

Lab 2 report







姓名：戚晓睿

学号：1811412

Github: <https://github.com/NickSkyyy/SimpleDB>











GitCommitPic: see it below

Commits on Mar 30, 2020

2.5 finished. NickSkyyy committed 7 days ago	 e6a8bc9	
2.4 finished. NickSkyyy committed 7 days ago	 64d03cb	
2.3 finished. NickSkyyy committed 7 days ago	 80322f5	





滚动鼠标轴或单击，开始截长图

Commits on Mar 29, 2020

2.2 last one. ... NickSkyyy committed 8 days ago	 93f3442	
2.2 HeapPage finished. NickSkyyy committed 8 days ago	 8b3638e	
2.2 finished. NickSkyyy committed 8 days ago	 c9ebd6e	
2.2 StrAgg finished. NickSkyyy committed 8 days ago	 e6043e8	
2.2 IntAgg finished. NickSkyyy committed 8 days ago	 f7136c8	









滚动鼠标轴或单击，开始截长图

Commits on Mar 27, 2020



2.1 finished. NickSkyyy committed 10 days ago	 0914996	
2.1 finished without system test JoinTest. ... NickSkyyy committed 10 days ago	 2186297	

滚动鼠标轴或单击，开始截长图

Commits on Mar 25, 2020

2.1 Filter finished. ... NickSkyyy committed 12 days ago	 7e9f024	
2.1 predicate finished. NickSkyyy committed 12 days ago	 b3ded9d	
Start NickSkyyy committed 12 days ago	 607ab89	
2.1 Predicate without filter. NickSkyyy committed 12 days ago	 1d1d492	

Commits on Mar 17, 2020

Lab 1	 70a2a39	
-------	---	---

1 设计思路

Lab2设计的是有关数据库诸多操作（Operator）的实现，包括链接（Join.java）、聚类（Aggregator.java）等等。

难度在于理解每一个操作的实现逻辑。这里采取的方法是通过查看Unit Test代码中给的例子进行实现逻辑的理解和内化。当把握了整体的逻辑，再回头实现代码就显得比较轻松。

第二部分将分练习，详细介绍每一块儿的内容。

2 详细介绍

2.1 Filter & Join

对两类操作（过滤、连接）进行实现，每一类操作先定义断言（判断）类（*Predicate.java）用于实现字段之间的匹配，再定义操作（Filter.java, Join.java）类用于实现业务逻辑。

过滤实现的内容是，通过和给定字段的比较匹配新元组，不断获取符合要求的元组直到方法结束；连接实现的内容是，通过对两个元组内指定的两个字段进行比较，合并（merge）匹配成功的两个元组，返回新值。

实现过程中涉及到Iterator的处理，需要阅读**AbstractDbFileIterator.java**获得帮助。最初写的时候没有注意，还是打算使用HeapFile时，对DbFileIterator的处理模式。但后来在读帮助文档的时候发现了这一点，及时做了纠正。

主要介绍方法：Join.fetchNext.

2.1.1 Join.fetchNext

一开始写的循环如下：

```
1 while (it1.hasNext()) {
2     ...
3 }
4 while (it2.hasNext()) {
5     ...
6 }
```

犯了很低级的失误。并没有好好读题，以为是需要记录AxB和BxA的两个结果。在知道是**计算笛卡尔积**之后，写出如下的循环实现业务逻辑：

```
1 while (it1.hasNext()) {
2     Tuple t1 = it1.next();
3     while (it2.hasNext()) {
4         Tuple t2 = it2.next();
5         if (p.filter(t1, t2))
6             tuples.add(merge(t1, t2));
7     }
8     it2.rewind();
9 }
```

这里使用了一个tuples数组记录组合后的结果，因为这样的计算方式使得fetchNext无法记录当前计算到的位置，所以使用该数据结构进行辅助，curTuple作为一个整数指针指向应该取出哪一个tuple作为输出。

```
1 if (curTuple != -1)
2     return ++curTuple < tuples.size() ? tuples.get(curTuple) : null;
```

合并的时候也出现了一点点问题，直接使用TupleDesc.merge方法并不可行，由于原方法的样式不能改变，于是在这里又按着merge方法的思路复现了一遍。

2.2 Aggregates

对两种字段 (Integer, String) 实现聚类操作, 先定义父类聚类 (Aggregate.java), 之后实现针对两种字段的子聚类 (*Aggregator.java)。由于Integer和String的实现差别不大, 且Integer需要实现的功能更多, 这里以IntegerAggregator为例做详细说明。

