



南開大學
Nankai University

计算机学院
数据库 SimpleDB 实验报告

Lab2

姓名：唐明昊

学号：2113927

专业：计算机科学与技术

2023 年 4 月 6 日

目录

1	Exercise 1	2
1.1	Predicate	2
1.2	JoinPredicate	2
1.3	Filter	2
1.4	Join	3
2	Exercise 2	4
2.1	IntegerAggregator	4
2.2	StringAggregator	6
2.3	Aggregate	6
3	Exercise 3	6
3.1	HeapPage	6
3.2	HeapFile	7
3.3	BufferPool	8
4	Exercise 4	9
4.1	Insert	9
4.2	Delete	10
5	Exercise 5	10
5.1	BufferPool	10
6	Git Commit History	11

1 Exercise 1

1.1 Predicate

Predicate 谓词类，被 Operator (Filter) 调用用来作比较，进而筛选 tuple。Predicate 里定义了枚举常量 Op，用于充当比较符。

Predicate 根据构造函数声明三个变量分别用来记录 tuple 中做比较的 field 的 index、比较符、要比较的 field 值。

filter 函数是专门被 Operator 调用的接口，使用 field 的 compare 函数，实现对特定的 tuple 比较筛选。

filter 函数

```

1 public boolean filter(Tuple t) {
2     // some code goes here
3     return t.getField(numOffField).compare(opForCom,
4         operand); // 用 t 去比较 operand，交换会出错？可能 number 不对？
5 }

```

1.2 JoinPredicate

JoinPredicate 类用来比较两个 tuple，由 Join 调用，用来筛选 tuple。

filter 函数同 Predicate，使用 field 的 compare 函数。

1.3 Filter

Filter 类是一个 Operator，直接继承自 Operator。

构造函数中接受一个 Predicate 对象和 OpIterator 对象。Filter 是上层迭代器，使用 Predicate 过滤 child 迭代器返回上来的 tuple。

fetchNext 函数返回一个 tuple，直接从 child 迭代器获取，做判断是否能够通过谓词过滤。

fetchNext 函数

```

1 protected Tuple fetchNext() throws NoSuchElementException,
2     TransactionAbortedException, DbException {
3     // some code goes here
4     // 用 child 去遍历，返回符合 predicate 的 tuple
5     while(child.hasNext()) {
6         Tuple temp=child.next();
7         if(p.filter(temp)) {
8             return temp;
9         }
10    }
11    return null;
12 }

```

1.4 Join

Join 类首先筛选元组，并将两个符合条件的元组组合到一起。

构造函数中会接受两个子迭代器，由两个子迭代器获取 tuple 返回到 Join 类来，进行 JoinPredicate 筛选，而后合并。

fetchNext 函数采用**循环嵌套**实现。child1 在外层循环，内层循环将 child2 不断迭代，**child1 每次得到一个 tuple 都会遍历 child2 中的所有元素**。

child1 需要谨慎移动，因为找到合适的两个 tuple 函数即会返回，如果下次再寻找时调用 child1 的 next 则会使得当前的 tuple 丢失，**没有让 child1 中的 tuple 遍历 child2 中的所有 tuple**。于是需要定义 current 保存上次 child1 返回的 tuple，只有当 current 为 null 时才会调用 child1 的 next。

child2 在内层不断迭代，当找到符合过滤条件的 tuple 后，构造一个新的 tuple 对象，把 child1 和 child2 返回的 tuple 中的字段融合到一起，返回。

当内层循环做完即 child2 里已经走到头，则需要将 child2 进行 rewind **重置**，同时让 current 为 null，使得在下一次外层循环时执行 child1 的 next。

