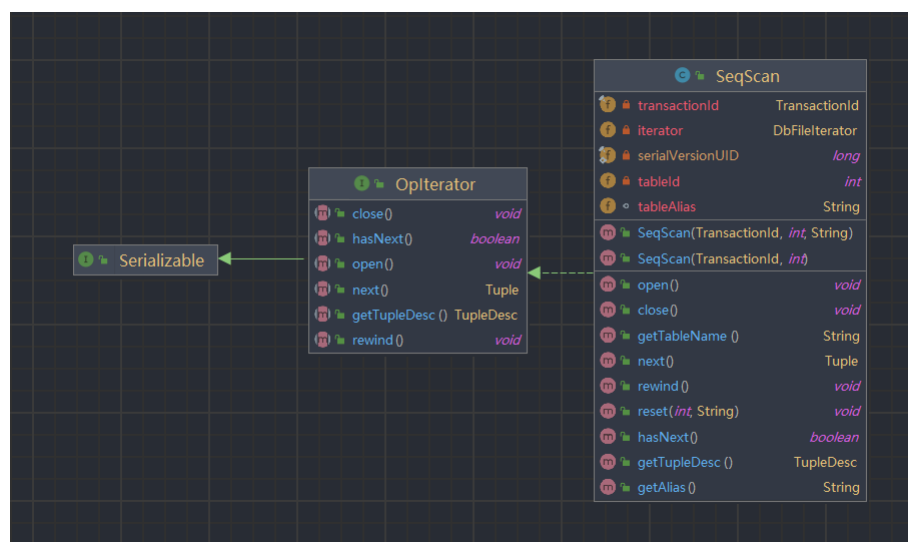
这个操作者从构造函数中的 `tableid` 指定的表中的页面中顺序扫描所有 tuple。

这个操作符应该通过 `DbFile.iterator()` 方法访问 tuple 。

在这一点上，你应该能够完成 `ScanTest` 系统测试。干得好！

(2) 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。

(3) `SeqScan` 类



参数：

`private final TransactionId tid`: 事务 id

`private int tableId`: 欲扫描的表的 `tableId`

`private String tableAlias`: 表的别名，当返回 `TupleDesc` 时，要在 `TupleDesc.fileName` 前面加上 `tableAlias`，如果 `tableAlias==null` 或 `fileName==null`，则将 `TupleDesc.fileName` 设置为 `null.fileName`、`tableAlias.null`、`null.null`

`private DbFileIterator iterator`: 用于遍历表中的所有 tuple

方法：

`public SeqScan(TransactionId tid, int tableid, String tableAlias)`: 构造函数

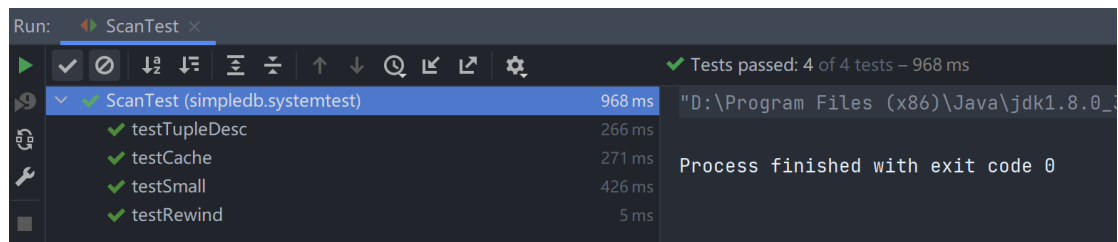
`public String getAlias()`: 返回表的别名

`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 `fileName` 前添加表的别名

还有遍历表需要用到的 `open()`、`hasNext()`、`next()`、`close()`、`rewind()`。

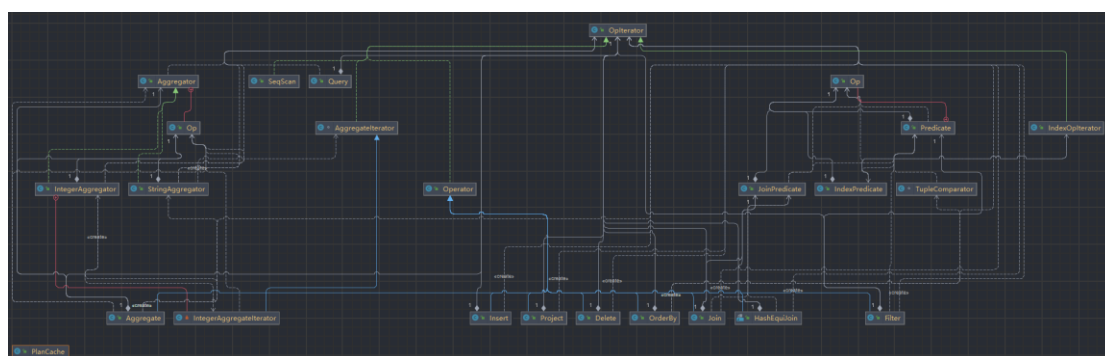
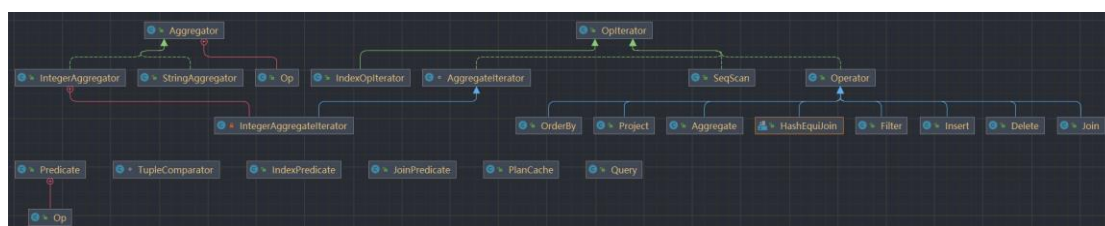
(4) 测试截图



2. Lab2 Operators

这个 lab 大致要实现。

- 实现 Filter 和 Join ， 已经提供了 Project 和 OrderBy 的实现。
- 实现 StringAggregator 和 IntegerAggregator 。编写计算一个特定字段在输入 tuple 序列的多个组中的聚合。其中 IntegerAggregator 使用整数除法来计算平均数，因为 SimpleDB 只支持整数。StringAggregator 只需要支持 COUNT 聚合，因为其他操作对字符串没有意义。
- 实现 Aggregate 操作符。和其他运算符一样，聚合运算符实现了 OpIterator 接口，这样它们就可以放在 SimpleDB 查询计划中。注意 Aggregate 运算符的输出是每次调用 next() 时整个组的聚合值，并且聚合构造器需要聚合和分组字段。
- 实现 insert 和 delete 操作符。像所有的操作符一样，Insert 和 Delete 实现了 OpIterator，接受一个要插入或删除的 tuple 流，并输出一个带有整数字段的单一 tuple ，表示插入或删除的 tuple 数量。这些操作者将需要调用 BufferPool 中的适当方法，这些方法实际上是修改磁盘上的页面。检查插入和删除 tuple 的测试是否正常工作。
- 实现 BufferPool 中的页面置换策略(LRU)。



2.1. Exercise 1 Filter and Join

(1) 实现

execution/Predicate.java

execution/JoinPredicate.java,

execution/Filter.java

execution/Join.java

并通过 PredicateTest、JoinPredicateTest、FilterTest 和 JoinTest 中的单元测试。此外，还需通过系统测试 FilterTest 和 JoinTest。

(2) Filter, Join, Project 和 OrderBy 都是数据库中常见的算子(operator)。

已经提供了 Project 和 OrderBy 的实现，实现 Filter 和 Join。

Project 是投影的意思。使用 SELECT * 表示查询表的所有列，使用 SELECT 列 1, 列 2, 列 3 则可以仅返回指定列，这种操作称为投影。研究一下 Project 类就会发现其实就选择指定列输出。

OrderBy 表示按照指定字段排序输出结果集。

接下来实现 Filter 直译是过滤的意思，将符合条件的留下，所以需要有一个判断语句。而判断也就是 Predicate 直译是谓词，第一次看到这个意思我是迷茫的，其实本质上就是一个真或假的表达式。准确的定义解释是：A predicate is an expression that evaluates to True or False。

Predicate.java 比较表内的字段和提供的数据，三个参数分别是待比较的字段序号、比较符和待比较的数。其中 filter() 方法输入一个 Tuple，然后比较 Tuple 的 Field 是否符合预期。

JoinPredicate.java 和 Predicate.java 类似，只是实现两个 Tuple 的比较。

Filter.java 在构造函数中实例化 Predicate 和 OpIterator。其中 fetchNext() 方法逐个读取 OpIterator 中的 Tuple，然后让他们与 Predicate 中 Field 进行比较，如果为真则返回该 Tuple。

Join.java 就是对 JoinPredicate.java 的使用，通过构造函数实例化 JoinPredicate 和两个 OpIterator。实现一系列 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 join 的字段进行 join。

fetchNext 中由两个 while 循环进行遍历，直到最外层迭代器遍历完成，每次遍历 child1 取出一个 Tuple，与 child2 中的所有 Tuple 做 filter 比较，直到有符合要求的，创建新的 TupleDesc，并且将 child1 和 child2 的字段（field），加入 newTuple 中，然后返回 newTuple，同时将 child2 重置到最开始。

（3）Predicate 类

参数：

Field operand: 指定的比较字段，Field.compare()中的参数

int field: tuple 中与指定字段对应的字段的序号

Op op: 执行的比较逻辑，Field.compare()中的参数

方法：

public Predicate(int field, Op op, Field operand): 初始化方法

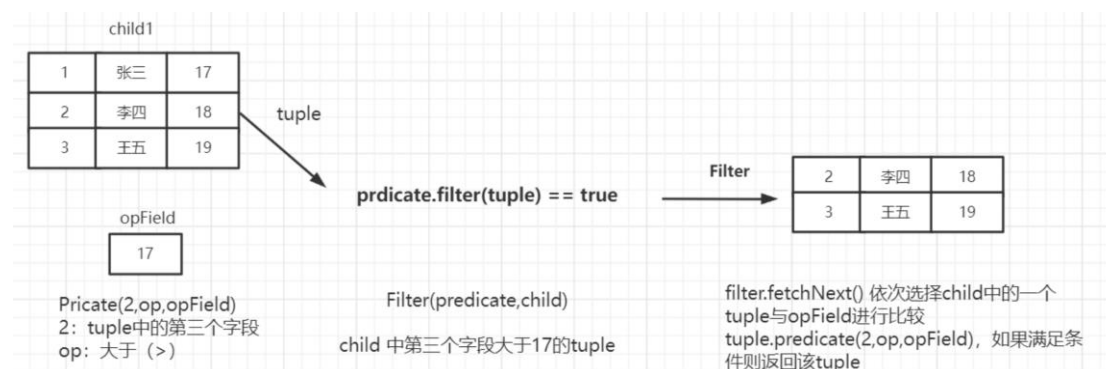
public int getField(): 返回 field

public Op getOp(): 返回 op

public Field getOperand(): 返回 operand

public boolean filter(Tuple t): 对元组 t 进行比较

t.getField(field).compare(op,operand)将 filed 与 opField 的进行比较



Filter 类

参数：

Predicate predicate;: 对 Predicate 封装，通过 predicate 实现对每一个 tuple 的过滤操作

OpIterator child; : 待过滤的 tuples 的迭代器

方法：

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，对 child 进行循环，如果 predicate.filter(tuple)为真，则返回 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

参数:

int field1; tuple1 中进行比较的字段的序号

int field2; tuple2 中进行比较的字段的序号

Predicate.Op op; 比较逻辑

方法:

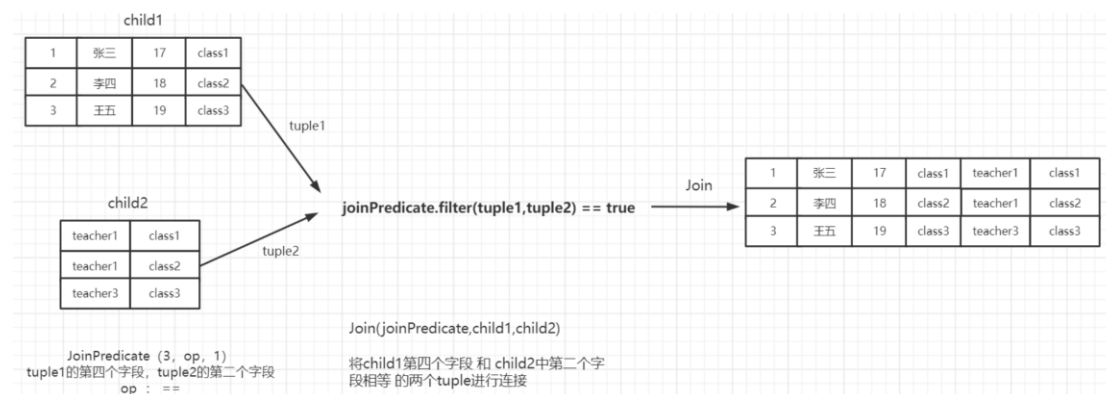
public JoinPredicate(int field1, Predicate.Op op, int field2): 初始化方法

public int getField1(): 返回 field1

public int getField2(): 返回 field2

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，即 t1.getField(this.field1).compare(this.op, t2.getField(this.field2))



Join 类

参数:

JoinPredicate predicate;

OpIterator child1;: 用于连接的 left tuples

OpIterator child2;: 用于连接的 right tuples

Tuple left;: fetchNext () 方法每次获得一个连接后的结果, 用嵌套循环每次选择 child 中的一个 left 依次与 child2 中符合条件的 right 进行连接, 将 child2 中的所有 tuples 比较完之后, left = child1.next(), child2.rewind()

方法:

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

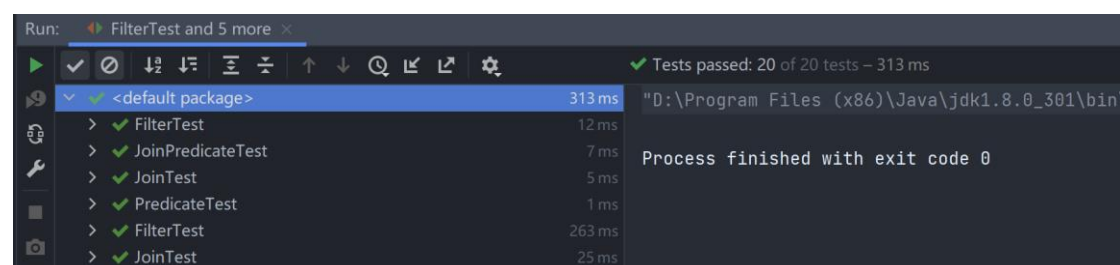
public void setChildren(OpIterator[] children)

public void open()

public void close()

public void rewind()

(4) 测试截图



2.2. Exercise 2 Aggregates

(1) 实现下面几个方法并通过 `IntegerAggregatorTest`、`StringAggregatorTest` 和 `AggregateTest` 单元测试。此外还需要通过 `AggregateTest` 的系统测试。

`src/java/simplydb/execution/IntegerAggregator.java`

`src/java/simplydb/execution/StringAggregator.java`

`src/java/simplydb/execution/Aggregate.java`

(2) 只需要实现单个字段的聚合 (aggregation) 和单个字段的分组 (group by) 即可。聚合其实就对一组数据进行操作 (加减乘除, 最值等)。具体可参考: [SQL GROUP BY 语句](https://www.runoob.com/sql/sql-groupby.html)。

`IntegerAggregator(0, Type.INT_TYPE, 1, Aggregator.Op.SUM)` 是生成一个整数聚合的对象。

其中 0 表示分组(Group By)字段位置, 也就是根据第零列来聚合。可以为 `NO_GROUPING`, 表示不进行聚合。

`Type.INT_TYPE` 表示这一列的数据类型, 目前只有整数和字符串。1 表示待聚合的字段, `Aggregator.Op.SUM` 表示执行加法操作。

需要看懂 `IntegerAggregatorTest` 测试类。其中 `scan1` 是一张基础表, `sum/min/max/avg` 是四张经过聚合操作后的表, 用于验证 `scan1` 经过聚合后的结果是否符合预期。

`mergeTupleIntoGroup()` 根据 `gbField` 字段先判断是否需要进行 group by。如果需要, 那么根据 `gbField` 从 `tup` 中提取待聚合的字段, 再判断是否是初次填入, 然后根据对应 `Op` 执行对应逻辑。如果不需要 group by 直接累加即可, 不需要映射,

`StringAggregator` 和 `IntegerAggregator` 逻辑类似, 并且仅支持 `COUNT`。

`Aggregate` 是将前两个整合一下。

(3) `IntegerAggregator` 类

对整数类型的字段进行分组聚合操作

参数:

`int gbfield;` 分组字段的序号

`Type gbfieldType;` 分组字段的类型

int aggField;: 聚合字段的序号

private Op aop; 聚合的操作符

方法:

public IntegerAggregator(int gbfield, Type gbfieldtype, int afield, Op what):初始化方法

public void mergeTupleIntoGroup(Tuple tup): 聚合操作的执行过程是:

- 1.根据构造器给定的 aggregateField 获取在 tup 中的聚合字段及其值;
- 2.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;
- 3.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。

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

StringAggregator 类

对 String 类型的字段实现分组聚合操作, 和实现 IntegerAggregator 类似, 只需要实现 count。

Aggregator 类

对 IntegerAggregator、StringAggregator 进行封装

参数:

OpIterator child;: 需要聚合的 tuples

int aggfield;: 待聚合字段的序号

int gbfield;: 分组字段的序号

Aggregator.Op aop;: 运算符

Aggregator aggregator;: 进行聚合操作的类

OpIterator aggIterator;: 聚合结果的迭代器

TupleDesc aggTupleDesc;: 聚合结果的属性行

方法:

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

Aggregator.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 Aggregator, 然后通过 Aggregator 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

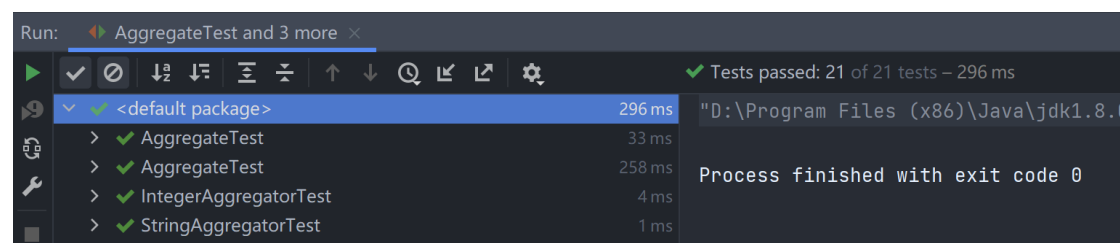
public void rewind()

public TupleDesc getTupleDesc(): 返回聚合后的结果集的 TupleDesc

public OpIterator[] getChildren()

public void setChildren(OpIterator[] children)

(4) 测试截图



2.3. Exercise 3 Heap File Mutability

(1) 增加 tuple 或删除 tuple

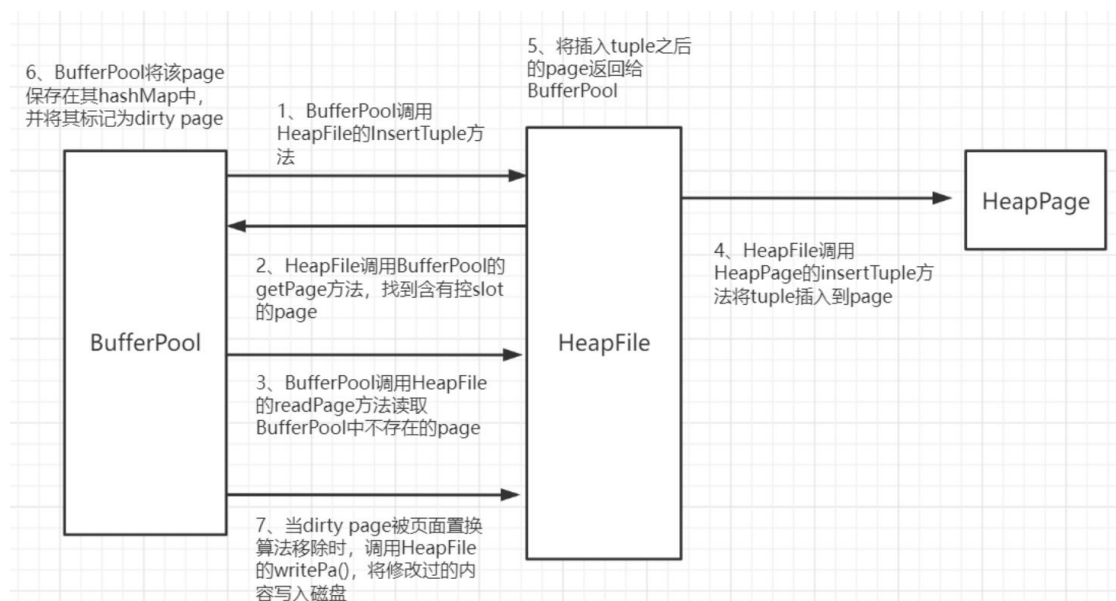
编写 HeapPage.java 并通过 HeapPageWriteTest 。

编写 HeapFile.java 并通过 HeapFileWriteTest

编写 BufferPool.java 中的 insertTuple() 和 deleteTuple() 并通过

BufferPoolWriteTest

(2)



首先根据要删除 tuple 的 RecordId 判断是否被使用，如果已经被使用就比较当前的 tuple 和待删除的 tuple 对象，一致就删除并标记。如果没有被使用那么 tuple slot 就是空。

markDirty() 用一个队列来记录脏页的 tid，如果是脏页就加入队列中，如果不是就从队列中删除。

isDirty() 返回队列中最后一个脏页，如果没有脏页就返回 null。

insertTuple() 首先判断当前页面 tid 和待插入 tuple 的 TupleDesc 是否匹配。然后遍历空余的 slot，寻找插入位置找到后插入并设置 RecordId。最后标记该位置已经被插入。

markSlotUsed() 修改 head 表示 tuple 被使用。

deleteTuple() 依旧是判断当前页面 tid 和待删除 tuple 的 TupleDesc 是否匹配。然后根据待删除的 tuple 找到 RecordId 判断是否存在，最后根据索引判断 slot

是否被使用，如果使用就删除。

`insertTuple()` 如果当前没有页面就调用 `writePage` 在磁盘中创建空页。然后去 `BufferPool` 取页，接下来判断取到的页中是否含有空 `slot`，然后插入 `tuple`。

`deleteTuple()` 从 `BufferPool` 中取出 `page` 然后删除 `tuple`。

(3)

`HeapPage` 类

参数:

`boolean dirty`;: 脏页标志位

`TransactionId dirtyId`;: 产生脏页的事务 `id`

方法:

`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)`: 插入 `tuple`，选择一个空的 `slot` 插入 `tuple`，同时修改该 `slot` 对应的 `bitmap`，表示该 `slot` 已被占用。

`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

方法:

`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

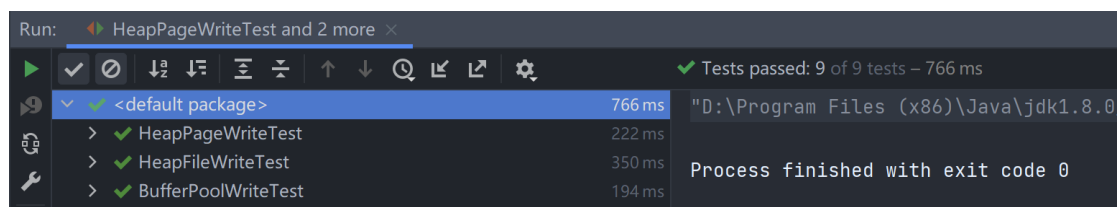
InsertTuple 中的 moveHead、addToHead 等操作是为了让将 pageId 所对应的 page 节点移动到链表的最前端，详见 exercise5 的页面置换策略

方法：

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

(4) 测试截图



2.4. Exercise 4 Insertion and Deletion

(1) 实现 execution/Insert.java 和 execution/Delete.java 并通过 InsertTest 和 InsertTest, DeleteTest system tests

(2) 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。

(3) Insert 类

实现了 Operator 接口，调用 BufferPool 的 insertTuple()方法向给定的表中插入 tuple

参数：

TransactionId transactionId;: 执行插入操作的事务 id

OpIterator child;: 待插入的 tuple 的迭代器

int tableId;: tuple 插入的表 id

boolean isInserted; 标志位，避免 fetchNext 操作可以无限制的向下取

TupleDesc tupleDesc;: fetchNext()会返回一个表示插入了多少 tuple 的一个 tuple, tupleDesc 是该 tuple 的属性行 fieldTypes == {Type.INT_TYPE}、fieldNames == { “numbers of instered tuples” }

方法:

public Insert(TransactionId t, OpIterator child, int tableId): 初始化方法

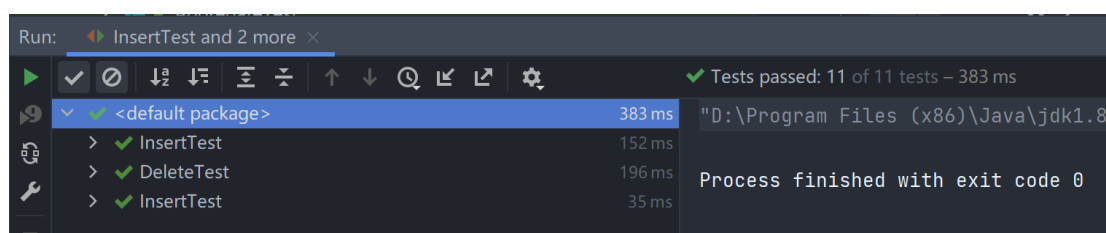
protected Tuple fetchNext(): 执行插入操作, 返回包含插入了多少 tuple 的一个 tuple

public TupleDesc getTupleDesc(): 返回 tupleDesc

Delete 类

和 Insert 基本类似

(4)



2.5. Exercise 5 Page Eviction

(1) 实现 BufferPool.java 中的 flushPage() 方法, 通过 EvictionTest system test writePage()、discardPage()同样需要实现。

(2) 该 exercise 实现了 BufferPool 中的页面置换策略, 采用 LRU (最近最少使用)。

LRU 的核心思想是: **最近使用的页面数据会在未来一段时期内仍然被使用, 已经很久没有使用的页面很有可能在未来较长的一段时间内仍然不会被使用。**为什么会有这样的结论, 是因为程序的运行具有时间上的局部性和空间上的局部性, 时间上的局部性是指某段已经执行过的程序指令可能在不久后会被执行, 空间上的局部性是指一旦程序访问了某个存储单元, 则不久之后, 其附近的存储单元也将被访问。

而我们去实现 LRU, 就是根据这个思想去写代码的。实现 LRU, 我们需要一个

双向链表和一个 `HashMap`，双向链表相当于一个队列，按淘汰顺序保存各个页面，为了加快页面的获取，而双向链表保证我们能够快速删除一个页面（结点）；为了加快获取结点的访问速度，我们还需要一个哈希表来存储 `key` 对应的页面（链表结点），而在我们的 `BufferPool` 中也是用 `pageId` 的 `hashCode` 作为 `key`，`Page` 作为 `value` 的。我们对页面的访问可以分为 `get` 和 `insert` 两种：

1.对于 `get` 操作，我们需要从 `hashmap` 中查找是否存在，如果存在则获取对应的结点，并依据结点在链表中删除并加入队尾；如果不存在返回 `null`；