主要介绍: IntegerAggregator.mergeTupleIntoGroup, IntegerAggregator.iterator。

2.2.1 IntegerAggregator.mergeTupleIntoGroup

第一点, 我们使用两个Map记录生成的聚类结果:

```
1 private Map<Field, Integer> groups; // (groupVal, aggregateVal)
2 private Map<Field, Integer> cnt; // count the tuples number of each field
```

groups记录对应组字段的结果, cnt记录对应组一共操作了多少字段, **主要适用于Op.AVG的情况**。因为计算平均数的时候可能出现**无法整除**的情况, 统一记录和式以及字段个数是最好的处理方法。

```
1 if (op == Op.AVG) {
2     int val = ((IntField)tup.getField(aNum)).getValue(), len = 1;
3     if (cnt.containsKey(key)) {
4         len = cnt.get(key);
5         val += groups.get(key);
6         len++;
7     }
8     groups.put(key, val);
9     cnt.put(key, len);
10    return;
11 }
```

第二点, 关注结果可以发现, 返回结果有两种: **group和non-Group**。结合Op中提到的non-Group标记我们设计了**Map中null键表示non-Group**的情况。

```
1 Field key = gbNum == -1 ? null : tup.getField(gbNum);
```

2.2.2 IntegerAggregator.iterator

根据上面2.2.1提到的结果的特殊性, 这里参照HeapFile中iterator的实现, 创建实体类IntIterator实现Oplerator接口, 对两种结果分别创建对应的结果类型:

```
1 if (key == null) {
2     typeAr = new Type[] {Type.INT_TYPE};
3     td = new TupleDesc(typeAr);
4     t = new Tuple(td);
5     t.setField(0, new IntField(groups.get(key) / cnt.get(key)));
6 }
7 else {
8     typeAr = new Type[] {gbType, Type.INT_TYPE};
9     td = new TupleDesc(typeAr);
10    t = new Tuple(td);
11    t.setField(0, key);
12    t.setField(1, new IntField(groups.get(key) / cnt.get(key)));
13 }
```

结果分类创建后放入IntegerAggregator类中的数组中存储, 返回普通数组指针供IntIterator使用。后附上完整类实现代码:

```

1 public class IntIterator implements OpIterator {
2     private TupleDesc td;
3     private Iterator<Tuple> it;
4     public IntIterator() {
5         Type[] typeAr;
6         if (tuples == null)
7             tuples = new ArrayList<>();
8         Set<Field> keySet = groups.keySet();
9         Iterator<Field> it = keySet.iterator();
10        for (int i = 0; i < groups.size(); i++) {
11            Field key = it.next();
12            TupleDesc td;
13            Tuple t;
14            if (key == null) {
15                typeAr = new Type[] {Type.INT_TYPE};
16                td = new TupleDesc(typeAr);
17                t = new Tuple(td);
18                t.setField(0, new IntField(groups.get(key) /
19 cnt.get(key)));
20            }
21            else {
22                typeAr = new Type[] {gbType, Type.INT_TYPE};
23                td = new TupleDesc(typeAr);
24                t = new Tuple(td);
25                t.setField(0, key);
26                t.setField(1, new IntField(groups.get(key) /
27 cnt.get(key)));
28            }
29            tuples.add(t);
30        }
31        @Override
32        public void open() throws DbException, TransactionAbortedException {
33            it = tuples.iterator();
34        }
35        @Override
36        public boolean hasNext() throws DbException, TransactionAbortedException
37        {
38            return it.hasNext();
39        }
40        @Override
41        public Tuple next() throws DbException, TransactionAbortedException,
42        NoSuchElementException {
43            return it.next();
44        }
45        @Override
46        public void rewind() throws DbException, TransactionAbortedException {
47            open();
48        }
49        @Override
50        public TupleDesc getTupleDesc() {
51            return td;
52        }
53        @Override
54        public void close() {
55            it = null;
56        }
57    }
58 }

```

2.3 HeapFile Mutability

Lab1中已经对HeapPage和HeapFile进行了Read的实现，这次实现Write操作。包括元组的增添和删除，以及写入磁盘等操作。和Lab1中的逻辑很像，对于Write操作，HeapPage实现**元数据与页更新**，是最为主要的实现部分，HeapFile实现**表更新**，BufferPool面向更为底层的**缓冲池和磁盘**，任何对Page的调用都需要经过BufferPool.getPage的获取。