fetchNext 函数

```

1  protected Tuple fetchNext() throws TransactionAbortedException, DbException {
2      // some code goes here
3      // gtJoin会报错?
4      // child1每次都在动，可能当前tuple正合适，
5      // 还可以与child2中进行连接，而next直接将他移过了
6      while(child1.hasNext() || current!=null) {
7          // 即使child1没有next，但current中保存有值
8          if(child1.hasNext()&&current==null){// 如果current没有值child1才执行next
9              current=child1.next();
10         }
11
12         // 循环嵌套，child1中一个tuple遍历child2中所有
13         while(child2.hasNext()) {
14             Tuple temp2=child2.next();
15             if(p.filter(current, temp2)) {// 二者合适
16                 // 构造一个新的tuple
17                 TupleDesc newTd=getTupleDesc();
18                 Tuple result=new Tuple(newTd);
19
20                 // 给Tuple的每个字段设置值
21                 result.setRecordId(current.getRecordId());
22                 for(int i=0;i<child1.getTupleDesc().numFields();i++)
23                     result.setField(i, current.getField(i));
24                 for(int j=0;j<child2.getTupleDesc().numFields();j++)
25                     result.setField(j+child1.getTupleDesc().numFields(), temp2.getField(j));
26
27                 if(!child2.hasNext())
28                     {//child2走到头刚好找到，得重置，否则下次直接不进入循环
29                         child2.rewind();
30                         current=null;
31                     }

```

```

31         return result;
32     }
33 }
34 // 重置
35 child2.rewind();
36 current=null;
37 }
38 return null;
39 }

```

2 Exercise 2

2.1 IntegerAggregator

IntegerAggregator 对 int 类型的 field 进行聚合操作，构造函数传入分组字段，聚合字段，以及比较符。

为了实现保存每个组的聚合值，使用 **HashMap 数据结构**，将 gbfield 做为 key，聚合值作为 value。这里不使用 gbfield 的 value 做 key，因为在 mergeTupleIntoGroup 函数中返回时返回的还是字段，如果直接用 field 做 key 可以省去繁琐的取值和包装。

mergeTupleIntoGroup 函数接受一个 tuple，根据 gbfield 字段找到它所属的组，用它的 afield 字段去计算并更新聚合值。

特别的，对于求平均值操作，需要使用一个列表保存组里每一个元素的值，每次更新该组聚合值时将列表元素求和求平均。如果为了图方便只保存当前组有多少个元素，用平均值乘个数得总数再求平均进行计算，会因为四舍五入而误差逐渐放大。保存该组的列表同样使用 HashMap 数据结构，用 gbfield 做 key，列表做 value。

mergeTupleIntoGroup 函数

```

1  public void mergeTupleIntoGroup(Tuple tup) {
2      // some code goes here
3      // 对新加的一个tuple进行聚合操作，并进行分组
4      // 先得到gbField和aField两个字段
5      Field gb;
6      //int gbValue=-1;
7      if (this.gbField==Aggregator.NO_GROUPING)
8          gb=null;
9      else {
10         gb=tup.getField(this.gbField);
11         if (gbFieldType!=gb.getType())
12             throw new IllegalArgumentException("wrong type!");
13     }
14
15     IntField aField=(IntField) tup.getField(this.aField);
16     int aValue=aField.getValue();
17
18     // 对新加的这一行先判断是哪一组，再做aggregate
19     switch (this.operator) {

```

```

20     case MIN:
21         if (!group.containsKey gb) // 该组还没有东西
22             group.put gb, aValue;
23         else // 组里有东西, 执行一次 aggregate
24             group.put gb, Math.min(aValue, group.get gb));
25         break;
26     case MAX:
27         if (!group.containsKey gb)
28             group.put gb, aValue;
29         else
30             group.put gb, Math.max(aValue, group.get gb));
31         break;
32     case SUM:
33         if (!group.containsKey gb)
34             group.put gb, aValue;
35         else
36             group.put gb, aValue+group.get gb);
37         break;
38     case COUNT:
39         if (!group.containsKey gb)
40             group.put gb, 1; // 第一次添加
41         else
42             group.put gb, 1+group.get gb); // 加1
43         break;
44     case AVG:
45         if (!group.containsKey gb) {
46             group.put gb, aValue;
47             // 新建一个数组
48             ArrayList<Integer> newGroup = new ArrayList();
49             newGroup.add(aValue);
50             avgCount.put gb, newGroup; // 用来记录该组的值
51         }
52         else
53         {
54             ArrayList <Integer> get=avgCount.get gb);
55             get.add(aValue); // 加入新的一行
56             avgCount.put gb, get);
57
58             int sum = get.stream().reduce(Integer::sum).orElse(0); // 求和
59             int avg=sum/get.size(); // 求平均值
60             group.put gb, avg);
61         }
62         break;
63     default:
64         throw new IllegalArgumentException("Aggregate wrong!");
65     }
66 }

