



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

数据库系统实验报告

SimpleDB Lab2

姓名：韩佳迅

学号：2012682

年级：2020 级

专业：计算机科学与技术

2022 年 4 月 17 日

摘要

本次实验完成了 SimpleDB 的 Lab2, 并通过了测试。通过完成 Lab2 实现了数据库的运算符, 重点在于理解运算符实现方法与文件之间的调用关系。

本报告中给出了每个类的具体设计和实现思路、重难点, 借此实验和报告理清了数据库运算符关系。

目录

一、 Lab2 总览	1
(一) Filter and Join	1
(二) Aggregates	1
(三) HeapFile Mutability	1
(四) Insertion and deletion	1
(五) Page eviction	1
二、 设计思路及重难点	1
(一) Exercise 1	1
1. 类属性及方法	1
2. 思路及重难点	4
(二) Exercise 2	4
1. 类属性及方法	4
2. 思路及重难点	7
(三) Exercise 3	8
1. 类属性及方法	8
2. 思路及重难点	9
(四) Exercise 4	10
1. 类属性及方法	10
2. 思路及重难点	11
(五) Exercise 5	11
1. 类属性及方法	11
2. 思路及重难点	11
三、 Git Commit History	12
四、 改动部分	12

一、 Lab2 总览

GitLab 仓库地址: <https://gitlab.com/hanmaxmax/SimpleDB>

Lab2 主要是实现数据库的运算符, 来实现表的修改。分为以下几个部分:

(一) Filter and Join

Predicate 将元组与指定的字段值进行比较, 目的是实现条件过滤。

JoinPredicate 利用一个 predicate 对两个 tuples 的 fields 进行比较, 助于连接操作的过滤。

Join 实现关系连接的操作符, 并实现类似迭代器的功能, 可以获得对连接后的数据的迭代。

Filter 实现关系选择的操作符, 利用 Predicate 起到条件过滤的作用, 并实现类似迭代器的功能, 可以获得对过滤后的数据的迭代。

(二) Aggregates

Aggregate 实现聚合操作, 如 count、sum、avg、min、max 等, 并且支持分组 grouping。

IntegerAggregator 和 StringAggregator 分别实现对 int 和 string 类型的聚合。

(三) HeapFile Mutability

实现 HeapPage、HeapFile、BufferPool 的插入元组和删除元组的功能。

(四) Insertion and deletion

Insert, 插入运算符, 通过调用 HeapPage、HeapFile、BufferPool 中插入元组的方法实现插入功能。

Delete, 删除运算符, 通过调用 HeapPage、HeapFile、BufferPool 中删除元组的方法实现删除功能。

(五) Page eviction

在 BufferPool 中实现页面淘汰策略, 因为 BufferPool 缓冲池的最大 page 数有限, 当所写页面超过最大容量时, 要把暂时不用的页面从中淘汰出去。

二、 设计思路及重难点

(一) Exercise 1

1. 类属性及方法

Predicate.java

类的成员属性:

- int field;

进行比较的 field num (field 在 tuple 中的 index)

- Op op;
用于比较的 Operation 操作符
- Field operand;
传入的要比较的 field 值

类的成员方法:

- Predicate(int field, Op op, Field operand) 构造函数
将类属性赋值。
- getField()、getOp()、getOperand()
返回 Predicate 类中的属性值。
- filter(Tuple t)
使用 Field.java 的 compare 函数, 实现对参数元组 t 的 (下标为 field 的) field 值与类属性——operand 值在 op 操作符进行比较, 并返回比较结果的 bool 值。
- toString()
Returns something useful, like “f = field_id op = op_string operand =operand_string”。

JoinPredicate.java:**类的成员属性:**

- int field1; int field2;
用于比较连接的两个 tuple 的 field 下标值
- Predicate.Op op;
用于比较的 Operation 操作符

类的成员方法:

- JoinPredicate(int field1, Predicate.Op op, int field2) 构造函数
对类属性赋值。
- filter(Tuple t1, Tuple t2)
使用 Field.java 的 compare 函数, 实现对参数元组 t1 的 (下标为 field1 的) field 值与参数元组 t2 的 (下标为 field2 的) field 值在 op 操作符下进行比较。
- getField1()、getField2()、getOperator()
返回类属性。

Filter.java

类的成员属性:

- Predicate p;
用于过滤的断言 p。
- OpIterator child;
要从中读取要过滤的元组的子迭代器。

类的成员方法:

- Filter(Predicate p, OpIterator child) 构造函数
将类属性赋值。
- getPredicate()、getTupleDesc()
返回用来过滤元组的断言 Predicate p、返回要过滤的元组的 tupleDesc (即 child 的 tupleDesc)。
- open()、close()、rewind()
分别对 child 迭代器和基类 super 迭代器 open、close、rewind。
- fetchNext()
从子迭代器 child 遍历元组, 将断言 Predicate p 应用于它们, 并返回 child 迭代的下一条满足断言过滤关系的元组 (predicate .filter() 返回 true 的元组)。
- getChildren()、setChildren(OpIterator[] children)
获取/重设子迭代器 child。

Join.java

类的成员属性:

- JoinPredicate p;
用于连接过滤的断言 p。
- OpIterator child1; OpIterator child2;
由于是连接过滤, 所以有两个要从中读取要过滤的元组的子迭代器。
- Tuple t;
用于 fetchNext 的 tuple, 来存当前迭代到了哪个 tuple

类的成员方法:

- Join(JoinPredicate p, OpIterator child1, OpIterator child2) 构造函数
将类属性赋值。
- getJoinPredicate()、getJoinField1Name()、getJoinField2Name()、getTupleDesc()
返回连接过滤的断言 JoinPredicate;
用于连接的元组 1 的 field 的 name、用于合并的元组 2 的 field 的 name (这两个分别通过 child1 和 2 的 tupleDesc 和 JoinPredicate 里存的 field 下标获得);
连接后的 tupleDesc (由 child1 和 2 的 tupleDesc merge 获得)
- open()、close()、rewind()
分别对 child1、child2 迭代器和基类 super 迭代器 open、close、rewind。
- fetchNext()
先对 child1 迭代作为外层循环, 然后对 child2 迭代进入内层循环, 寻找 child2 里的元组能和 child1 的元组满足 JoinPredicate 的 filter 关系; 然后将这两个元组合并, 返回合并后的新元组。
- getChildren()、setChildren(OpIterator[] children)
获取/重设子迭代器 child1 和 child2。

2. 思路及重难点

Exercise 1 完成了过滤和连接过滤的代码。

过滤 (Predicate 和 Filter) 的思路: 先要实现 Predicate.java 断言, 在其中对 tuple 里的某个 field 值与预先设好的 field 值在几种操作符下作比较。然后在 Filter.java 里面使用 Predicate 断言, 对传入的 child 元组迭代器依次迭代其中的元组, 对其判断断言条件, 过滤出满足条件的元组。(并且 Filter.java 实现了迭代器的功能, 可以对过滤后的元组迭代)。

连接过滤 (JoinPredicate 和 Join) 的思路: 连接过滤与过滤类似。先要实现 JoinPredicate.java 断言, 在其中对传入的两个 tuple 里的某个 field 值在几种操作符下作比较。然后在 Join.java 里面使用 JoinPredicate 断言, 对传入的两个 child 元组迭代器依次迭代其中的元组, 对其判断连接断言条件, 过滤出满足连接条件的元组对, 然后将其合并, 返回合并后的元组。(Join.java 也实现了迭代器的功能, 可以对过滤且合并后的新元组迭代)。

(具体思路见上面【类属性及方法】小节)

重难点:

难点在于 Join 的 fetchNext, 要注意先在类属性里添加元组 tuple t, 用于存 child1 迭代到了哪里, 从而实现迭代器 next 的功能。

