南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

数据库实验报告

Lab2

学号：2113839

姓名：林帆

2023 年 4 月 10 日

目录

一、 exercise1

(一) Predicate

Predicate 类用于实现 Tuple 内特定位置字段和给定字段的比较，是 Filter 类的辅助类，便于后续分组聚合操作中筛选符合条件的 tuple，类内部分别有三个成员变量：字段位置 field，比较方式 op 和给定比较字段 operand。

类内的 filter 函数实现比较过程，返回比较结果。函数参数为 tuple，通过 getfield 函数得到给定位置的字段后，调用 field 类内自带的 compare 函数，传入参数比较方式 op 和比较字段 operand，返回 boolean 型比较结果。函数的实现如下所示：

filter() 函数的实现

```
1  public boolean filter(Tuple t) {
2      //两个比较对象operand和t.field+比较方法op，用compare实现
3      return t.getField(field).compare(op,operand);//反过来不行
4  }
```

问题思考：尝试发现，调用 operand 的 compare 函数却不行，是有可能为 Null 的缘故?

(二) JoinPredicate

JoinPredicate 类用于实现对两个 tuple 给定位置字段的比较。相比于 Predicate，只是把给定字段 operand 换成另一 tuple 的特定位置字段，通过 getfield 获取该位置的字段后，其他函数的实现与 Predicate 无异。

(三) Filter

Filter 类用于实现给定条件下对 Tuple 的筛选和过滤, 其中对 Tuple 的过滤条件由 Predicate 型变量 p 对应的给定字段及比较方式确定。因而该类中有两个主要的成员变量：上述提到的变量 p 和用于读取待筛选过滤的 tuples 的迭代器 child（OpIterator 型)。

由于 Filter 类由 Operator 继承而来，因此需要实现 Operator 的基本函数：open()，close() 和 rewind() 等。特别注意，在 open() 函数中，需要先执行父类 open，因为在继承的 hasNext 中会调用父类进行判断。open() 函数如下所示，close() 函数也同理：

open() 函数的实现

```
1  public void open() {
2      super.open();
3      child.open();
4  }
```

Filter 类中需要重点实现的是 fetchNext() 函数，用于返回 child 中下一个符合筛选条件的 tuple，实现思路是通过 child.next() 迭代查找下一个待读 tuple，用 Predicate 类中实现的 filter 函数对当前的 tuple 进行判断。若判断返回值为 true，则返回当前 tuple；否则继续读取 child 中的下一个 tuple，直到 filter(tuple) 为 true，或读完 child 中所有的 tuple。函数体如下所示：

fetchNext() 函数的实现

```
1  protected Tuple fetchNext() throws NoSuchElementException,
2          TransactionAbortedException, DbException {
```

```
        //返回下一个满足过滤条件的tuple
        //先判断还有没有下一个待过滤的tuple
        if(child==null)
            return null;
        while(child.hasNext()) {
            Tuple temp=child.next();
            //执行.next时迭代器自动往后跳一位
            if(p.filter(temp))
                return temp;
            if(child==null)
                return null;
        }
        return null;
    }
```

Filter 类还需要实现对迭代器的包装和设置。在 getChildren() 中建立 OpIterator 类的数组 children，将 child 添加。setChildren() 中读取 children 数组中的第一个元素，赋值给 child。

(四) Join

Join 类实现对满足过滤条件的两组 tuple 的连接操作。类中主要有三个成员变量：指向第一组 tuple 的迭代器 child1，指向第二组 tuple 的迭代器 child2 和判断两个 tuple 是否满足筛选条件的 JoinPredicate 型变量 p。除了基于父类 Operator 的基本函数的实现外，还有下面几个需要实现的函数。

getJoinField1Name() 和 getJoinField2Name() 函数需要返回给定的字段名。先获取 p 中给定的两种 tuple 的对应字段位置偏移量 index，然后通过调用迭代器的 getTupleDesc() 函数获取表头，再调用表头的函数 getFieldName(index) 即可。getJoinField1Name() 函数的返回语句如下所示：

getJoinField1Name() 函数的实现

```
    //返回tuple1需要比较的给定字段名
     int index=p.field1;
     return child1.getTupleDesc().getFieldName(index);
```

getTupleDesc() 函数需要返回两种 tuple 顺序连接后的表头。分别获取两个表头后，使用 TupleDesc 类里的 merge() 函数创建新表头并返回即可。

fetchNext() 函数依然需要重点实现，用于返回下一个合并好的 Tuple。考虑该函数前，可以先实现合并两个 tuple 的函数 **createLink()**。因而可在 **Tuple 类**中定义该函数。实现思路为：先通过 merge 创建新表头，继而创建新 tuple，再依次将两个 tuple 的 field 依次赋给新 tuple，然后返回即可。createLink() 函数的实现如下所示：

createLink() 函数的实现

```
    public Tuple create_link(Tuple t){
        TupleDesc newtd=TupleDesc.merge(tupleDesc,t.tupleDesc);//注意先后顺序
        Tuple t3=new Tuple(newtd);
        int len1=tupleDesc.numFields();
        int len2=t.tupleDesc.numFields();
        for(int i=0;i<len1;i++)
```

```
            t3.fields[i]=fields[i];
        for(int i=0;i<len2;i++)
            t3.fields[len1+i]=t.fields[i];
        //recordID默认为null
        return t3;
    }
```

