



南開大學
Nankai University

计算机学院
数据库系统实验报告

SimpleDB Lab-2

姓名：曹珉浩

学号：213619

专业：计算机科学与技术

2023 年 4 月 2 日

目录

1	Exercise-1	2
2	Exercise-2	3
2.1	IntegerAggregator	3
2.2	Aggregate	5
3	Exercise-3	5
3.1	HeapPage	5
3.2	HeapFile	6
4	Exercise-4	7
5	Exercise-5	8
6	Commit	9

1 Exercise-1

Predicate(谓词) 将元组与指定的字段值进行比较, 在该类中有一个内部类, 是一个枚举, 定义了七个枚举值, 每一个都是 Op 的实例。

JoinPredicate 使用谓词比较两个元组的字段, 它最有可能被 Join 运算符使用。

Filter 是一个实现关系选择的运算符, 在该类中有一个 fetchNext() 函数值得说明, 这个函数迭代来自子运算符的元组, 将谓词应用于它们并返回那些通过谓词的元组, 意思就是找 Predicate.filter() 返回 true 的元组, 代码实现如下:

```

1  protected Tuple fetchNext() throws NoSuchElementException,
2      TransactionAbortedException, DbException {
3      Tuple temp=null;
4      while(child.hasNext()) {
5          temp=child.next();
6          if(pred.filter(temp))
7              return temp;
8      }
9      return null;
10 }

```

Join 运算符实现关系连接操作。该类中有一个关键函数 fetchNext(), 返回连接生成的下一个元组, 这个函数的过程实际上就是一个类似于求笛卡尔积的过程, filter(t1,t2) 就是要对每一个 t1 遍历一次 t2, 来寻找满足条件的关系, 为了保证能找到所有的结果, t1 应放在函数外面, 通过迭代器不断 next, 遍历所有 t1, 然后在函数内部, 声明元组 t2, 不断遍历寻找满足 filter(t1,t2) 的结果, 找到就返回他们的连接, 找不到就返回空值, 要注意每次执行完函数都要重置关系 2 的迭代器, 代码的具体实现如下:

```

1  protected Tuple fetchNext() {
2      while(child1.hasNext()||t1!=null) { //关系 1 未遍历完或者此时 t1 不为空, 算法就没结束
3          if(child1.hasNext()&&t1==null)
4              t1=child1.next();
5          while(child2.hasNext()&&t1!=null) {
6              Tuple t2=child2.next();
7              if(this.pred.filter(t1, t2)) {
8                  Tuple union=new Tuple(this.getTupleDesc());
9                  int i=0;
10                 for(;i<t1.data_size();i++)
11                     union.setField(i,t1.data.get(i));
12                 for(int j=0;j<t2.data_size();j++)
13                     union.setField(i+j,t2.data.get(j));
14                 return union;
15             }
16         }
17     }
18 }

```

```

17         t1=null;
18         child2.rewind(); //重置 child2 迭代器
19     }
20     return null;
21 }

```

2 Exercise-2

在本实验中，使用的五个聚合运算符为 max, min, sum, count 和 avg, sum 用于计算匹配条件下某个数值列的总和, avg 用于计算匹配条件下某个数值列的平均值, max/min: 用于计算匹配条件下某个数值列的最大值或最小值, count 用于计算 (指定) 行的数目, 这五个聚合运算符可以用于 SELECT 语句中的 GROUP BY 子句或者 HAVING 子句中, 以对匹配条件下的数据进行分组和统计。使用聚合函数可以方便地从数据库中提取出有用的信息。

2.1 IntegerAggregator

根据构造函数的提示, IntegerAggregator 类有以下几个字段:

- int gbfield : 元组中分组字段的从 0 开始的索引, 如果没有分组则为 NO_GROUPING
- int afield : 元组中聚合字段的从 0 开始的索引
- Type gbfieldtype : 按字段分组的类型 (例如 Type.INT_TYPE), 如果没有分组则为 null
- Op what : 使用什么聚合运算符

仅仅依靠这四个字段是无法完成指定要求的, mergeTupleIntoGroup() 函数要求将一个新的元组合并到聚合中, 按照进行什么运算 (构造函数中的 what 变量) 进行分组, 而 what 取值的几个运算都是静态整型变量, 也就是说, 我们要建立一个 Integer 和 Field 之间的关系, 因此我们采用映射:

- Map<Field,Integer> groupPairMap

采用映射还有一个好处, 就是 Key 值是唯一的, 无论之后 put 是否存在 Key, 一个 Key 也只有一个记录, 方便我们在该函数中实现将元组合并到聚合之中。