2.对于 `insert` 操作，我们需要从 `hashmap` 中查找是否存在，如果存在，我们需要获取对应的结点并从链表中删除结点，并把结点加入队尾；如果不存在，则需要以（`key`，`val`）的形式加入哈希表中，并把结点加入链表尾部，如果哈希表的页面数超过了给定的容量，那么需要把链表头部的结点删掉，并根据其 `key` 从哈希表中去除。

（3）

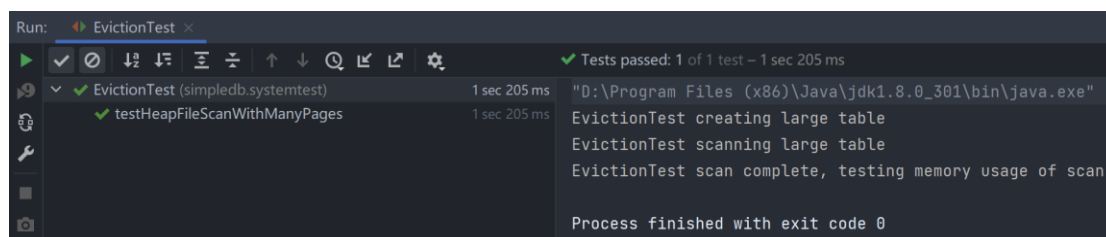
`public Page getPage(TransactionId tid, PageId pid, Permissions perm)`：当从 `BufferPool` 中读取 `page` 时，若 `BufferPool` 中有要读取的 `page` 则将其对应的 `LinkedListNode` 移动到头部，若没有则去硬盘中读取，并将其生成 `LinkedListNode` 放在链表的头部，如果 `BufferPool` 已满，则调用 `evictPage()` 置换 `BufferPool` 中的页面

`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

（4）测试截图



3. Lab3 Optimizer

这个 lab 大致要实现一个查询优化器，实现一个选择性估计框架和一个基于成本(Selinger)的优化器。

实现 TableStats 类中的方法，使其能够使用直方图（IntHistogram 类提供的骨架）或你设计的其他形式的统计数据来估计过滤器的选择性和扫描的成本。

实现 JoinOptimizer 类中的方法，使其能够估计 join 的成本和选择性。

编写 JoinOptimizer 中的 orderJoins 方法。这个方法必须为一系列的连接产生一个最佳的顺序（可能使用 Selinger 算法），给定前两个步骤中计算的统计数据。

基于开销优化器的主要思想：评估不同查询计划下的成本，根据成本选择最佳的排列和连接方式。精确的估计是很难的，这个实验只关注连接序列和基本表访问的成本。

根据 table 的统计数据来估计不同查询计划的开销。通常一个计划的成本与 intermediate joins 和 tuple 数量，以及 selectivity of filter 和 join predicates 的选择性有关。

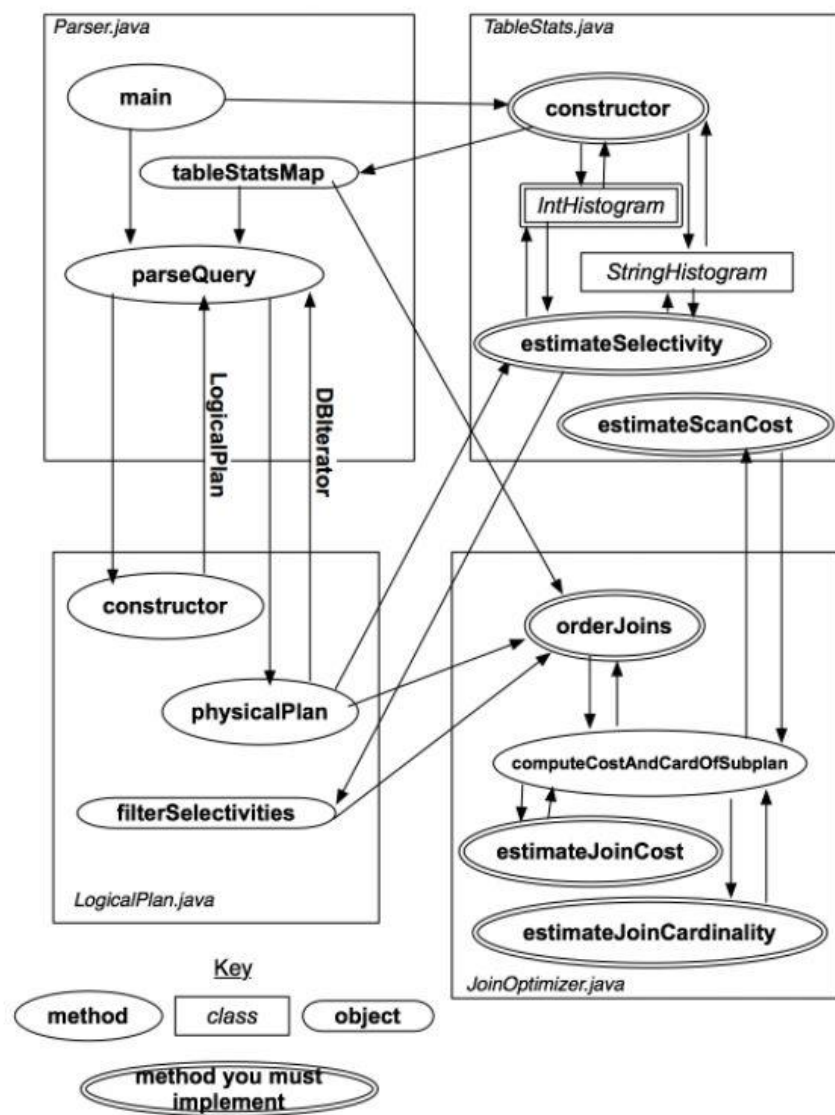
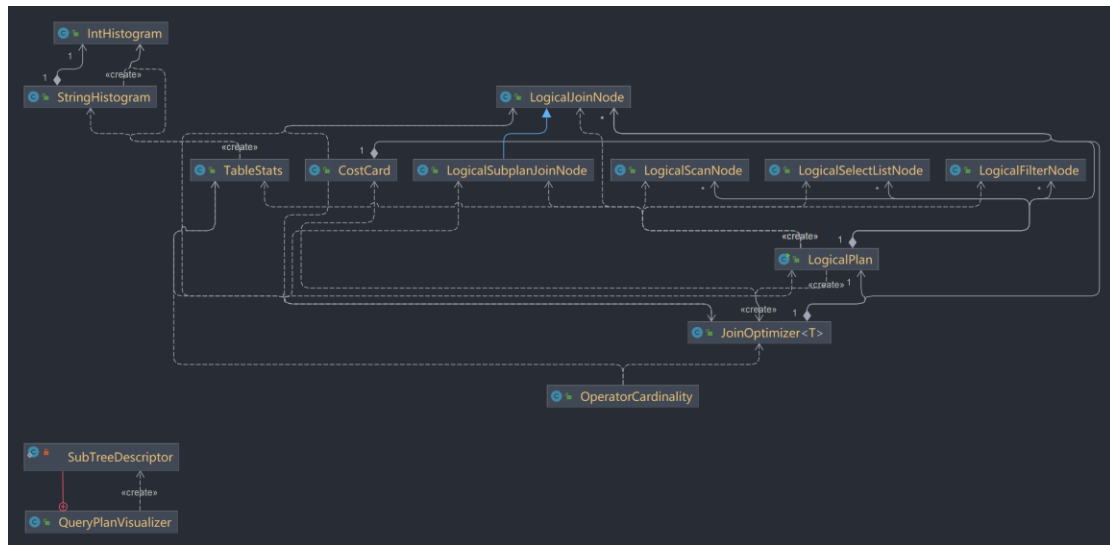
根据统计数据以最佳方式排列连接和选择，并从几个备选方案中选择连接算法的最佳实现。

当使用嵌套循环连接时，记得两个表 t1 和 t2（其中 t1 是外表）之间的连接成本是简单的。

$$\begin{aligned} \text{joincost}(t1 \text{ join } t2) &= \text{scancost}(t1) + \text{ntups}(t1) \times \text{scancost}(t2) // \text{IO cost} \\ &\quad + \text{ntups}(t1) \times \text{ntups}(t2) // \text{CPU cost} \end{aligned}$$

Here, ntups(t1) is the number of tuples in table t1.

这里，ntups(t1)是表 t1 中 tuples 的数量。首先要计算出开销，而开销由 I/O 开销和 CPU 开销两部分组成。其中需要用到表中 tuple 的数量。下面是如何统计 tuple 数。



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:

1.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

2.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。

3.1. Exercise 1 and 2 Filter Selectivity

(1) 实现 IntHistogram 并通过 IntHistogramTest。

实现 TableStats 并通过 TableStatsTest。实现 TableStats 构造函数

(2) 针对一个属性构建一张直方图, 横坐标代表属性对应范围, 纵坐标代表对应范围内 tuple 的数量。

其实就是计算对应条件下 tuple 的数量对于总数的占比。

占比计算 $(h / w) / ntups$ ntups 是纵坐标的累加和, 也就是 tuple 的总数。

(h / w) 表示桶中含有值常数的 tuples 的预期数量。h 的表示桶中 tuple 的总数, 并不是均匀的高度!!!

部分区间的占比: $b_part = (b_right - const) / w_b$ $b_f = h_b / ntups$ $b_f \times b_part$
addValue(int v)

根据输入数据构建直方图的分部, 计算出对应桶序号累加即可。

`estimateSelectivity(Predicate.Op op, int v)` 估计

这个类用来计算占比。具体的计算规则是根据运算符 `op` 判断(大于, 小于, 等于...), `v` 就是 `const`, 遍历。例如 `op` 是大于, `v` 是 3, 那么就是计算横坐标大于 3 所有 `tuple` 个数除以总 `tuple` 个数(`ntuple`)。也就是大于 3 `tuple` 占总 `tuple` 的百分比。

为 `table` 的每一个 `field` 构建一张直方图。

根据 `tableid` 拿到 `table`, 然后遍历 `table` 的每个字段(`field`)构建直方图。注意 `field` 分为整数和字符串两种类型, 分别用 `map` 来存。

首先获取每一列对应的内容, 放入 `list` 中。然后获取所有列的内容, 一列就是一个 `field`, 一列生成一个直方图。

`estimateScanCost()`

IO 成本是页数乘上单页 IO 的开销。

`estimateTableCardinality()`

`tuple` 总数乘上系数 (`selectivityFactor`)。

`estimateSelectivity()`

根据输入的参数来估计 `Selectivity`, 三个参数分别是待估计的字段, 比较符号, `const`。区分 `field` 的 `int` 和 `string` 分别调用 `estimateSelectivity()` 即可。

(3)

`IntHistogram` 类

参数:

`int[] buckets`;: 直方图中的桶, 用于记录每个桶的高度

`int min`;: 直方图的最小值

`int max`;: 直方图的最大值

`double width`;: 桶的宽度

`int tuplesNum`;: 构造此直方图的记录总数

方法:

`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

TableStats 类

参数:

`HeapFile table`: 需要进行数据统计的表

`int ioCostPerPage`: 读取每页的 IO 成本

`int tupleNum`: 表中 tuple 的总数

`int pageNum`: 表中 page 的总数,用于计算扫描成本

`HashMap<Integer,IntHistogram> integerIntHistogramMap`: 整型字段与其直方图的映射

`HashMap<Integer,StringHistogram> stringIntHistogramMap`: 字符串型字段与其直方图的映射

`HashMap<Integer,Integer> maxField`: 字段与该字段中最大值的映射

`HashMap<Integer,Integer> minField`: 字段与该字段中最小值的映射

`ArrayList<Tuple> tuples`: 表中的所有 tuples

`TupleDesc tupleDesc`: 表的属性行

方法:

`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

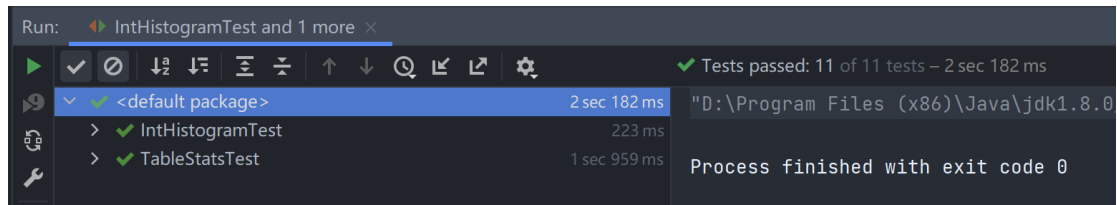
`double estimateScanCost()`: 估计扫描成本

`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

(4) 测试截图



3.2. Exercise 3 Join Cardinality

(1) 编写 JoinOptimizer 并通过 JoinOptimizerTest 中的 estimateJoinCostTest 和 estimateJoinCardinality 即可。实现 estimateJoinCost() 方法，估计 join 的成本。

(2) 计算公式：

$$\text{joincost}(t1 \text{ join } t2) = \text{scancost}(t1) + \text{ntups}(t1) \times \text{scancost}(t2) // \text{IO cost} + \text{ntups}(t1) \times \text{ntups}(t2) // \text{CPU cost}$$

Nested-loop (NL) join 是所有 join 算法中最 naive 的一种。假设有两张表 R 和 S，NL join 会用二重循环的方法扫描每个(r, s)对，如果行 r 和行 s 满足 join 的条件，就输出之。显然，其 I/O 复杂度为 $O(|R||S|)$ 。随着参与 join 的表个数增加，循环嵌套的层数就越多，时间复杂度也越高。因此虽然它的实现很简单，但效率也较低。