```

为了实现 iterator 函数, 使用**内部类**。定义 IntAggIterator, 实现 OpIterator 接口。类内定义下

层 iterator 在 group 的 hashmap 里迭代: `private Iterator<HashMap.Entry<Field,Integer> it;`
`next` 函数直接调用 `it` 的 `next`, 再根据是否有分组, 将一组的 key 和 value (即 `gbfield` 和 `afield`) 封装为一个 tuple 返回。

next 函数

```

1  public Tuple next() throws DbException, TransactionAbortedException,
    NoSuchElementException {
2      // TODO Auto-generated method stub
3      Map.Entry<Field, Integer> temp=it.next();
4      Tuple ret=new Tuple(td);
5      //做复杂了, 又把gbField包装回了field :(
6      if (gbField==Aggregator.NO_GROUPING) // (aggregateVal)
7          ret.setField(0, new IntField(temp.getValue()));
8      else { // (groupVal, aggregateVal)
9          ret.setField(0, temp.getKey());
10         ret.setField(1, new IntField(temp.getValue()));
11     }
12     return ret;
13 }

```

2.2 StringAggregator

StringAggregator 对 String 类型的字段做聚合操作, 并且只支持 count 操作, 于是 `mergeTupleIntoGroup` 函数只需要在每次输入一个 tuple 时, 将其所属的组的聚合值加一即可。迭代器的实现与 IntAggregator 一样, 实现内部类即可。

2.3 Aggregate

Aggregate 实现聚合操作。根据 `afield` 字段的 type 构造相应的底层 Aggregator, 由于底层 Aggregator 都做了很好的实现, 故只需要声明一个 `OpIterator` 变量让它直接等于 Aggregator 的 iterator, 再用其进行迭代即可。

在 `open` 函数里, 将 child 迭代器 open 后, 将 child 中的 tuple 全部放入 aggregator 里, 即循环调用 **mergeTupleIntoGroup** 函数, 就完成了 open 操作。其余的函数操作只需要对 child 和 it 两个迭代器进行调用即可。

3 Exercise 3

3.1 HeapPage

首先实现 `markSlotUsed` 函数, 该函数用来对 header 数组进行修改, 更改页中 slots 的使用情况。与 `isSlotUsed` 实现思想一样, 首先找到在第几个 byte, 然后找到位于第几个 bit。如果要将此位置 1 则让它与 1 相或, 置 0 则与 0 相与。

markSlotUsed 函数

```

1  private void markSlotUsed(int i, boolean value) {

```

```

2 //首先要确定在第几个byte
3 int positionOfByte=i/8;
4 //再确定在第几个bit
5 int positionOfBit=i%8;
6 if(value) {// 为真, 已使用, 置1
7     header[positionOfByte]=(byte)(header[positionOfByte]|(0x1<<positionOfBit));
8 }
9 else {//置0
10    header[positionOfByte]=(byte)(header[positionOfByte]&~(0x1<<positionOfBit));
11 }
12 }

```

insertTuple 在检查 tuple 合法性后, 遍历 header 数组, 如果 slot 没有被使用, 则将 tuple 放入此位置并做标记。需要注意的是, 此时需要对 tuple 设置 **recordId**, 记录他的具体位置, 用于后续删除操作。

deleteTuple 需要进行严格的合法性检查, 需要确保当前位置 slot 不为空且等于要删除的 tuple, 然后将数组中该位置设 null, 并标记为未使用。