同时因为聚合函数 avg 计算列均值, 与其他四个可通过遍历实现的聚合不同, 我们单独为该函数维护一个映射, 由 Field 映射到整形数组:

- Map<Field,ArrayList<Integer> > avg

下面是 mergeTupleIntoGroup() 的具体实现过程: 我们先获得传入元组的 Field 类型, 计算出他的整形聚合值, 然后根据 what 变量的五种取值, 列出 switch 语句:

max, min, sum, count 的实现方式很相似, 只详细介绍 sum 的实现方法:

```

1 case SUM:
2     int sum=aggregateValue;
3     for(Map.Entry<Field, Integer> entry:this.groupPairMap.entrySet()) {
4         if(entry.getKey().equals(groupValue))

```

```

5         sum+=entry.getValue();
6     }
7     this.groupPairMap.put(groupValue, sum);
8     break;

```

Map 中的 `entrySet()` 方法返回一个 Set 类型的集合，其中包含了 Map 集合中的所有键值对。每个键值对都表示为一个 Map.Entry 对象，该对象包含了键 (key) 和值 (value) 两个部分。可以通过遍历 `entrySet()` 方法返回的集合来访问 Map 集合中的所有键值对。键相等 sum 累加，由上述分析，最后重新 put(Map 的键是唯一的) 即可。

然后是 avg 函数的实现：除数不能为 0，要先检查一下原先是否已有这个 Key 值，没有则新建一个整形数组，和 Field 组成映射加入到聚合，如果有就让对应的 Value 值加上这个整形聚合值，因为使用 ArrayList，我们无需担心数量问题，最后做除法即可，具体实现如下：

```

1 case AVG:
2     if(!avg.containsKey(groupValue)) {
3         avg.put(groupValue, new ArrayList<>());
4         avg.get(groupValue).add(aggregateValue);
5     }
6     else
7         avg.get(groupValue).add(aggregateValue);
8     ArrayList<Integer> sum_for_avg=avg.get(groupValue);
9     int temp=0;
10    for(int i=0;i<sum_for_avg.size();i++)
11        temp+=sum_for_avg.get(i);
12    this.groupPairMap.put(groupValue, temp/sum_for_avg.size());
13    break;

```

然后是迭代器方法，要求在组聚合结果上创建一个 OpIterator，并给出了两种元组的描述方式，我们记为 td1 和 td2，根据这两种不同的描述方式，我们要实现不同的方法。还有一点需要注意：返回类型为 OpIterator，我们不能直接返回 `tupleList.iterator()`，可以通过内部类继承之后向上转型，内部类部分为简单的抽象方法实现，不再赘述，下面给出两种 Desc 的实现方法：

```

1 先描述两个元组模式：
2  TupleDesc td1 = new TupleDesc(new Type[]{gbfieldtype,Type.INT_TYPE},
3  new String[]{"groupVal","aggregateVal"});
4  TupleDesc td2 = new TupleDesc(new Type[]{Type.INT_TYPE},
5  new String[]{"aggregateVal"});
6
7 根据两种描述方式，实现不同的方法：
8  for(Map.Entry<Field, Integer>entry:this.groupPairMap.entrySet()) {
9  if(entry.getKey()!=null) { //td1 模式
10  Tuple tuple=new Tuple(td1);

```

```

11 tuple.setField(0,entry.getKey());
12 tuple.setField(1,new IntField(entry.getValue())); //IntField 为存储单个整数的 Field 实例
13 this.tupleList.add(tuple);
14 }
15 else {
16 Tuple tuple=new Tuple(td2);
17 tuple.setField(0,entry.getKey());
18 this.tupleList.add(tuple);
19 }
20 }

```

2.2 Aggregate

Aggregate 计算聚合，在构造函数中首先要判断是整形还是字符串类型，对不同的类型要有不同的聚合器。因为构造函数给定的迭代器 `OpIterator child` 是为我们提供元组的迭代器，在迭代器打开函数中应该利用上面实现的 `merge` 函数全部聚合在一起，并不是用于判断 `next` 的，为了方便地实现功能，我们应该自己再构造一个迭代器，在 `merge` 完毕后，利用之前实现的聚合器的 `iterator` 函数赋值给这个新迭代器，新增字段如下：

- Aggregator aggregator: IntegerAggregator 或 StringAggregator
- OpIterator it: 聚合器的迭代器
- TupleDesc td: 方便两个获取名字函数的实现