总结：成本 (cost) 分为 I/O 成本和 CPU 成本。I/O 成本是扫描表时和磁盘交互所产生的，而 CPU 成本是判断数据是否符合条件所产生的。其中 cost1 是扫描 t1 的 I/O 成本，cost2 同理。因为是 NL join 所以总的 I/O 开销就是 $\text{cost1} + \text{card1} * \text{cost2}$ 。而 CPU 开销则是 $\text{card1} * \text{card2}$ 。总成本相加即可。

estimateJoinCardinality 估计 join 后产生的 tuple 数。lab3.md 中 2.2.4 Join

Cardinality 部分有详细解释。

Cardinality 表示一系列数据中数据的重复程度，如果等于 1 那么数据没有重复的，如果等于 0 那么全部都重复，其他情况加载 [0, 1] 之间。

对于等价连接() 其中一个属性是主键时，由连接产生的 tuples 的数量不能大于非主键属性的 cardinality。只要保证这一点成立即可，所以其中一个是主键的话就选择一个小的，两个都是主键的话选择小的，两个都不是主键的话选择大的。这块的实现很灵活。

对于非等价连接文档给了公式 $\text{card1} * \text{card2} * 0.3$ 。

(3) 方法:

`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

`public static int estimateTableJoinCardinality()`: 估算两个表连接后的基数,

对于等值连接:

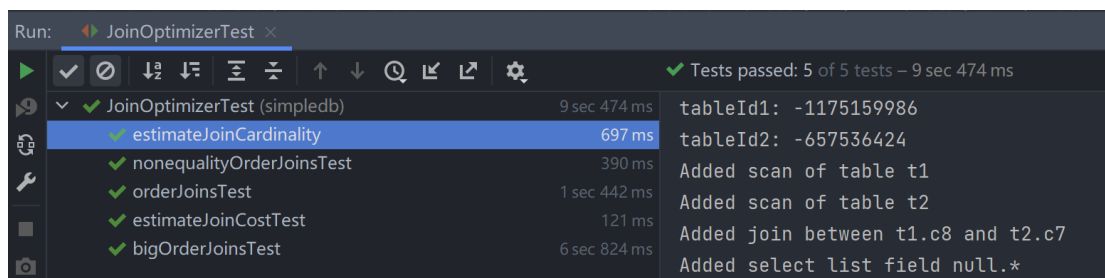
当一个属性是 `primary key`, 把 `non-primary key` 属性的记录数取做连接后的基数。因为主键是唯一的, 也就是说非主键的每一条记录最多能连接一个主键的记录数。(为什么不能选主键记录数当作基数呢? 如果非主键字段的记录数远远小于主键字段的记录数, 那这个基数可以是十分不准确了。)

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。

(4) 测试截图



3.3. Exercise 4 Join Ordering

(1) 实现 `JoinOptimizer.java` 中的 `orderJoins` 方法并通过 `JoinOptimizerTest` 和系统测试 `QueryTest`。

(2) `ex3` 实现了开销估计和基数个数的估计。这个练习则是在多表连接的情况下根据开销分析选择最优的连接顺序。直接枚举的话复杂度是 $O(n!)$ 。此处选择了一种 DP 的方法将复杂度降低到了 $O(2^n)$ 。

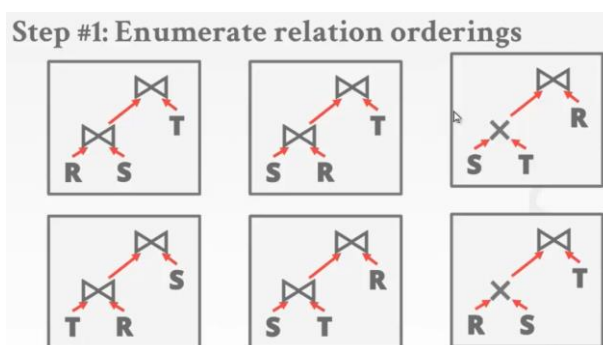
`JoinOptimizer` 中的 `join` 属性是一个队列, 其中存的都是 `LogicalJoinNode` 对象。

PlanCache 类，用来缓存 Selinger 实现中所考虑的连接子集的最佳顺序，接下来的任务就是找到最佳顺序。

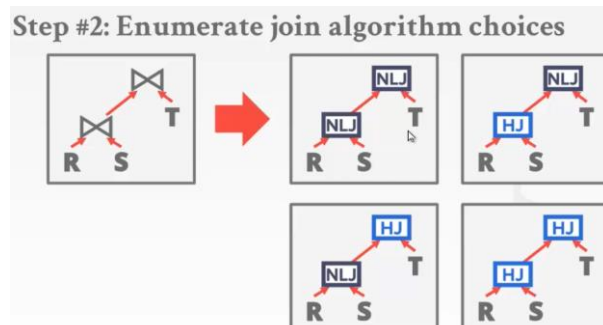
enumerateSubsets(joins, i); 其中 i 表示子集中的子集的元素个数。例如 a,b,c 三张表，当 i=1 时，返回数据大致形态 set(set(ab), set(ac), set(bc))，注意 ab 是一个 LogicalJoinNode 所以尺寸是 1。如果 i=2，那么返回的数据类似 set(set(ab, c), set(ac, b), set(bc, a))。可以优化为回溯，避免创建大量对象。

DP step:

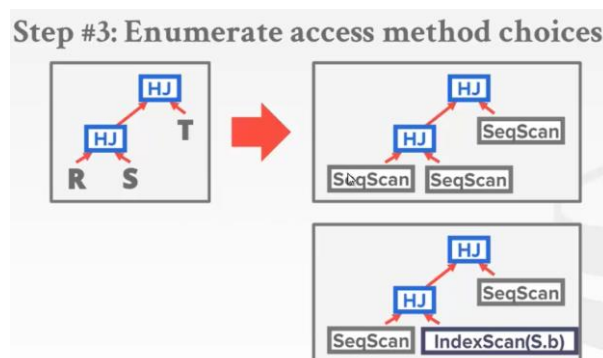
1. 首先枚举左深树的组合顺序。



2. 枚举不同顺序下不同 Join 算法的开销。



3. 枚举每一个表的读表方式的开销。



4. 暴力计算。

小于 12 个表用 DP，否则开销巨大。

(3) 参数:

`List<LogicalJoinNode> joins`: `joins` 是一系列 join 节点的集合, 而不是需要连接的表集合。比如 $r1 \bowtie r2 \bowtie r3$, `logicalJoinNode1 = r1 \bowtie r2`, `logicalJoinNode2 = r2 \bowtie r3`

辅助方法:

涉及到的类:

`CostCard` 类:

`List<LogicalJoinNode> plan`: 按照某一顺序连接的查询计划

`double cost`: 该连接顺序下的代价

`int card`: 该连接顺序下产生的基数

`PlanCache` 类: 类似于 DP 数组

`Map<Set<LogicalJoinNode>,List<LogicalJoinNode>> bestOrders`: 关系集合与其最优连接顺序的映射

`Map<Set<LogicalJoinNode>,Double> bestCosts`: 关系集合与其最优连接顺序的代价的映射

`Map<Set<LogicalJoinNode>,Integer> bestCardinalities`: 关系集合与其最优连接顺序的基数的映射

对集合进行拆分, 得到具有 `size` 个 `logicalJoinNode` 节点的所有集合。

`CostCard computeCostAndCardOfSubplan()`: 计算子计划的查询代价

`printJoins`: 将连接计划进行显示的图形表示

需要实现的方法:

`public List<LogicalJoinNode> 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

可优化的方法:

`enumerateSubsets(joins, i)`:

求子集的方法, 需要不断的创建新对象, 当 `size` 很大时, 时间复杂度非常高。

可以用回溯的方法求子集, 降低其时间复杂度。

优化后代码如下:

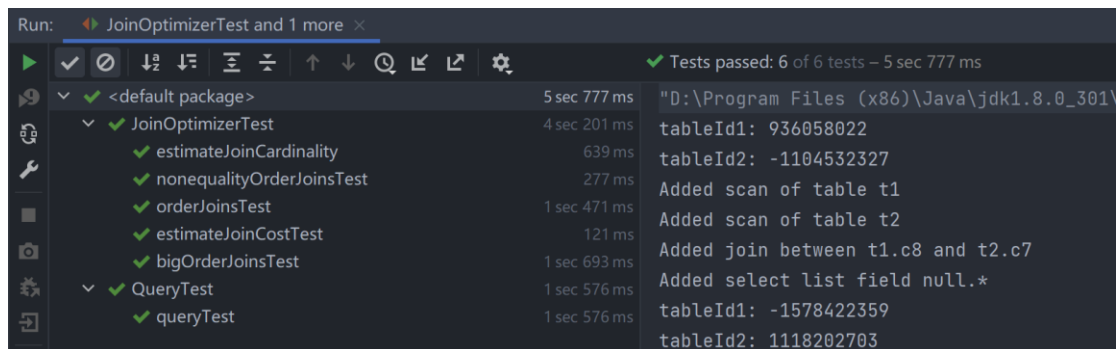

```

private <T> void dfs(List<T> list, int cur, int size, Deque<T> subset,
Set<Set<T>> subsets) {
    if (subset.size() == size) {
        subsets.add(new HashSet<>(subset));
    }
    for (int i = cur; i < list.size(); i++) {
        subset.addLast(list.get(i));
        dfs(list, i+1, size, subset, subsets);
        subset.removeLast();
    }
}

public <T> Set<Set<T>> enumerateSubsets(List<T> v, int size) {
    Set<Set<T>> els = new HashSet<>();
    Deque<T> subset = new ArrayDeque<>();
    dfs(v, 0, size, subset, els);
    return els;
}

```

(4) 测试截图:



4. Lab4 Transaction

在这个实验中，你将在 SimpleDB 中实现一个简单的基于锁的事务系统。你将在代码中的适当位置添加锁和解锁调用，以及跟踪每个事务所持有的锁的代码，并在需要时授予事务锁。

事务

事务是一组以原子方式执行的数据库操作（例如，插入、删除和读取）；也就是说，要么所有的动作都完成了，要么一个动作都没有完成，而且对于数据库的外部观察者来说，这些操作没有作为一个单一的、不可分割的操作的一部分完成，这一点是不明显的。

ACID 属性

原子性：通过两段锁协议和 BufferPool 的管理实现 simpleDB 的原子性

一致性：通过原子性实现事务的一致性，simpleDB 中没有解决其他一致性问题（例如，键约束）

隔离性：严格的两段锁提供隔离

持久性：事务提交时将脏页强制写进磁盘

恢复和缓冲管理

实施 "不偷不抢" (NO STEAL/FORCE) 的缓冲区管理政策：

不应该从缓冲池中驱逐脏的（更新的）页面，如果它们被一个未提交的事务锁定（this is NO STEAL）。

在事务提交时，应该把脏页强制到磁盘上（例如，把页写出来）（this is FORCE）。

假设 SimpleDB 在处理 transactionComplete 命令时不会崩溃。请注意，这三点意味着不需要在这个实验中实现基于日志的恢复，因为永远不需要撤销任何工作（你永远不会驱逐脏页），也不需要重做任何工作（在提交时强制更新，不会在提交处理期间崩溃）。

锁的赋予

将需要增加对 SimpleDB 的调用（例如在 BufferPool 中），允许调用者代表特定事务请求或释放特定对象的（共享或独占）锁。（假设页级锁），锁的工作方式：在事务可以读取一个对象之前，它必须拥有一个共享锁。

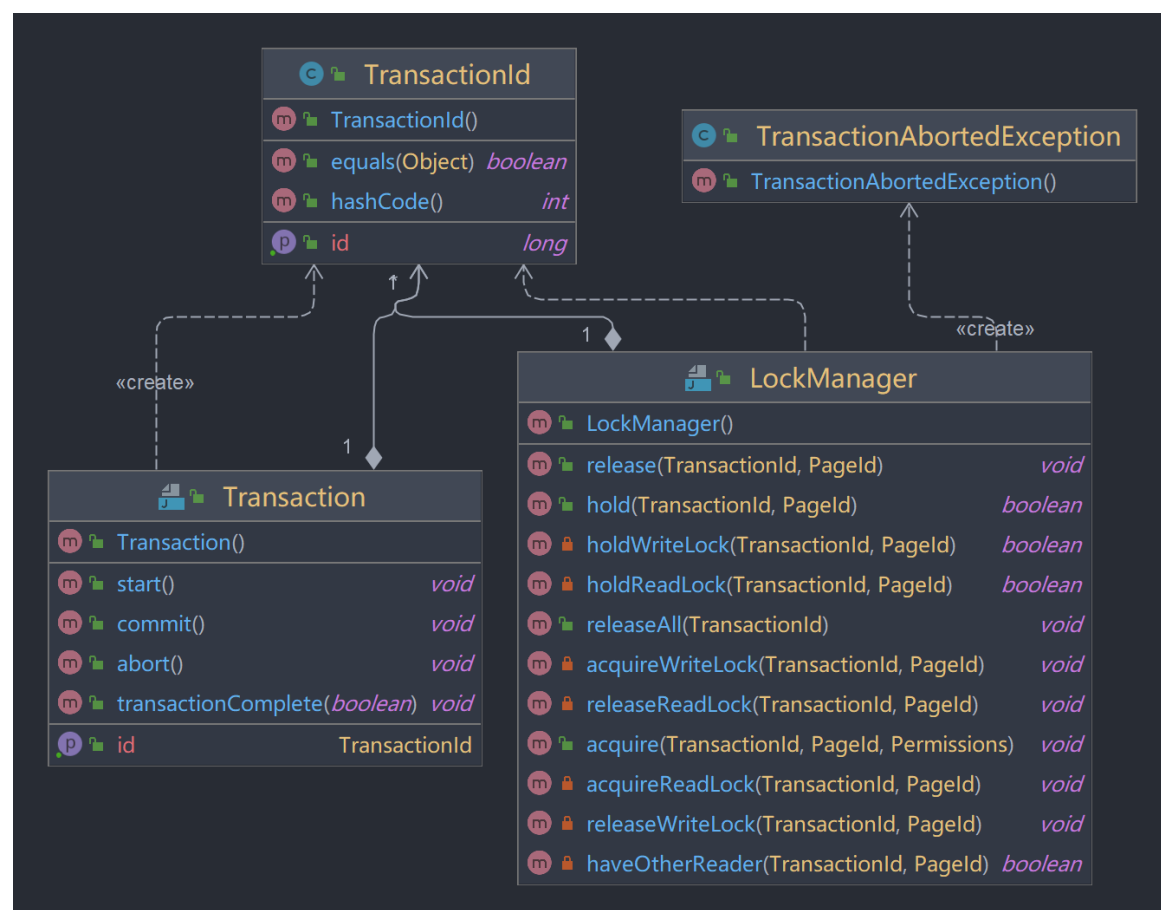
在一个事务可以写一个对象之前，它必须有一个排他锁。

多个事务可以在一个对象上拥有一个共享锁。

只有一个事务可能对一个对象具有排他锁。

如果对象 *o* 上只有事务 *t* 持有共享锁，则 *t* 可以将其对 *o* 的锁升级为排他锁。

如果一个事务请求一个不能立即授予的锁，代码应该阻塞，等待该锁变得可用（即，被另一个在不同线程中运行的事务释放）。



4.1. Exercise 1 Granting Locks

(1) 编写 **BufferPool** 中获取和释放锁的方法。

修改 **getPage()**，使其在返回页面前阻塞并获得所需的锁。

实现 **unsafeReleasePage()**。这个方法主要用于测试，以及在事务结束时使用。

实现 **holdsLock()**，以便练习 2 中的逻辑能够确定一个页面是否已经被事务锁定。

(2) **exercise1** 需要做的是在 **getPage** 获取数据页前进行加锁，这里我们使用一个 **LockManager** 来实现对锁的管理，**LockManager** 中主要有申请锁、释放锁、