明白了上面这一点，实现插入和删除，在实现**保证系统和磁盘信息的同步**时就能更清晰。

主要介绍三个方法：HeapPage.markSlotUsed，HeapPage.deleteTuple，HeapFile.insertTuple。

2.3.1 HeapPage.markSlotUsed

介绍这一个方法的原因是精巧的位运算更新头指针信息，插入时使用对应位为1的或运算使头指针更新，删除时使用对应位为0的与运算使内容删除。

```
1  if (value)
2      header[i / 8] = (byte)(header[i / 8] | (1 << (i % 8)));
3  else
4      header[i / 8] = (byte)(header[i / 8] & ~(1 << (i % 8)));
```

2.3.2 HeapPage.deleteTuple

结合上一点介绍的内容，这里格外强调一下。delete卡了很久，也尝试咨询了一些同学的实现方法。最初会报错，提示信息是在操作指针的时候，Map**内容发生了变更**。

```
1  markSlotUsed(i, false);
2  usedTp.remove(t);
```

经过网上查阅资料和多次修改，定位到这样一个小问题上。删除时，如果已经在**header**中进行过增删的操作，则**无需**进行再一次更新，执行remove操作反而会**导致iterator遍历内容变更**出现错误。

2.3.3 HeapFile.insertTuple

向页中插入元组时可能插入多条或者**没有任何页被插入**，此时需要**建立新的页**，因此在循环处理后需要检查插入长度，并对创建的新的页面进行处理。

```
1  if (pages.size() == 0) {
2      HeapPageId pid = new HeapPageId(getId(), maxPage++);
3      HeapPage hp = new HeapPage(pid, HeapPage.createEmptyPageData());
4      writePage(hp);
5      hp = (HeapPage)Database.getBufferPool().getPage(tid, pid,
6          Permissions.READ_WRITE);
7      hp.insertTuple(t);
8      pages.add(hp);
9  }
```

注意到尽管没有要求实现writePage的操作，但是在Lab1中readPage的实现已经提供了样例，**RandomAccessFile**的使用正是实现该方法的关键，这里也将第4行的实现附在下面：

```

1  try {
2      RandomAccessFile raf = new RandomAccessFile(f, "rw");
3      raf.seek(page.getId().getPageNumber() * BufferPool.getPageSize());
4      byte[] data = page.getPageData();
5      raf.write(data);
6  } catch (Exception e) {
7      e.printStackTrace();
8  }

```

2.4 Insertion & deletion

实现插入 (Insert.java) 和删除 (Delete.java) 操作对Operator接口实现的实体类。

主要介绍两个方法: Insert.fetchNext, Delete.fetchNext.

2.4.1 Insert.fetchNext

最初实现时将下面的代码放在了这一部分:

```

1  while (it.hasNext()) {
2      Tuple t = it.next();
3      try {
4          Database.getBufferPool().insertTuple(tid, id, t);
5          cnt++;
6      } catch (Exception e) {
7          e.printStackTrace();
8      }
9  }

```

cnt负责记录插入处理的个数, 但是测试并没有通过。仔细阅读注释后发现, fetchNext方法的返回值, **是一个new出的新值**。在此处不能进行iterator的遍历, 应该是一个**结果**, 执行到此处时应该已经插入完毕。

于是做出的处理是, 将上述代码转移到了Insert.open里面, 作为该操作初始化时的一部分内容, 而这里仅关注产生的结果。

2.4.2 Delete.fetchNext

这里介绍方法的原因同上, 而附加强调的一点在2.3.2处有所提及。此处删除操作在测试时出现**Map动态变更**的问题, 核心却在HeapPage那里出现了小小的问题。于是出现了,

Delete → BufferPool → HeapFile → HeapPage

这样一个漫长的debug搜索过程, 会稍微吐槽一下自己之前写的代码像*一样, 这个部分卡了1-2天。

2.5 Page eviction

结合Lab1的后续问题, 对缓冲池进行多余页面的“驱逐”操作。这一问题比较开放, 自己做的处理是**根据Page压入BufferPool的时间戳**进行相应的处理, 认为**距离目前操作最远的一次操作**应该是可以被驱逐出的。于是有了如下的算法:

```
1 PageId pid = order.get(0);
2 order.remove(pid);
3 try {
4     HeapPage hp = (HeapPage)pool.get(pid);
5     if (hp.isDirty() != null)
6         flushPage(pid);
7     pool.remove(pid);
8 } catch (Exception e) {
9     e.printStackTrace();
10 }
```

此处定义了新的数据结构order按顺序记录每个压入Page的时间戳，每次evict取出最前面的进行剔除。
flushPage等操作借鉴HeapFile的writePage操作即可，实现起来不算很难。