`open` 函数的实现：

```

1 public void open(){
2     super.open();
3     this.child.open();
4     while(this.child.hasNext()) //把所有元组合并到聚合中
5         this.aggregator.mergeTupleIntoGroup(this.child.next());
6     it=this.aggregator.iterator();
7     it.open();
8 }

```

3 Exercise-3

3.1 HeapPage

在实验一中，我们已经实现了 `HeapPage` 的一些方法，我们将对剩下的修改表有关的方法（插入元组，删除元组）做出实现：

删除元组要考虑这个元组在不在给定的页上以及元组的槽是否已经为空的情况，插入元组要考虑这个页是否满了或者给定元组的模式不匹配等问题，这两项方法都需要用到有关槽的函数。在实验一

中，我们已经实现了判断槽是否为空的函数，要实现这两个方法，我们首先要实现一个设定槽状态的函数 `markSlotUsed()`：

```

1 private void markSlotUsed(int i, boolean value) {
2     int headerNum=i/8;
3     int movement=i-i/8*8;
4     if(value)
5         this.header[headerNum]=(byte) (header[headerNum]|(1<<movement));
6     else
7         this.header[headerNum]=(byte) (header[headerNum]&(~(1<<movement)));
8 }

```

与 `isSlotUsed()` 函数一样，我们先获得数组中的索引位置和数组的偏移量。`value` 参数是我们给定的布尔值，`true` 就是想要占用这个槽，要把目标位设为 1，而其他位不发生变化，可采用的一种简便方式是：把目标位设置为 1，而其他位为 0(通过位移量实现)，然后做一个按位或运算。同理，`value=false` 时，目标位为 0，其余位为 1，然后做按位与运算即可达成目标。

利用这个标记函数，并注意特殊情况，插入和删除函数比较容易实现，不做赘述。

3.2 HeapFile

1. writePage 方法

`writePage` 方法将传入的页面写入到 `HeapFile` 中，与实验一中的 `readPage` 方法类似，该方法也使用随机访问文件的对象，方法的实现过程就是定位到页数 * 大小的位置，即新的要写入的页，具体代码如下：

```

1 public void writePage(Page page) throws IOException {
2     int size=BufferPool.getPageSize();
3     int num=page.getId().getPageNumber();
4     byte data[]=page.getPageData();
5     RandomAccessFile randomAccessFile=new RandomAccessFile(f,"rw");
6     randomAccessFile.seek(size*num);
7     randomAccessFile.write(data);
8     randomAccessFile.close();
9 }

```

2. insertTuple 方法

该方法插入事务编号为 `tid` 的元组 `t`，并返回被修改的页的数组，根据提示，插入和删除元组的方法要使用 `BufferPool.getPage()` 方法访问页面，要注意几个问题：因为 `getPage()` 方法会获得锁，当页面已满时，该页要及时释放锁；当所有页都满时，要创建一个新的空页并写入数据，实现代码如下：

```

1 public ArrayList<Page> insertTuple(TransactionId tid, Tuple t)
2     throws DbException, IOException, TransactionAbortedException {

```

```

3  BufferPool bufferPool=Database.getBufferPool();
4  ArrayList<Page> pages=new ArrayList<Page>();
5  int tableId=this.getId();
6  int pid=0;
7  for(;pid<this.numPages();pid++) {
8      HeapPage temp=(HeapPage) bufferPool.getPage(tid,
9          new HeapPageId(tableId,pid),Permissions.READ_WRITE);
10     if(temp.getNumEmptySlots()==0)
11         Database.getBufferPool().releasePage(tid,new HeapPageId(tableId,pid)); //没有空位了，释放锁
12     else {
13         temp.insertTuple(t);
14         pages.add(temp);
15         break;
16     }
17 }
18 //前面的所有页全满了，就要新建一个空页：
19 if(pid==this.numPages()) {
20     HeapPage newPage=new HeapPage(new HeapPageId(tableId,pid),HeapPage.createEmptyPageData());
21     newPage.insertTuple(t);
22     pages.add(newPage);
23     this.writePage(newPage);
24 }
25 return pages;
26 }

```

deleteTuple 方法实现比 insertTuple 方法容易，因为构建 HeapPage 时不需要遍历，直接使用传入元组的 getRecordId().getPageId() 方法即可，不再赘述。