查看指定数据页的指定事务是否有锁这三个功能，其中加锁的逻辑比较麻烦，需要基于严格两阶段封锁协议去实现。事务 *t* 对指定的页面加锁时，思路如下：

1.锁管理器中没有任何锁或者该页面没有被任何事务加锁，可以直接加读/写锁；如果 *t* 在页面有锁，分以下情况讨论：

2.1 加的是读锁：直接加锁；

2.2 加的是写锁：如果锁数量为 1，进行锁升级；如果锁数量大于 1，会死锁，抛异常中断事务；

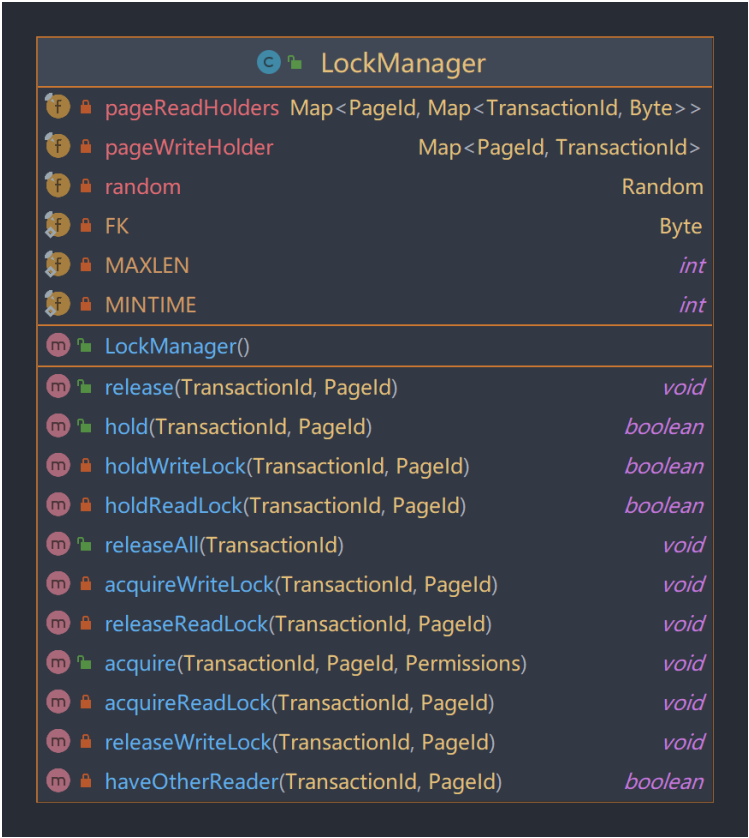
如果 *t* 在页面无锁，分以下情况讨论：

3.1 加的是读锁：如果锁数量为 1，这个锁是读锁则可以加，是写锁就 wait；如果锁数量大于 1，说明有很多读锁，直接加；

3.2 加的是写锁：不管是多个读锁还是一个写锁，都不能加，wait

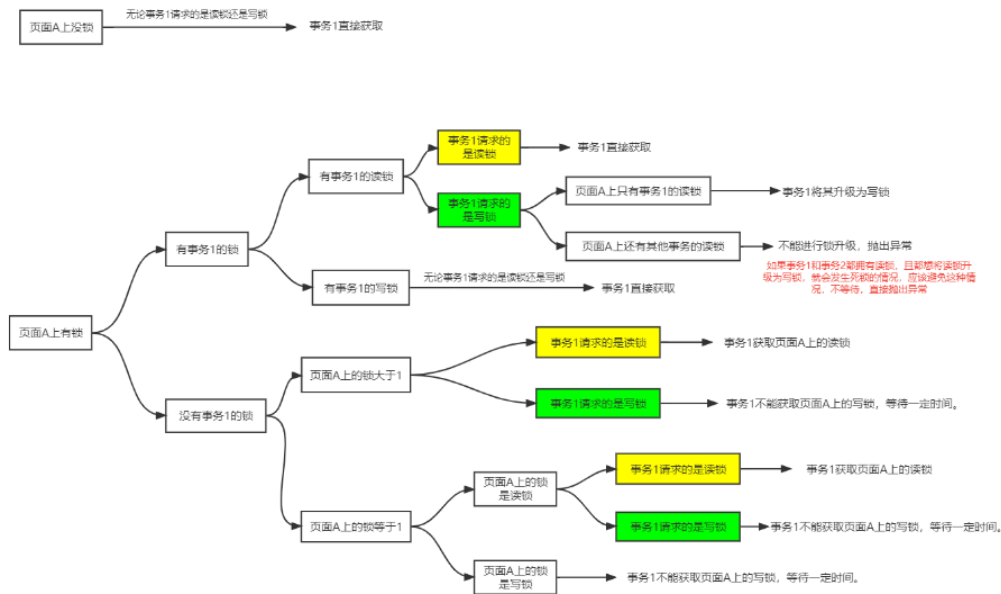
其它两个就比较容易了。

(3) LockManager 类



| LockManager | |
|---|---------------------------------------|
| pageReadHolders | Map<PageId, Map<TransactionId, Byte>> |
| pageWriteHolder | Map<PageId, TransactionId> |
| random | Random |
| FK | Byte |
| MAXLEN | int |
| MINTIME | int |
| LockManager() | |
| release(TransactionId, PageId) | void |
| hold(TransactionId, PageId) | boolean |
| holdWriteLock(TransactionId, PageId) | boolean |
| holdReadLock(TransactionId, PageId) | boolean |
| releaseAll(TransactionId) | void |
| acquireWriteLock(TransactionId, PageId) | void |
| releaseReadLock(TransactionId, PageId) | void |
| acquire(TransactionId, PageId, Permissions) | void |
| acquireReadLock(TransactionId, PageId) | void |
| releaseWriteLock(TransactionId, PageId) | void |
| haveOtherReader(TransactionId, PageId) | boolean |

Acquire 流程如下：



参数:

`private final Map<PageId, Map<TransactionId, Byte>> pageReadHolders;` : `pageId` 与读锁的映射, 记录 `page` 上现存的读锁

`private final Map<PageId, TransactionId> pageWriteHolder;` : `pageId` 与写锁的映射, 记录 `page` 上现存的写锁

方法:

`public void acquire(TransactionId tid, PageId pid, Permissions perm)`, 根据 `perm` 的类型分别获取对应的锁, `READ_ONLY` 执行 `acquireReadLock(tid, pid)`; `READ_WRITE` 执行 `acquireWriteLock(tid, pid)`;

`private void acquireReadLock(TransactionId tid, PageId pid)`, 申请读锁

`private void acquireWriteLock(TransactionId tid, PageId pid)`, 申请写锁

`private boolean haveOtherReader(TransactionId tid, PageId pid)`, 判断是否由其他读者

`public void release(TransactionId tid, PageId pid)`, 释放事务 `tid` 上的页 `pid` 上的锁

`private void releaseReadLock(TransactionId tid, PageId pid)`, 释放读锁

`private void releaseWriteLock(TransactionId tid, PageId pid)`, 释放写锁

`public boolean hold(TransactionId tid, PageId pid)`, 判断是否持有锁 (写锁或读锁)

`private boolean holdReadLock(TransactionId tid, PageId pid)`, 判断是否持有读锁

`private boolean holdWriteLock(TransactionId tid, PageId pid)`, 判断是否持有写锁

`public void releaseAll(TransactionId tid)`, 释放两个 `Holder` 上的所有锁。

`BufferPool` 类

方法:

`public Page getPage(TransactionId tid, PageId pid, Permissions perm)`, 需要修改的是在返回 `page` 前, 调用 `lockManager.acquire(tid, pid, perm)`;来获取锁。

`public void unsafeReleasePage(TransactionId tid, PageId pid)`, 调用 `lockManager.release(tid, pid)`;即可

`public boolean holdsLock(TransactionId tid, PageId p)`, 返回 `return lockManager.hold(tid, p)`;即可

(4) 需完成下一个 exercise 再测试

4.2. Exercise 2 Lock Lifetime

(1) 这个 exercises 主要让我们考虑什么时候加锁, 什么时候解锁, 在这一点上, 你的代码应该通过 `LockingTest` 中的单元测试。

(2) 在读取任何页面或元组之前获得一个共享锁 (读锁), 在写入任何页面或元组之前获得一个独占锁 (写锁)。请注意, 在对 `HeapFile.insertTuple()`和 `HeapFile.deleteTuple()`的实现, 以及 `HeapFile.iterator()`返回的迭代器的实现应该使用 `BufferPool.getPage()`访问页面。检查这些 `getPage()`的不同用法是否传递了正确的权限对象 (例如, `Permissions.READ_WRITE` 或 `Permissions.READ_ONLY`)。检查实现的 `BufferPool.insertTuple()`和 `BufferPool.deleteTupe()`是否在它们访问的任何页面上调用 `markDirty()`。这个和 exercise1 实际上可以放在一块。

(3)

`BufferPool` 的 `insertTuple()`实现:

```
public void insertTuple(TransactionId tid, int tableId, Tuple t)
    throws DbException, IOException, TransactionAbortedException {
    // some code goes here
    // not necessary for lab1
    List<Page> pageList =
Database.getCatalog().getDatabaseFile(tableId).insertTuple(tid, t);
    for (Page page : pageList) {
```

```

        addToBufferPool(page.getId(), page);
    }
}

```

BufferPool 的 deleteTuple()实现:

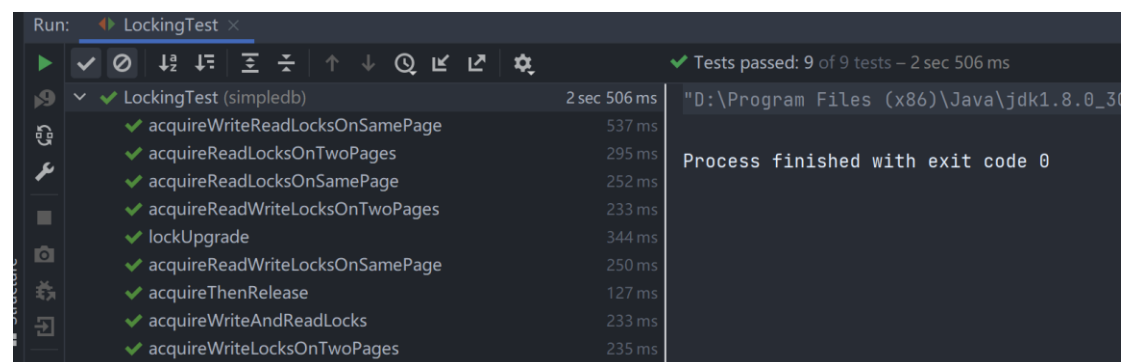
```

public void deleteTuple(TransactionId tid, Tuple t)
    throws DbException, IOException, TransactionAbortedException {
    // some code goes here
    // not necessary for lab1
    DbFile dbFile =
Database.getCatalog().getDatabaseFile(t.getRecordId().getPageId().get
TableId());
    dbFile.deleteTuple(tid, t);
}

```

同时 HeapFile 的 insertTuple()和 deleteTuple()以及 HeapFile.iterator()在实现时均调用了 BufferPool 的 getPage()。

(4) 测试截图



4.3. Exercise 3 Implementing NO STEAL

(1) 在 BufferPool 的 evictPage 方法中实现必要的页面驱逐逻辑，而不驱逐脏页。

(2) 一个事务的修改只有在它提交之后才会被写入磁盘。这意味着我们可以通过丢弃脏页并从磁盘重读来中止一个事务。因此，我们必须不驱逐脏页。这个策略被称为 NO STEAL。

将需要修改 BufferPool 中的 evictPage 方法。特别是，它必须永远不驱逐一个脏页。如果驱逐策略倾向于驱逐一个脏页，则不得不找到一种方法来驱逐一个替代页。在缓冲池中的所有页面都是脏的情况下，应该抛出一个 DbException。如果驱逐策略驱逐了一个干净的页面，要注意事务可能已经持有被驱逐的页面