3.2 HeapFile

insertTuple 需要用 bufferPool 的 **getPage** 函数去遍历该 file 中所有的 page, 如果该 page 还有空位 (即 getNumEmptySlots 不为 0), 则调用 page 的 insertTuple 函数。若该表中所有页都被占满, 则需要在 file 后面**追加新的一页**, 然后再用 bufferPool 去获取该页做插入。

insertTuple 函数

```

1 public ArrayList<Page> insertTuple(TransactionId tid, Tuple t)
2     throws DbException, IOException, TransactionAbortedException {
3     // some code goes here
4     ArrayList<Page> ret=new ArrayList<Page>();
5     // 查看file里面所有的page,寻找一个合适的插入tuple
6     for(int i=0;i<numPages();i++) {
7         // 用bufferPool获取page
8         // we should be able to add 504 tuples on an empty page. 报错↓
9         //HeapPageId pid=(HeapPageId) t.getRecordId().getPageId();//
            tuple插入后才有recordId! recordId就是用来记录这个的!
10        HeapPageId pid =new HeapPageId(this.getId(),i);
11        HeapPage page=(HeapPage) Database.getBufferPool().getPage(tid, pid,
            Permissions.READ_WRITE);
12        if(page.getNumEmptySlots()!=0){// 该页有空位可以插入
13            page.insertTuple(t);
14            ret.add(page);
15            return ret;// 也别break了, 直接return吧
16        }
17    }
18    // 报错->the next 512 additions should live on a new page
19    // 在file里追加一页
20    RandomAccessFile raf=new RandomAccessFile(table,"rw");
21    int offset=numPages()*BufferPool.getPageSize();// 从尾部追加

```



```

22 raf.seek(offset);
23 byte[] emptyPageData=HeapPage.createEmptyPageData();
24 raf.write(emptyPageData);
25 raf.close();
26 // 拿出新的一页做插入
27 HeapPageId pid=new HeapPageId(this.getId(),numPages()-1);
28 HeapPage page=(HeapPage) Database.getBufferPool().getPage(tid, pid,
    Permissions.READ_WRITE);
29 page.insertTuple(t);
30 ret.add(page);
31
32 return ret;
33 }

```

deleteTuple 函数相对容易，根据要删除的 tuple 的 recordId 找到它所在的页，在调用 page 的 deleteTuple 函数即可。

3.3 BufferPool

insertTuple 函数由 tableId 定位到对应的 DbFile，再调用 file 的 insertTuple 函数插入 tuple。插入后要将修改过的 page（file 返回的 page 数组）**标记为脏**，而后把 page 放入缓存。

insertTuple 函数

```

1 public void insertTuple(TransactionId tid, int tableId, Tuple t)
2     throws DbException, IOException, TransactionAbortedException {
3     // some code goes here
4     // 先获取HeapFile
5     DbFile f=Database.getCatalog().getDatabaseFile(tableId);
6     // 插入tuple
7     ArrayList<Page> p=f.insertTuple(tid, t);
8     // makeDirty
9     for(Page page:p) {
10         page.markDirty(true, tid);
11         // 写报告时经提醒修改，插入删除要将它放入cache，需要判断空间
12         // 那有意义吗？f里面调用getPage不是已经将他放进缓存了吗
13         // 有可能并发处理？虽然它在缓存中，但过程中可能缓存又放入了page？
14         if(pages.size()>=numPages) {// insufficient space
15             evictPage();
16         }
17         pages.put(page.getId().hashCode(), page);// update
18
19         pageOrder.remove(page.getId());// 最近进行了调用，LRU原则对他进行更新
20         pageOrder.add(page.getId());
21     }
22 }

```

deleteTuple 函数与插入一样，定位到对应的 DbFile，调用相应的函数，最后再进行标记更新即可。在写报告过程中，经同学提醒，将修改过的 page 放入缓存中时，需要考虑空间不足而调用 evict-

Page。但思考到在 DbFile 插入删除时都是通过 bufferPool 的 getPage 函数获取 page 的，已经保证了该页在缓存中，是否还有必要考虑空间不足。**考虑并发处理**，有可能在该函数处理过程中缓存中又放入了其他的 page 导致空间不足，故有必要添加。