4 Exercise-4

Insert 和 Delete 的实现方法基本一致，只介绍 Insert 的实现方法。

Insert 中有一个关键函数 fetchNext()，它将从 child 读取的元组插入到构造函数指定的 tableId 中，返回一个包含插入记录数的单字段元组，记录数为 Type.INT，因此我们需要指定一个整形的模式，函数还需要判断重复插入的情况，如果不是第一次调用，则返回空，因此除构造函数需要的几个变量之外，我们还需要定义：

- TupleDesc td: 指定 INT 的描述模式
- Tuple tuple: 用于返回的元组
- boolean iscalled: 判断是不是第一次调用

实现代码如下：

```

1 protected Tuple fetchNext() throws TransactionAbortedException, DbException {
2     if(iscalled)
3         return null;
4     iscalled=true;
5     int num=0;
6     while(child.hasNext()) {
7         Database.getBufferPool().insertTuple(t, tableId, child.next());
8         num++;
9     }
10    tuple.setField(0,new IntField(num));
11    return tuple;
12 }

```

5 Exercise-5

当缓冲池中的页面超过 numPages 时，应在加载下一页之前将一页从缓冲池中逐出。从缓冲池中移除页面的唯一方法是 evictPage()，它应该在它驱逐的任何脏页面上调用 flushPage()，为了达成实验目的，我们应该实现 flushPage() 方法，进而实现 evictPage() 方法，最后依据这个驱逐方法，修改之前的 getPage() 方法。

flushPage() 方法用于将某个页面刷新到磁盘，参数 pid 指示要刷新的页面的 ID，根据提示，flushPage() 应该将任何脏页写入磁盘并将其标记为不脏，同时将其留在 BufferPool 中。要实现这个方法，应先使用页面的 isDirty() 函数，注意到函数返回一个事务对象，可能为 null，因此要先判断 pid 指向的页面是否为空，若非空则将其写入 BufferPool 中，在 BufferPool 中维护了一个 PageId 到 Page 的映射 Map，可以保证 Page 的唯一性，并且可以方便地利用 PageId 查找并对相应的 Page 进行操作，实现代码如下：

```

1 private synchronized void flushPage(PageId pid) throws IOException {
2     if(pages.get(pid).isDirty() != null)
3         Database.getCatalog().getDatabaseFile(pid.getTableId()).writePage(pages.get(pid));
4 }

```

flushAllPages() 方法就是利用映射遍历所有页即可，代码不再赘述，接下来利用 flushPage() 方法实现驱逐策略 evictPage()：

```

1 private synchronized void evictPage() throws DbException {
2     boolean allDirty = true;
3     for(Map.Entry<PageId, Page> entry : pages.entrySet()) {
4         if(entry.getValue().isDirty() != null) continue;
5         try {
6             allDirty = false;

```

```

7         flushPage(entry.getKey());
8     } catch (IOException e) {
9         e.printStackTrace();
10    }
11    discardPage(entry.getKey());
12    break;
13 }
14 if(allDirty)throw new DbException("All pages are dirty");
15 }

```

该驱逐策略可以概括为：遍历了当前所有的缓存页，检查它们是否被修改过（即是否为脏页），如果是脏页则跳过，否则将该页写回到磁盘，然后从缓存中删除该页。如果所有的缓存页都是脏页，则抛出异常。























最后，修改之前的 `getPage()` 方法：当缓冲池中的页面超过 `numPages` 时，应在加载下一页之前将一页从缓冲池中逐出，即在 `BufferPool` 中添加页的代码之前加入：

```

1  if(pages.size()>=numPages())
2      evictPage();

```

6 Commit

02 Apr, 2023 2 commits		
 完成了Lab2, 上一条commit是Exercise-4, 打错了()	cmh authored 1 minute ago	c6ed6774  
 完成了Lab2(上一条commit记录是Exercise-4, 打错了)	cmh authored 1 minute ago	0e23b73e  
31 Mar, 2023 1 commit		
 完成了Exercise-5	cmh authored 1 day ago	5ff1b7bb  
28 Mar, 2023 1 commit		
 完成了Exercise-3	cmh authored 5 days ago	e572e156  
27 Mar, 2023 1 commit		
 完成HeapPage	cmh authored 5 days ago	d8576e21  
26 Mar, 2023 1 commit		
 完成了Exercise-2	cmh authored 6 days ago	d6ddcfb1  
25 Mar, 2023 1 commit		
 通过了IntegerAggregator的测试	cmh authored 1 week ago	b666ee45  
21 Mar, 2023 1 commit		
 完成了Lab2的Exercise-1	cmh authored 1 week ago	a3340835 