的任何锁，并在实现中适当地处理它们。

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。

(3) evictPage() 代码如下：

```
private synchronized void evictPage() throws DbException,
IOException {
    // some code goes here
    // not necessary for lab1
    //assert pages.size() == numPages;
    for (Page page : pagesMap.values()) {
        if (page.isDirty() == null) {
            pagesMap.remove(page.getId());
            return;
        }
    }
    throw new DbException("No clean page to EVICT");
}
```

(4) 暂无测试

4.4. Exercise 4 Transactions

(1) 实现 transactionComplete()

通过 TransactionTest 单元测试和 AbortEvictionTest 系统测试

(2) 如果 commit 那么就把 tid 对应的所有页面持久化，也就是写入磁盘否则把该事物相关的页面加载进缓存中。

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。

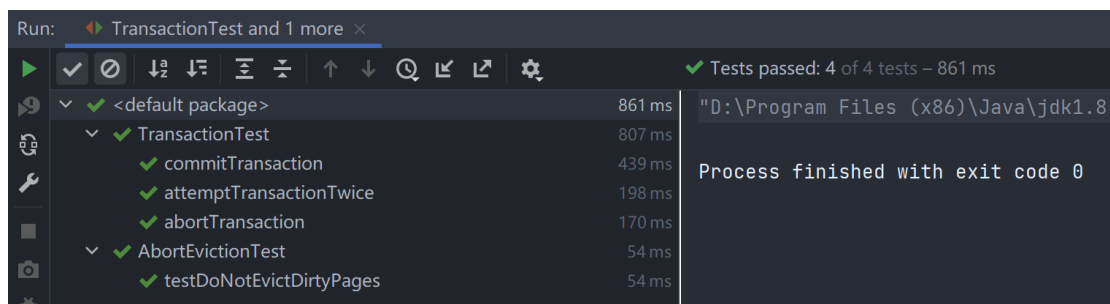
(3) transactionComplete() 代码：


```

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 {
        for (Page page : pagesMap.values()) {
            if (tid.equals(page.isDirty())) {
                discardPage(page.getId());
            }
        }
    }
    LockManager.releaseAll(tid);
}
}

```

(4) 测试截图



4.5. Exercise 5 Deadlocks and Aborts

(1) 在 src/simpledb/BufferPool.java 中实现死锁检测或预防。通过 DeadlockTest 和 TransactionTest 系统测试。

(2) 什么时候会发生死锁？

1.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；

2.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。

死锁的解决方案？一般有两种解决方案：

1.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。

2.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。

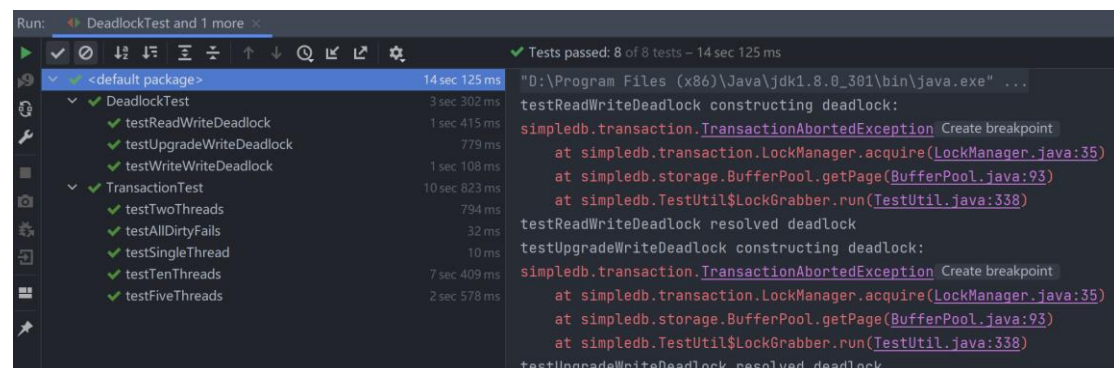
本文通过对每个事务设置一个获取锁的超时时间，如果在超时时间内获取不到锁，我们就认为可能发生了死锁，将该事务进行中断。这以简单的方法实现思索和终止。

(3) 在 LockManager 类中 acquireReadLock 和 acquireWriteLock 实现了超时检测

内部部分代码如下：

```
Thread thread = Thread.currentThread();
Timer timer = new Timer(true);
timer.schedule(new TimerTask() {
    @Override
    public void run() {
        thread.interrupt();
    }
}, MINTIME + random.nextInt(MAXLEN));
while (pageWriteHolder.containsKey(pid)) {
    pid.wait(5);
}
timer.cancel();
```

(4) 测试截图



5. B+-Tree Index

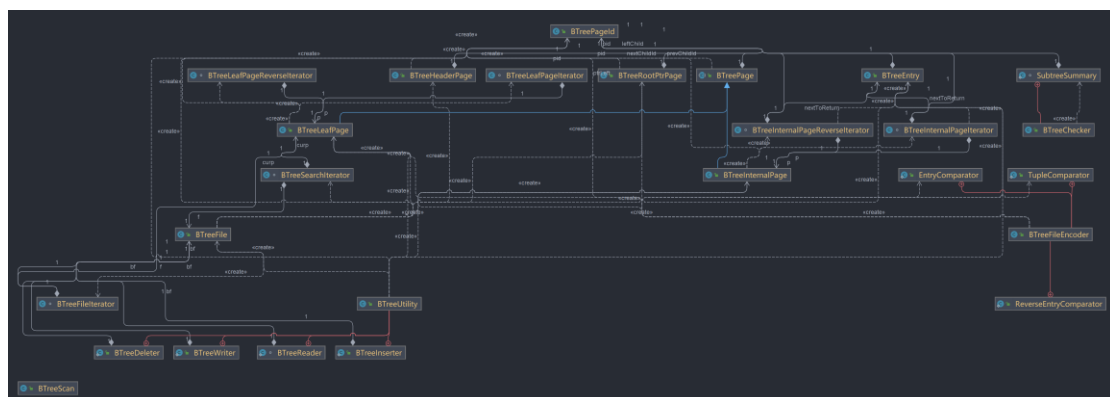
BTreeFile 由四种不同的页面组成,

BTreeInternalPage.java 内部页

BTreeLeafPage.java 叶子页

BTreePage.java 包含了叶子页和内部页的共同代码

BTreeHeaderPage.java 跟踪文件中哪些页正在使用



实现 B+ 树索引的查询、节点分裂、兄弟节点中元素的重新分配、兄弟节点的合

并。

- 根据 B+树的特性去查找所需元素。(exercise1 的内容)
- 插入元素时会出现分裂节点的情况，实现内部节点和叶子节点的分裂。
(exercise2 的内容)
- 删除元素时根据兄弟节点的情况会进行元素的重新分配、兄弟节点的合并。
实现内部节点和叶子节点的重新分配、合并。(exercise3、4 的内容)

辅助类：

- BTreePageId: BTreeInternalPage、BTreeLeafPage、BTreeHeaderPage、BTreeRootPtrPage 的唯一标识符，主要有三个属性
 - tableid: 该 page 所在 table 的 id。
 - pgNo: 该 page 所在 page 的序号 (table 中的第几个页)。
 - pgcateg: 用于标识 BTreePage 的类型。
- BTreeInternalPage: B+树的内部节点
 - byte[] header;: 记录 slot 的占用情况
 - Field[] keys;: 存储 key 的数组
 - int[] children;: 存储 page 的序号，用于获取左孩子、右孩子的 BTreePageId
 - int numSlots;: 内部节点中能存储的指针的数量 (即 n，内部节点中最多能存储 key 的数量为 n-1)
 - int childCategory;: 孩子节点的类型 (内部节点或叶节点)
- BTreeLeafPage: B+树的叶节点
 - byte[] header;: 记录 slot 的占用情况
 - Tuple[] tuples;: 存储 tuple 的数组
 - int numSlots;: 叶节点中能存储的 tuple 数量 (即 n-1)
 - int leftSibling;: 左兄弟的 pageNo，用于获取左兄弟的 BTreePageId，为 0 则没有左兄弟
 - int rightSibling;: 右兄弟的 pageNo，用于获取右兄弟的 BTreePageId，为 0 则没有右兄弟
- BTreeEntry: 内部节点中的 entry，内部节点对 key 的查找、插入、删除、迭

代，都是以 entry 为单位的。

- Field key;: 内部节点中的 key
- BTreePageId leftChild;: 左孩子的 BTreePageId
- BTreePageId rightChild;: 右孩子的 BTreePageId
- RecordId rid;: 标识该 entry 所在的位置。(即该 entry 是哪个 page 中的)

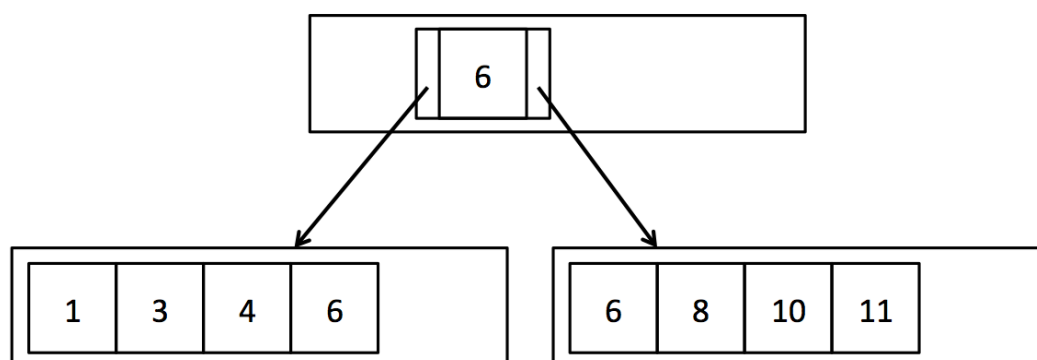
BTreeInternalPage 底层页面并不存储 BTreeEntry，而是存储 n-1 个 key 和 n 个指向子节点的指针（子页的 pageNo，父页与子页同处一个 table 中，知道子页的 pageNo 就能获取到子页）。

5.1. Exercise 1 Search

(1) 在 BTreeFile.java 中实现 findLeafPage() 方法，功能是给定一个特定的键值的情况下找到合适的叶子页。

通过 BTreeFileReadTest.java 中的所有单元测试和 BTreeScanTest.java 中的系统测试。

(2) 具体流程如下图：根节点是 6 是一个内部页，两个指针分别指向了叶子页。如果输入 1 那么 findLeafPage() 应当返回第一个叶子页。如果输入 8 那么应当返回第二个叶子页。如果输入 6 此时左右叶子页都含有 6，函数应当返回第一个叶子页，也就是左边的叶子页。



findLeafPage() 递归搜索节点，节点内部的数据可以通过

BTreeInternalPage.iterator() 访问。

当 key value 为空的时候，应当递归做左边的子页进而找到最左边的叶子页。

BTreePageId.java 中的 pgcateg() 函数检查页面的类型。可以假设只有叶子页和

内部页会被传递给这个函数。

`BTreeFile.getPage()` 和 `BufferPool.getPage()` 原理一样但需要一个额外的参数来跟踪脏页。

`findLeafPage()` 访问的每一个内部（非叶子）页面都应该以 `READ_ONLY` 权限获取，除了返回的叶子页面，它应该以作为函数参数提供的权限获取。这些权限在本实验中不重要但是后续实验中很重要。

具体步骤：

1. 获取数据页类型；
2. 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；
3. 如果不是则说明该页面是内部节点，将页面进行类型转换；
4. 获取内部节点的迭代器；
5. 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；
6. 找到第一个大于（或等于）`filed` 的 `entry`，然后递归其左孩子；
7. 如果到了最后一个页面，则递归其右孩子；

(3) `BTreeLeafPage findLeafPage`: 递归函数，在 B+ 树中查找可能包含字段 `f` 的叶节点。它使用只读权限锁定叶节点路径上的所有内部节点，并使用权限 `perm` 锁定叶节点。如果 `f` 为 `null`，它将查找最左边的叶节点，用于迭代器。当前节点如果是叶子节点，则返回 `pid` 对应的 `BTreeLeafPage`。否则向下递归找到 `f` 所在的叶子节点。当 `f = null` 时，返回最左侧的叶子结点。

参数：

`Map<PageId, Page> dirtypages`: 当创建新 `page` 或更改 `page` 中的数据、指针时，需要将其添加到 `dirtypages` 中

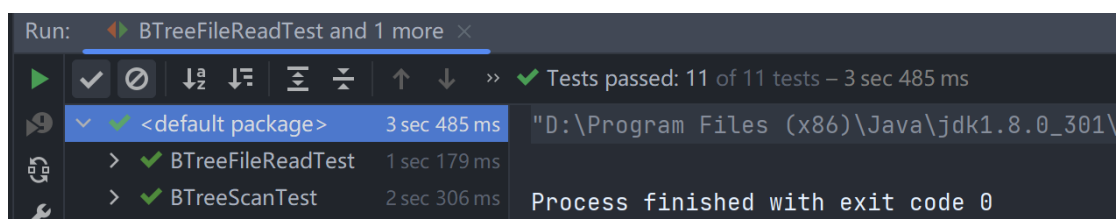
方法：

`getPage`: 调用 `getPage()` 获取 `page` 时，将检查页面是否已经存储在本地缓存 `dirtypage` 中，如果不再，则调用 `BufferPool.getPage` 去获取。`getPage()` 如果使用读写权限获取页面，也会将页面添加到 `dirtypages` 缓存中，因为它们可能很快就会被弄脏。这种方法的一个优点是，如果在一个元组插入或删除过程中多次访问相同的页面，它可以防止更新丢失。