并且在 fetchNext 里还要注意嵌套循环, 外层对 child1 迭代, 内层对 child2 迭代, 当内层的 child2 迭代完而外层 child1 还未迭代完时, 要注意把 child2 进行 rewind。

其次, 要实现对满足连接过滤条件的元组的合并, 并返回连接后的新元组。注意实现 tupleDesc 的 merge, 以及对新元组 field 和 tupleDesc 的 set。

(二) Exercise 2

1. 类属性及方法

StringAggregator.java

类的成员属性:

- `int gbfield;`
该索引值指定了要使用 tuple 的哪一个列来分组。
- `Type gbfieldtype;`
指定了作为分组依据的那一列的值的类型。
- `int afield;`
该索引指定了要使用 tuple 的哪一个列来聚合
- `Op what;`
聚合操作符
- `Map<Field, Integer> groupMap;`
存放聚合操作的映射关系（从 Field 到 Integer）。

类的成员方法:

- `StringAggregator` 构造函数
对类属性赋值,并且 new 哈希表。注意在构造函数中要检查 Op what 操作符只能是 COUNT, 否则抛出异常。
- `mergeTupleIntoGroup(Tuple tup)`
将新元组 tup 合并到聚合中, 并按照构造函数中的指示进行分组。注意: `StringAggregator` 只需要实现 COUNT 聚合操作。
- `iterator()`
利用 `OpIterator` 的接口写一个 `class AggregateIterator` 迭代器, 在类里重新实现迭代器的功能。

IntegerAggregator.java**类的成员属性:**

- `int gbfield;`
该索引值指定了要使用 tuple 的哪一个列来分组。
- `Type gbfieldtype;`
指定了作为分组依据的那一列的值的类型。
- `int afield;`
该索引指定了要使用 tuple 的哪一个列来聚合

- Op what;
聚合操作符
- Map<Field, Integer> groupMap;
存放聚合操作的映射关系（从 Field 到 Integer）。
- Map<Field, List<Integer>» avgMap;
存放聚合操作的映射关系（从 Field 到 List）——> 用于做 avg 的聚合（因为做 avg 运算时，要做除法，所以先存到 list 里，等聚合完之后再算）。

类的成员方法:

- IntegerAggregator 构造函数
对类属性赋值，并且 new 哈希表。
- mergeTupleIntoGroup(Tuple tup)
将新元组 tup 合并到聚合中，并按照构造函数中的指示进行分组。在 IntegerAggregate 里需要分别实现 COUNT , SUM , AVG , MIN , MAX 这几组操作符。
- iterator()
写一个迭代器类，继承自 StringAggregator 里的 AggregateIterator，因为 StringAggregate 只需要实现 COUNT，也就是从 Field 到 Integer 的映射，所以在 IntegerAggregate 中讨论完 int 里面特有的运算符之后，剩下的就返回它的基类 StringAggregator 中所定义过的运算符的计算方法。

Aggregate.java

类的成员属性:

- OpIterator child;
OpIterator 类型的 child 迭代器，用于不断迭代 tuples。
- int gbfield;
该索引值指定了要使用 tuple 的哪一个列来分组。
- int afield;
该索引指定了要使用 tuple 的哪一个列来聚合
- Aggregator.Op aop;
我们需要使用的 Aggregation operator 聚合操作符
- Aggregator aggregator;
在 Aggregate 类里面 new 一个 StringAggregator 或 IntegerAggregator 作为 aggregator

- OpIterator it;

上面定义的 aggregator 的迭代器

- TupleDesc td;

若没有分组, 则 td 为 (the aggregate column) 若有分组, 则 td 应为 (the group by field, the aggregate value column)

类的成员方法:

- Aggregator 构造函数

对类属性赋值, 并且根据 child 子迭代器中元组的需要聚合列的类型来 new 一个 StringAggregator 或 IntegerAggregator 进行聚合操作; 并且在构造函数中生成 tupleDesc (聚合列的类型, 聚合列的名称)。