4 Exercise 4

4.1 Insert

Insert 将从子迭代器中读取的 tuple 全部插入到 page 中。根据构造函数声明相应的成员变量即可，但需判断子迭代器读取的 tuple 的 tupleDesc 要与将插入的 table 相匹配。

fetchNext 从子迭代器中不断读取 tuple，并调用 BufferPool 的 insertTuple 做插入。函数返回一个只有一个字段的 tuple，该字段记录插入的 tuple 数。但当 fetchNext 被调用多次时需要返回 null，故在类中还需声明 **boolean** 变量来记录当前迭代器的状态，如果已经被调用过则不再做插入。

fetchNext 函数

```

1  protected Tuple fetchNext() throws TransactionAbortedException, DbException {
2      // some code goes here
3      // insert DOES NOT need check to see if a particular tuple is a duplicate
4      // before inserting it.
5      // 报错-> return null if called more than once
6      if(isCalled)
7          return null;
8
9      isCalled=true;
10     int count=0;
11     while(child.hasNext()){
12         Tuple t=child.next();
13         try {
14             Database.getBufferPool().insertTuple(tid, tableId, t);
15             count++;
16         } catch (DbException e) {
17             // TODO Auto-generated catch block
18             e.printStackTrace();
19         } catch (IOException e) {
20             // TODO Auto-generated catch block
21             e.printStackTrace();
22         } catch (TransactionAbortedException e) {
23             // TODO Auto-generated catch block
24             e.printStackTrace();
25         }
26     }
27     Tuple ret=new Tuple(td);
28     ret.setField(0, new IntField(count));
29     return ret;
30 }

```

4.2 Delete

Delete 将从子迭代器中读取的 tuple 全部删除，整体上与 Insert 的实现类似，在实现 fetchNext 函数时去循环调用 BufferPool 的 deleteTuple 即可。

5 Exercise 5

5.1 BufferPool

flushPage 函数刷新一页，如果该页已经被标记为脏，则调用 HeapFile 的 writePage 将他重新写入磁盘，并更新标记。

discardPage 函数被 evictPage 调用，完成最后一步工作。不经 flush 直接从缓存中移出相应的页。

evictPage 函数逐出缓存中的某一页。在逐出某一页时首先需要对其进行刷新操作，确保不将脏数据写入磁盘，而后调用 discardPage 将其从缓存中删除。

evictPage 采用的是 **LRU 原则（最近最少使用原则）** 每次都选择最近最少使用的 page 进行逐出。实现上，定义一个 pageOrder 链表：在将某一页放入缓存时将它的 pageId **链入到链表尾部**，如 getPage 读入 page 时；在使用到某页时也将它**从链表摘出放到尾部**，如 insertTuple 更新某一页时。在 evictPage 函数内，每次都选择链表头节点，即是最近最少使用的 page。

evictPage 函数

```
1 private synchronized void evictPage() throws DbException {
2     // cache-> LRU原则
3     PageId pid=pageOrder.getFirst();
4     try {
5         flushPage(pid);
6     } catch (IOException e) {
7         // TODO Auto-generated catch block
8         e.printStackTrace();
9     }
10    discardPage(pid);
11 }
```

更新 pageOrder

```
1 pageOrder.remove(page.getId()); // 最近进行了调用，LRU原则对它进行更新
2 pageOrder.add(page.getId());
```

6 Git Commit History





























04 Apr, 2023 1 commit		
 review & revise details during report writing tang authored 1 minute ago	516e28b5	 
01 Apr, 2023 2 commits		
 Lab2 exercise5 tang authored 3 days ago	fd7823ed	 
 Lab2 exercise4 tang authored 3 days ago	1151ce78	 
31 Mar, 2023 1 commit		
 Lab2 exercise3 tang authored 4 days ago	5660406d	 
30 Mar, 2023 1 commit		
 Lab2 exercise3 HeapPage.java tang authored 4 days ago	ae388962	 
28 Mar, 2023 1 commit		
 Lab2 exercise2 tang authored 1 week ago	3e8bc066	 
27 Mar, 2023 1 commit		
 Lab2 part of exercise2 tang authored 1 week ago	6f6cfcdb	 
26 Mar, 2023 1 commit		
 Lab2 exercise1 tang authored 1 week ago	c7871de5	 
17 Mar, 2023 2 commits		
 Lab 1 tang authored 2 weeks ago	 Lab1 c3afae57	 

图 6.1: Git Commit History