具体代码如下：

```
private BTreeLeafPage findLeafPage(TransactionId tid, Map<PageId, Page>
dirtypages, BTreePageId pid, Permissions perm,
                                   Field f)
    throws DbException, TransactionAbortedException,
IOException {
    switch (pid.pgcateg())
    {
        case BTreePageId.LEAF:
            return (BTreeLeafPage) getPage(tid, dirtypages, pid,
perm);
        case BTreePageId.INTERNAL:
            BTreeInternalPage page = (BTreeInternalPage)
getPage(tid, dirtypages, pid, perm);
            Iterator<BTreeEntry> iterator = page.iterator();
            if (iterator == null || !iterator.hasNext()) {
                throw new DbException("No that entry");
            }
            if (f == null) return findLeafPage(tid, dirtypages,
iterator.next().getLeftChild(), perm, f);
            BTreeEntry entry = null;
            while (iterator.hasNext()) {
                entry = iterator.next();
                if (entry.getKey().compare(Op.GREATER_THAN_OR_EQ,
f)) {
                    return findLeafPage(tid, dirtypages,
entry.getLeftChild(), perm, f);
                }
            }
            return findLeafPage(tid, dirtypages,
entry.getRightChild(), perm, f);
        case BTreePageId.HEADER:
        case BTreePageId.ROOT_PTR:
        default:
            throw new DbException("not valid page");
    }
}
```

(4) 测试截图：



5.2. Exercise 2 Insert

(1) 在 `BTreeFile.java` 中实现 `splitLeafPage()` 和 `splitInternalPage()` 并通过 `BTreeFileInsertTest.java` 中的单元测试和 `systemtest/BTreeFileInsertTest.java` 中的系统测试。

(2) 通过 `findLeafPage()` 可以找到应该插入 tuple 的正确叶子页，但是页满的情况下插入 tuple 可能会导致页分裂，进而导致父节点分裂也就是递归分裂。

如果被分割的页面是根页面，你将需要创建一个新的内部节点来成为新的根页面，并更新 `BTreeRootPtrPage` 否则，需要以 `READ_WRITE` 权限获取父页，进行递归分割，并添加一个 entry。`getParentWithEmptySlots()` 对于处理这些不同的情况非常有用。

在 `splitLeafPage()` 中将键“复制”到父页上，页节点中保留一份。而在 `splitInternalPage()` 中，应该将键“推”到父页上，内部节点不保留。

当内部节点被分割时，需要更新所有被移动的子节点的父指针。

`updateParentPointers()` 很有用。

每当创建一个新的页面时，无论是因为拆分一个页面还是创建一个新的根页面，都要调用 `getEmptyPage()` 来获取新的页面。这是一个抽象函数，它将允许我们重新使用因合并而被删除的页面（在下一节涉及）。

`BTreeLeafPage.iterator()` 和 `BTreeInternalPage.iterator()` 实现了叶子页和内部页进行交互，除此之外还提供了反向迭代器 `BTreeLeafPage.reverseIterator()` 和 `BTreeInternalPage.reverseIterator()`。

`BTreeEntry.java` 中有一个 key 和两个 child pointers，除此之外还有一个 `recordId` 用于识别底层页面上键和子指针的位置。

具体步骤：

- 1、通过 `getEmptyPage` 创建一个 `newRightPage`,
- 2、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）。
- 3、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。

4、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。

5、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。

6、更新 page、newRightPage 的父指针。

7、返回 field 所在的页 (page 或 newRightPage)。

(3) 代码如下:

splitLeafPage:

```
public BTreeLeafPage splitLeafPage(TransactionId tid, Map<PageId, Page>
dirtypages, BTreeLeafPage page, Field field)
    throws DbException, IOException,
TransactionAbortedException {
    BTreeLeafPage rightPage = (BTreeLeafPage) getEmptyPage(tid,
dirtypages, BTreePageId.LEAF);
    int moveNum = page.getNumTuples() / 2;
    Iterator<Tuple> iterator = page.reverseIterator();
    for (int i = 0; i < moveNum; i++) {
        Tuple tuple = iterator.next();
        page.deleteTuple(tuple);
        rightPage.insertTuple(tuple);
    }
    Field midField = rightPage.iterator().next().getField(keyField);
    BTreeInternalPage parent = getParentWithEmptySlots(tid,
dirtypages, page.getParentId(), midField);

    rightPage.setParentId(parent.getId());
    page.setParentId(parent.getId());

    BTreeEntry newEntry = new BTreeEntry(midField, page.getId(),
rightPage.getId());
    parent.insertEntry(newEntry);

    rightPage.setLeftSiblingId(page.getId());
    rightPage.setRightSiblingId(page.getRightSiblingId());
    page.setRightSiblingId(rightPage.getId());

    if (rightPage.getRightSiblingId() != null) {
```

```

        BTreeLeafPage rrPage = (BTreeLeafPage) getPage(tid,
dirtypages, rightPage.getRightSiblingId(), Permissions.READ_WRITE);
        rrPage.setLeftSiblingId(rightPage.getId());
        dirtypages.put(rrPage.getId(), rrPage);
    }

    dirtypages.put(page.getId(), page);
    dirtypages.put(parent.getId(), parent);
    dirtypages.put(rightPage.getId(), rightPage);

    if (field.compare(Op.LESS_THAN_OR_EQ, midField)) {
        return page;
    } else {
        return rightPage;
    }
    //return null;
}

```

splitInternalPage:

```

public BTreeInternalPage splitInternalPage(TransactionId tid,
Map<PageId, Page> dirtypages,
        BTreeInternalPage page, Field field)
        throws DbException, IOException,
TransactionAbortedException {
    BTreeInternalPage rightPage = (BTreeInternalPage)
getEmptyPage(tid, dirtypages, BTreePageId.INTERNAL);

    int moveNum = page.getNumEntries() / 2;

    Iterator<BTreeEntry> iterator = page.reverseIterator();
    if (iterator == null || !iterator.hasNext())
        throw new DbException("Internal Page has no entry");
    for (int i = 0; i < moveNum; i++) {
        BTreeEntry entry = iterator.next();
        page.deleteKeyAndRightChild(entry);
        rightPage.insertEntry(entry);
    }
    BTreeEntry midEntry = iterator.next();
    page.deleteKeyAndRightChild(midEntry);

    midEntry = new BTreeEntry(midEntry.getKey(), page.getId(),
rightPage.getId());
    BTreeInternalPage parent = getParentWithEmptySlots(tid,
dirtypages, page.getParentId(), midEntry.getKey());
}

```

```

        parent.insertEntry(midEntry);

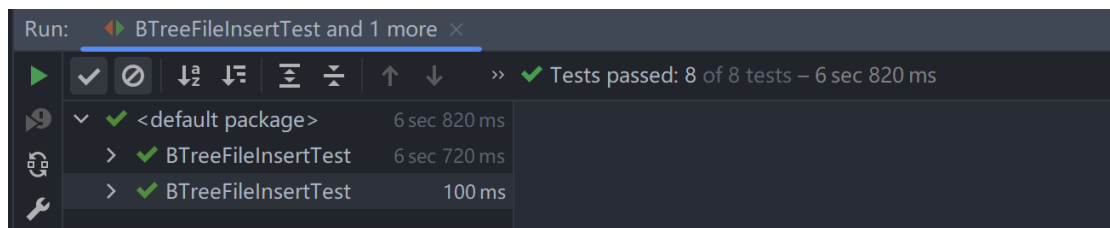
        updateParentPointers(tid, dirtyPages, rightPage);
        updateParentPointers(tid, dirtyPages, parent);

        dirtyPages.put(page.getId(), page);
        dirtyPages.put(rightPage.getId(), rightPage);
        dirtyPages.put(parent.getId(), parent);

        if (field.compare(Op.LESS_THAN, midEntry.getKey())) {
            return page;
        } else {
            return rightPage;
        }
        //return null;
    }
}

```

(4) 测试截图:



5.3. Exercise 3 Delete

(1) 实现 `BTreeFile.stealFromLeafPage()`, `BTreeFile.stealFromLeftInternalPage()`, `BTreeFile.stealFromRightInternalPage()` 并通过 `BTreeFileDeleteTest.java` 中的一些单元测试 (如 `testStealFromLeftLeafPage` 和 `testStealFromRightLeafPage`)

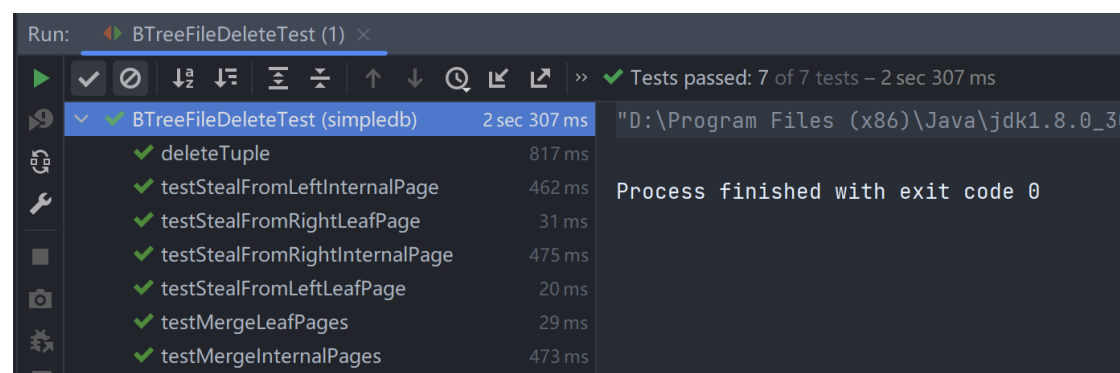
(2) 删除存在两种情况, 如果兄弟节点数据比较多可以从兄弟节点借, 反之数据较少可以和兄弟节点合并。

`stealFromLeafPage()` 两个页面 tuple 加一起然后除二, 平均分成两个 leaf page。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。在前面

插入的操作能够完成后，后面的套路会发现也是差不多的，然后讲义给的图示也是很容易懂的，可以帮我们理清思路。

(3) 照上述思路分别实现三个函数即可。

(4) 测试截图：



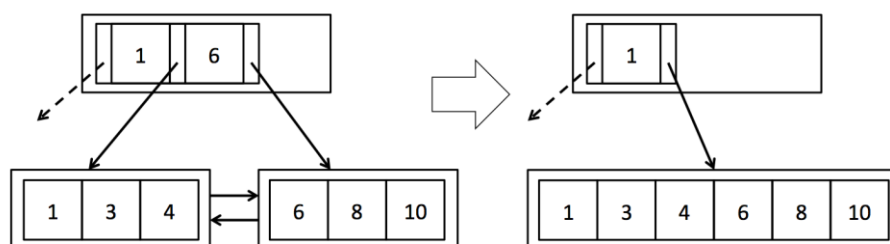
5.4. Exercise 4 Merging pages

(1) 实现 `BTreeFile.mergeLeafPages()` 和 `BTreeFile.mergeInternalPages()`。通过 `BTreeFileDeleteTest.java` 中的所有单元测试和 `systemtest/BTreeFileDeleteTest.java` 中的系统测试。

(2) merge page 要注意的同样是指针的更新，思路还是比较清晰的。

`MergeLeafPages` 步骤：

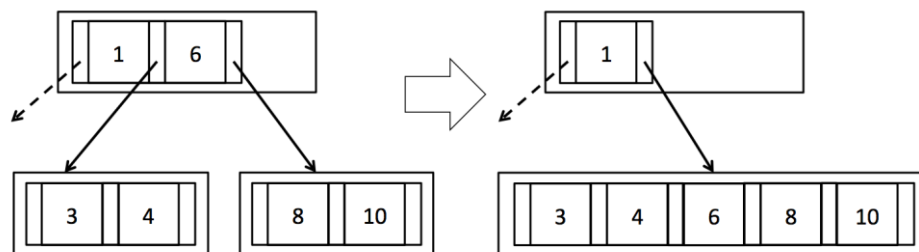
- 1、将 `rightPage` 中的所有 tuple 添加到 `leftPage` 中。
- 2、判断 `rightPage` 是否有右兄弟，如果没有 `leftPage` 的右兄弟为空，如果有 `leftPage` 的右兄弟指向 `rightPage` 的右兄弟。
- 3、调用 `setEmptyPage` 方法将 `rightPage` 在 header 标记为空。
- 4、调用 `deleteParentEntry` 方法，从父级中删除左右孩子指针指向 `leftPage` 和 `rightPage` 的 entry。
- 5、将 `leftPage` 与 `parent` 添加到 `dirtyPages` 中



Merging Leaf Pages

MergeInternalPages 步骤:

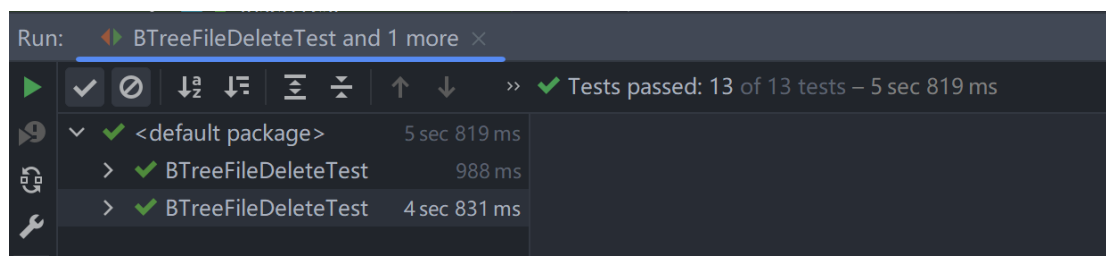
- 1、先将父节点中的指向 leftPage 和 rightPage 的 entry 添加到 leftPage 中
- 2、将 rightPage 中的 entry 添加到 leftPage 中
- 3、更新 leftPage 孩子节点的指针（将原本父节点指向 rightPage 的孩子节点的父节点更新为 leftPage）
- 4、调用 setEmptyPage 方法将 rightPage 在 header 标记为空。
- 5、调用 deleteParentEntry 方法，从父级中删除左右孩子指针指向 leftPage 和 rightPage 的 entry。
- 6、将 leftPage 与 parent 添加到 dirtyPages 中



Merging Internal Pages

(3) 根据上述思路和步骤实现两个函数即可

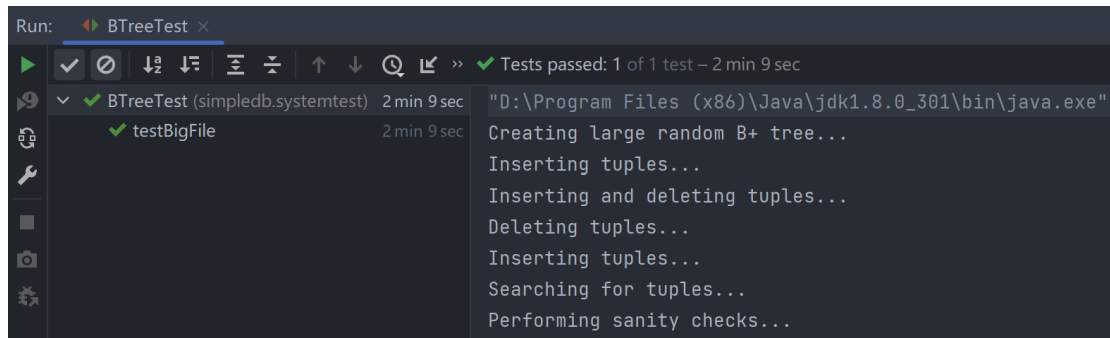
(4) 测试截图:



在 B+ 树代码中正确实现了锁定，应该能够通过 test/simpledb/BTreeDeadlockTest.java 的测试.