- groupField(), groupFieldName(), aggregateField(), aggregateFieldName(), aggregateOp(), getTupleDesc()

返回分组列的 field 下标, 若没有分组则返回 NO_GROUPING;

返回分组列的 field name, 若无分组则返回 null;

返回聚合列的 field 下标;

返回聚合列的 field name;

返回聚合操作符;

返回 tupleDesc, 其中的 type 是分组列类型 (若有) + 聚合列类型, name 是分组列名称 (若有) + 聚合列名称

- open(), rewind(), close()

分别对 child1、child2 迭代器和基类 super 迭代器 open、close、rewind。

- fetchNext()

由于在构造函数中已经 new 了 StringAggregator 或 IntegerAggregator, 所以在 fetchNext 中直接使用它们的迭代器迭代即可。

- getChildren(), setChildren(OpIterator[] children)

获取/重设子迭代器 child, 注意当重设 child 时, 还要重新更新 tupleDesc。

2. 思路及重难点

Exercise 2 完成了数据库聚合、分组操作的代码。

聚合/分组的思路: 共编写了三个文件来实现这个功能, Aggregate 实现 Operator 接口, 是聚合操作类。IntegerAggregator 和 StringAggregator 是 Aggregate 的辅助类, 它们的区别在于聚合操作字段的属性, 如果是 Integer, 可以有 count、sum、max、min、avg 操作; 如果是 String, 只有 count 操作。最后在 Aggregate 中将它们整合在一起即可。调用过程如下:

(1) 从需求中提取分组字段、聚合字段、聚合操作符; 从数据库中获取要操作的表, 并获得其迭代器作为子迭代器 child (数据源)。

(2) 根据分组字段、聚合字段、聚合操作符、child 创建 Aggregate, Aggregate 根据字段们的具体类型构造聚合器 IntegerAggregator 或 StringAggregator, 以及聚合结果元组的描述信息 td。

(3) Aggregate 相当于对聚合结果的迭代器, 调用 child 子迭代器的 open、next 等方法, 不断从数据源 child 取出记录, 并调用聚合器的 mergeTupleIntoGroup 进行聚合运算; 运算结束后通过聚合器的 iterator 方法生成结果迭代器 it。

(4) 不断从结果迭代器 it 取出结果。

(具体思路见上面【类属性及方法】小节)

重难点:

1. 重点一在于两个 mergeTupleIntoGroup 函数的编写, 在 StringAggregator 中, 只需要实现 COUNT 聚合, 在 IntegerAggregator 中需要实现多个聚合的操作, 其中要注意 avg 聚合时, 要写一个 <Field,List> 的哈希映射, 因为 avg 运算时, 要做除法, 所以先存到 list 里, 等聚合完之后再算。

2. 重点二在于两个 Aggregator 的迭代, 要自己写一个迭代器, 在其中主要实现 next 操作, 返回的 tuple 是以 (field, 字段下标) 格式; 但在 IntegerAggregator 类型里, 要注意其他聚合操作符的讨论 (与 COUNT 不同)。因此, 在这里的设计中, 先在 StringAggregator 里实现一个接口, 然后再在 IntegerAggregator 里继承它, 完善剩下几个特殊聚合操作符的迭代实现。最后在 Aggregate 里 new 两个 Aggregator, 调用它们的迭代器。

3. 难点在于 mergeTupleIntoGroup 中要讨论异常情况, 比如参数给的类型与要判断的类型不相符。

(三) Exercise 3

1. 类属性及方法

HeapPage.java

类的成员方法:

- deleteTuple(Tuple t)

删除 page 里的某个 tuple, 并更新 header 里这个位置的使用信息。

- insertTuple(Tuple t)

往 page 里添加 tuple, 遍历 page 中的 slots, 寻找空位置, 并把新 tuple 加进去。

- markDirty(boolean dirty, TransactionId tid)、isDirty()

标记这页 page 为脏页, 且 dirtyId 为 tid。

若这页是脏页, 则返回它的 dirtyId, 否则返回 null。

- markSlotUsed(int i, boolean value)

标记下标为 i 的 slot 为 used, 其中要用到位运算, 先对 i 整除 8, 找到对应字节, 然后再对 8 取余, 找到对应位, 使用与或非操作更改位。

HeapFile.java

类的成员方法:

- writePage(Page page)

将某一页写进磁盘, 要先在 file 定位 page, 然后把其中的 data 拿出来写进内存。

- insertTuple(TransactionId tid, Tuple t)

遍历整张表 (HeapFile) 的所有数据页 (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;

若所有 page 都没有空位, 则在磁盘创建一个空的数据页, 再调用 HeapPage 的 insertTuple 方法进行插入。

最后把 BufferPool 中被插入元组后的页面保存到 list 中并返回, 表明这是脏页, 在以后的文件中会用到。

- deleteTuple(TransactionId tid, Tuple t)

找到 tuple t 对应的 HeapPage (用 BufferPool.getPage() 获取), 然后调用对应 page 的 deleteTuple 函数。

BufferPool.java**类的成员方法:**

- insertTuple(TransactionId tid, int tableId, Tuple t)

根据传入的参数 tableId 找到对应的 file 表, 在对应的 file 中调用其 insertTuple 函数, 得到该函数返回的被插入数据的 page 列表 pagelist, 遍历这个 page 列表, 标记脏页并加到 BufferPool 里 (若 cache 满了, 则调用 evictPage() 驱逐一些不用的页面)。

- deleteTuple(TransactionId tid, Tuple t)

思路同 Insert。

2. 思路及重难点

Exercise 3 实现了数据库 HeapPage、HeapFile、BufferPool 的插入元组和删除元组的方法。由于 Exercise 4 也是实现插入删除的操作, 所以整体的插入删除思路写到下一节 Exercise 4 里。

(具体思路见上面【类属性及方法】小节)

重难点:

1. 难点在于异常的考虑: HeapFile 的 writePage() 中要讨论写入磁盘失败的情况, 如 pageNum 超过 HeapFile 中最大 page 数。HeapPage 的 deleteTuple 要考虑比较 HeapPage 的 pageId 与 tuple 所在的 pageId 是否一致 (隐含问题) 等。

2. 重点在于理解 HeapPage、HeapFile、BufferPool 之间的调用关系。

(四) Exercise 4

1. 类属性及方法

Insert.java

类的成员属性:

- TransactionId transactionId; OpIterator child; int tableId;
transaction 的 tid;
用于数据源的子迭代器 child;
要插入的表的 id。
- TupleDesc td; int count; boolean inserted;
存插入数据信息的结构;
用于迭代计数的 count;
用于标记插入操作是否开放 (锁)。

类的成员方法:

- Insert 的构造函数
为类属性赋初值, 同时设置 tupleDesc 的结构, 以及将计数器 count 设为 0, 将插入标志记为 false。
- getTupleDesc()
返回在构造函数中定义好的 tupleDesc, 即记录插入数据信息的结构。
- fetchNext()
首先要检查是否开锁 (Inserted 是否为 true), 然后迭代数据源 child, 即要插入的 tuple 集, 调用 Database.getBufferPool().insertTuple(), 进行插入; 并使相应的计数器 count++, 返回插入的以 td 为结构的 tuple。
- open()、close()、rewind()
分别对 child 迭代器和基类 super 迭代器 open、close、rewind; 并重置 count 和 inserted
- getChildren()、setChildren(OpIterator[] children)
获取/重设子迭代器 child。

Delete.java

思路与 Insert 几乎一致, 只需将调用 BufferPool 的 insert 操作改为 delete。

2. 思路及重难点

Exercise 4 完成了数据库 Insert 和 Delete, 实际上就是插入、删除两个操作的迭代器。

数据库插入删除的过程:

(1) 根据需求获得要插入/删除的表、元组等信息。
(2) 根据表获取表的 id, 并将记录信息封装成数据源 child (实质是一个迭代器), 生成本次批量插入操作的事务 id。

(3) 利用上述生成的参数构造 Insert/Delete 对象。

(4) 调用 Insert 的 hasNext 方法, 判断是否有结果。hasNext 会调用我们写的 fetchNext 方法, 去执行插入操作并获取结果。

(5) 插入/删除的具体执行过程 (以插入为例, 删除与之类似):

调用 Database.getBufferPool().insertTuple(tid,tuple) 方法进行插入, BufferPool 的 insertTuple 会根据 tableId 从获取数据库文件 HeapFile, 并调用 HeapFile 的 insertTuple 方法;

而 HeapFile 的 insertTuple 方法会调用 BufferPool.getPage() 方法从缓冲池取出页面 HeapPage (如果缓冲池没有才会从磁盘中取并放入缓冲池);

获取 HeapPage 后, 调用 HeapPage.insertTuple() 方法, 去插入元组;

插入完成后, HeapFile 会返回从 BufferPool 中获取并插入了元组的页面, 在 BufferPool 的 insertTuple 中把它标记为脏页并写回缓冲池。

(6) 插入/删除操作全部完成后, 则得到一个结果元组。

(具体思路见上面【类属性及方法】小节)

重难点:

1. 理解 Insert、Delete 的迭代器功能, 重点在于实现 fetchNext 功能。
2. 理清 Insert/Delete、HeapPage、HeapFile、BufferPool 的调用关系。

(五) Exercise 5

1. 类属性及方法

BufferPool.java

类的成员方法:

- discardPage(PageId pid)
用于移除 BufferPool 中的索引里移除页。
- flushPage(PageId pid)
如果是脏页, 将脏页写入内存, 并取消脏页标记。
- evictPage()
实现一个页面淘汰策略。

2. 思路及重难点

Exercise 5 完成了数据库 BufferPool 中实现页面淘汰策略。

思路: BufferPool 缓冲的最大页数有限, 当我们写入的页面超过最大页数时, 需要将暂时不需要的页面从 BufferPool 中淘汰出去。

(具体思路见上面【类属性及方法】小节)
重难点在于对磁盘和 BufferPool 的读写。

三、 Git Commit History

GitLab 仓库地址: <https://gitlab.com/hanmaxmax/SimpleDB>








































16 Apr, 2022 2 commits			往 后
	finish BufferPool.java KasonHan authored 1 day ago	74eec6e5	 
	finish Insert.java and Delete.java KasonHan authored 1 day ago	fc9277be	 
15 Apr, 2022 5 commits			
	finish HeapFile.java and BufferPool.java, also correct HeapPage.java KasonHan authored 1 day ago	4ffa6381	 
	finish HeapPage.java KasonHan authored 1 day ago	41d1353b	 
	finish Aggregate.java KasonHan authored 2 days ago	cd882fb2	 
	finish StringAggregator.java and add something to IntegerAggregator.java KasonHan authored 2 days ago	f174809b	 
	finish StringAggregator.java KasonHan authored 2 days ago	47ed6aa4	 
14 Apr, 2022 1 commit			
	finish IntegerAggregator.java KasonHan authored 2 days ago	4fc33b2b	 
13 Apr, 2022 4 commits			
	finish Join.java KasonHan authored 3 days ago	e2cbb57d	 
	finish Filter.java KasonHan authored 3 days ago	f5f6b3eb	 
	finish JoinPredicate.java KasonHan authored 3 days ago	24907679	 
	finish Predicate.java KasonHan authored 3 days ago	44fcc150	 
09 Apr, 2022 2 commits			
	Merge remote-tracking branch 'origin/master' KasonHan authored 1 week ago	ffef2c3e	 

图 1: 提交记录

四、 改动部分

未改动 API。