Btree 的系统测试也侥幸通过，但有点奇怪的是偶尔测试可能会出错。



The screenshot shows the 'Run' window of an IDE. At the top, it says 'Run: BTreeTest'. Below this is a toolbar with various icons. The main area displays the test results: 'BTreeTest (simplifiedb.systemtest)' with a green checkmark and a duration of '2 min 9 sec'. Underneath it, 'testBigFile' is also marked with a green checkmark and '2 min 9 sec'. To the right, the command used to run the test is shown: '"D:\Program Files (x86)\Java\jdk1.8.0_301\bin\java.exe"'. Below the command, the test's output is displayed: 'Creating large random B+ tree...', 'Inserting tuples...', 'Inserting and deleting tuples...', 'Deleting tuples...', 'Inserting tuples...', 'Searching for tuples...', and 'Performing sanity checks...'. The status bar at the top right indicates 'Tests passed: 1 of 1 test - 2 min 9 sec'.

```
Run: BTreeTest
✓ Tests passed: 1 of 1 test - 2 min 9 sec
✓ BTreeTest (simplifiedb.systemtest) 2 min 9 sec
  ✓ testBigFile 2 min 9 sec
    "D:\Program Files (x86)\Java\jdk1.8.0_301\bin\java.exe"
    Creating large random B+ tree...
    Inserting tuples...
    Inserting and deleting tuples...
    Deleting tuples...
    Inserting tuples...
    Searching for tuples...
    Performing sanity checks...
```

6. Lab 6 Recovery and Rollback

根据日志内容实现 rollback 和 recovery 。

当读取 page 时，代码会记住 page 中的原始内容作为 before-image 。当事务更新 page 时，修改后的 page 作为 after-image 。使用 before-image 在 aborts 进行 rollback 并在 recovery 期间撤销失败的事务。

6.1. Exercise 1 Rollback

(1) 实现 LogFile.java 中的 rollback()函数

通过 LogTest 系统测试的 TestAbort 和 TestAbortCommitInterleaved 子测试。

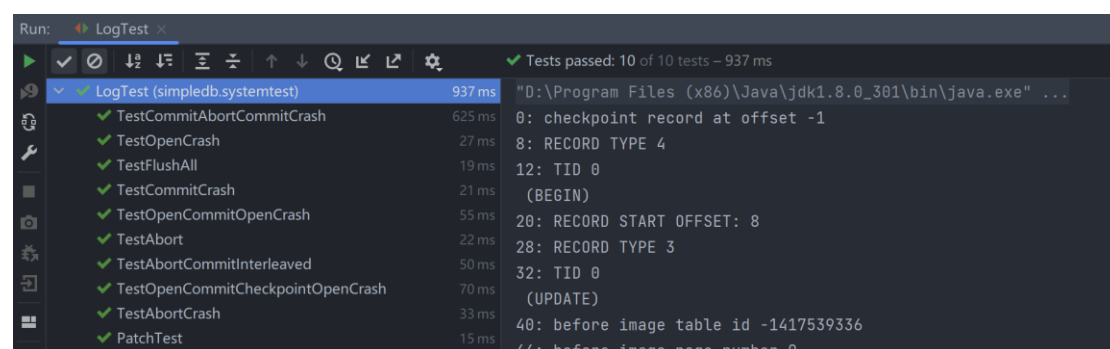
(2) rollback() 回滚指定事务，已经提交了的事务上不能执行该方法。将上一个版本的数据写回磁盘。

当一个事务中止时，在该事务释放其锁之前，这个函数被调用。它的工作是解除事务可能对数据库做出的任何改变。写这个 exercise 的时候要注意之前 buffer pool 也要进行相应的修改，一部分是刷入磁盘前需要记录；

(3) rollback 是 undo log 做的事，即提供上一个版本的快照（相比 MVCC 真是微不足道），在回滚时将上一个版本的数据写回磁盘，思路比较简单：

- 1.根据 tidToFirstLogRecord 获取该事务第一条记录的位置；
- 2.移动到日志开始的地方；
- 3.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。

(4) 测试截图：



6.2. Exercise 2 Recovery

(1) 实现 `LogFile.recover()`.

通过 `LogFile` 的所有测试.

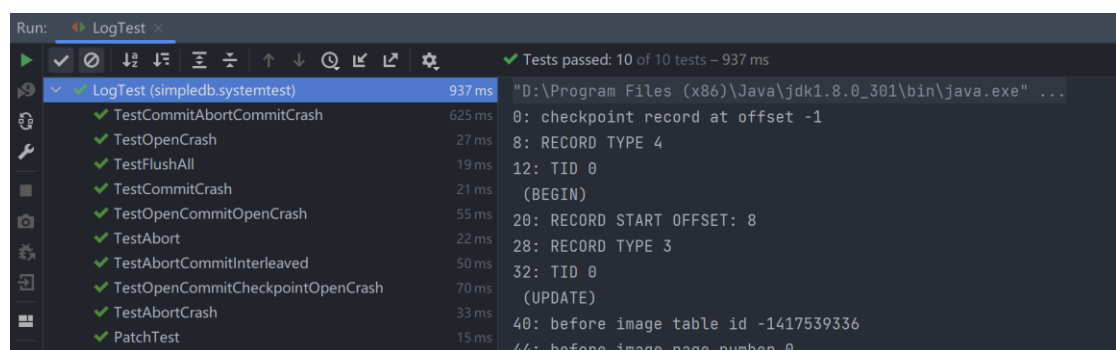
(2) 重启数据库时会率先调用 `LogFile.recover()`

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。

(3) 按照上述思路实现 `LogFile.recover()`即可

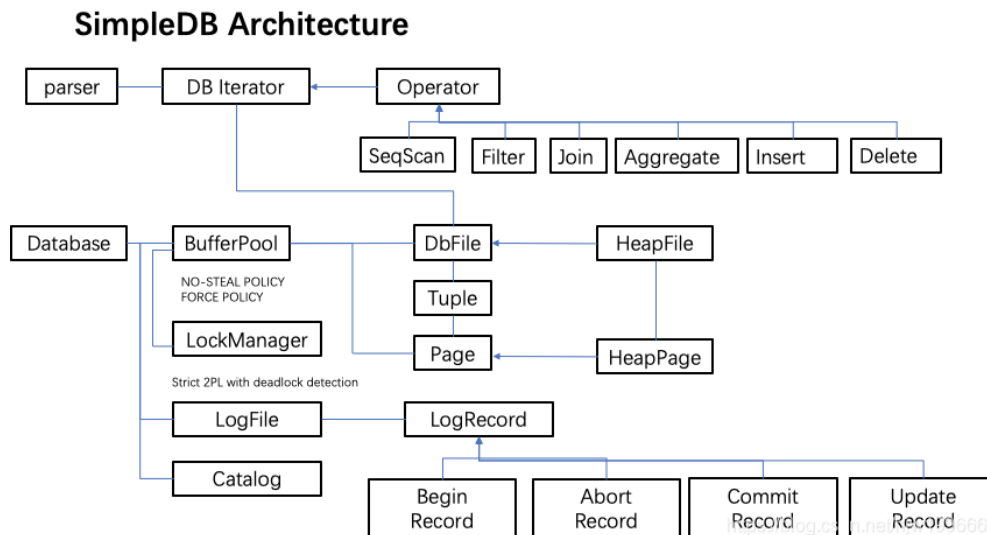
(4) 测试截图：同上



7. 总结

7.1. SimpleDB 总体架构

项目结构如下：



- lab1 实现基本的数据结构

tuple, page, tupleDesc, iterator 等等，难度不大

- lab2 实现 scan iterator

基于 scan iterator 来实现各种聚合函数，比如 avg, count, sum, join 等

- lab3 join 优化

建立一个优化模型，按照主键，非主键，scan 表代价，直方图等进行成本估计，根据估计值来确定多表 join 的顺序

- lab 4 事务以及锁

这一章相对较难，要自己实现一个简单的读写锁，但是 6.830 中简化了，实现了 page-level 的锁，粒度比较粗，还有多种死锁的情况，test 很给力，建议在写的时候一定要看清楚是哪个 transaction 拿到了哪些 page 的哪些 lock，而且这里的代码会影响到后面的 lab 5、6，这里主要是按照两阶段锁协议并且 no steal / force 的策略

- lab 5 B+ 树索引

实现 B+树索引，插入、删除、修改，难点在于要把 B+树结构以及这三种操作逻辑要捋清楚，还有父节点，子节点；叶子兄弟节点，非叶子节点的指针

问题，以及一些边界条件。

·lab 6 实现基于 log 的 rollback 和 recover

lab 中并没有真正存在 undo log 和 redo log，日志结构比较简单，只需要根据偏移处理即可，可以理解成是逻辑上的 undo log 和 redo log。

7.2. 实验收获

做完 6.830 这门实验课，实现了一个简单的数据库系统，收获还是很多的。虽然整个时间跨度比较大（半个学期 8 周吧），而且中间遇到了很多的困难，但当所有的测试通过的时候，还是感到很开心的。（即便仍有几处地方存在疑惑，不能尽善尽美）收获如下：

- （1）对数据库方面有了清晰的认识与理解，对其细节实现非常有利于今后数据库方面的应用，因为对数据库底层有了解之后对于数据库如何高效使用也有了一点方向，对以后从事数据库方面的发展作用很大。
- （2）编程能力有了较大提升，尤其 Java 语言的编程。之前并没有接触过比较大型的、代码量较大的 Java 项目，在做了这个实验之后，对 Java 的掌握有了很大提高，更加熟悉 Java 语言的各种特性。
- （3）系统设计方面，对于理解大型系统有了很好的训练，虽然这个 simpleDB 很小，但麻雀虽小，五脏俱全，做这个实验的过程终于对于系统整体结构有了很好的理解，对于以后再接触其他系统也会有很大的帮助。
- （4）实践能力的提升。数据库理论课程仅仅学会了数据库的使用知识，操作系统等其他课程也大多只涉及理论，而对于实验方面缺乏很大的训练。而这个实验极大地锻炼了动手实践的能力。页面置换策略在操作系统、组成原理等课程都学过相关地理论介绍，然后都没有亲自实现过置换策略，在这个实验中，我一开始也只使用了很直接简单的随机置换策略，在后面发现可以优化后，亲自实现了 LRU 页面置换策略，实现成功通过测试的时候还是很喜悦的。

7.3. 实验不足

虽然说顺利地做完了这个实验，收获也是很多，但在做的过程中还是有很

多不足和可以继续改进的地方，总结的几点如下：

- (1) 提前对整体结构有所了解。在刚开始做实验时，对于数据库的整体结构还很不清晰，直接就上手敲代码有点无从下手地感觉。后面还是多了即便实验的概述和指导，才开始写起来。所以，在上手之前，对整个实验的整体结构和大致内容有点了解是非常必要的。
- (2) 尽早定好实现策略。在页面置换策略这块，在 lab2 和 lab3 的时候就可以尝试实现 LRU，但我一直拖着，之后最后才下定决心修改 LRU 的页面置换策略，因此导致的是对之前代码较多的修改和重复的无用工作。
- (3) 做好版本管理。在初期并没有使用 git，只在本地进行保存，这样发现还是不太方便，而且没有修改记录等，对每个实验更改的文件显得有点杂乱，在调试和修改的时候受症颇多。
- (4) 对于测试的结果分析不够。实验中每个测试单独运行时都可以通过，然而整体全部运行时却会导致有些测试未能通过，对此十分疑惑。

7.4. 建议

(1) 课程内容很充实，整体难度还是很大的，因此完整做完这个实验后收获会非常大。而且该实验的指导、测试文档等方面非常完善，学生做起来对于系统的理解很有帮助，同时又留下许多可以自由发挥的地方供学生优化更改。

(2) 具体建议的话，希望可以增加课时和部分理论课的内容。由于之前所学的数据库系统课程只是关于如何使用和应用数据库的，并没有涉及关于数据库系统底层的知识，且该实验内容有点过大，但实验课时本身较短，而 MIT 的实验本身是和理论课结合的，我们只根据实验文档做实验的话对于某些地方的实现一开始并不是很清楚。因此希望可以延长一些课时，增加一些理论讲解的内容。

8. 参考资料

- [1] <https://blog.csdn.net/hjw199666/article/details/103824797>
- [2] https://blog.csdn.net/weixin_43414605/article/details/123517544
- [3] <https://zhuanlan.zhihu.com/p/158590975>
- [4] <https://www.cnblogs.com/cpaulyz/p/14606606.html>