实现合并 tuple 函数后考虑 fetchNext() 函数的实现思路：由于两组 tuple 中的任何一个 tuple 都需要与另一组的所有 tuple 进行匹配判断，因而需要设置两层循环。将 child1 所指向的 tuple 设为外层循环，对每一个 tuple1 都需要在内层遍历所有的 tuple2，因而在返回一组合适的 tuple1 tuple2 后，需要一个 tuple 类型的变量 t1 记录当前 child1 指向的 tuple1。过程中使用 JoinPredicate 类的 filter 函数对两个 tuple 能否合并进行检验。

第一次调用 fetchNext() 前置 t1 为空，后面每次当某个 tuple1 遍历完全部 tuple2 后，将 child2 重置，并将 child1 指向 next()。因而最终 fetchNext() 函数的实现如下所示：

fetchNext() 函数的实现

```
Tuple t1;//记录当前child1的指向
 protected Tuple fetchNext() throws TransactionAbortedException,
     DbException {
     //返回合并之后的tuple
     //先检查迭代器是否为空
     if(child1==null||child2==null)
         return null;
     //一开始t1为null
     if(t1==null){
         if(!child1.hasNext())
             return null;
         else
             t1=child1.next();
     }
     while(t1!=null) {
         //在当前的t2往下找(t1为Null时child2已重置, 不为Null时child2在下一
             个)
         while (child2 != null && child2.hasNext()) {
             Tuple t2 = child2.next();
             if (p.filter(t1, t2)) {
                 Tuple t3 = t1.create_link(t2);
                 System.out.println(t3.toString());
                 return t3;
             }
         }
         //没有合适的t2,找下一个t1
         if(child1.hasNext()) {
             t1 = child1.next();//为下一个t1寻找
             child2.rewind();//把child2重置
         }
         else
             return null;
```

```
        }
        return null;
    }
```

检验操作：使用 System.out.println 操作查看 test 文件里满足条件的合并 tuple 是否正确，方便纠错。

代码简化的思考：在最开始考虑 tuple1 遍历完所有 tuple2 时的情况，试图使用将 t1 置为 null 来标记，然后再对 tuple2 进行定位。但后面考虑到，无论 tuple1 是否改变，都可以直接从当前的 t2 往下找 t1 为 Null 时 child2 已重置，指向第一个不为 Null 时 child2 在下一个 。因而无须设置 null，一开始将 t1 置为 child1 指向的第一个 tuple 即可。

二、 exercise2

(一) IntegerAggregator

IntegerAggregator 类实现了对整数类型字段的分组聚合操作。

类内主要成员变量如下所示:

变量	变量类型	含义或用途
gbfield	int	分组字段的序号
gbfieldtype	Type	分组字段的类型
afield	int	聚合操作的字段序号，字段类型为 int 型
what	Op	聚合方式
map	HashMap<Field,Integer>	key 为字段，value 为聚合结果
tally	HashMap<Field,Integer>	key 为字段，value 对应组的个数
Sum	HashMap<Field,Integer>	key 为字段，value 为聚合结果

表 1: IntegerAggregator 类内主要成员变量

mergeTupleIntoGroup 是类内最重要的函数，需要实现将新 tuple 加入对应分组，并更新当前的聚合结果。哈希表 map 用于保存分组字段和对应的聚合结果，若读入 tuple 为未存入 map 中的分组字段，则直接将 分组字段 聚合字段 直接作为聚合结果加入 map；若能在 map 中找到 key 为分组字段的键值对，则把该组的分组字段和更新后的聚合结果，作为更新后的键值对，插入哈希表 map 中。是否存在某字段构成的键值通过 Hashmap 的 containsKey 语句判断

针对整型字段实现的聚合操作共有五个。需要在函数中构造 switch 判断语句，判断当前的 what 是哪个聚合操作，然后更新 map 中的聚合结果，以及更新对应组的计数结果 保存在哈希表 tally 中 ，和对应组的求和结果 保存在哈希表 Sum 中 。

mergeTupleIntoGroup 函数的实现如下所示:

mergeTupleIntoGroup() 函数的实现

```
public void mergeTupleIntoGroup(Tuple tup) {
    //将tup的gpfield加入分组
    Field f;
    //若无分组
    if(gbfield==NO_GROUPING) {
        f=null;
```

```
        }
        //有分组
        else{
            f=tup.getField(gbfield);
        }
        //判断f是否有value值
        //找到当前需要加入的afield
        IntField aField= (IntField) tup.getField(afield);
        int aValue=aField.getValue();
        //更新聚合操作结果
        //判断聚合操作
        switch (what){
            case MIN:
                if(!map.containsKey(f)){
                    map.put(f,aValue);
                    tally.put(f,1);
                }
                else{
                    map.put(f,Math.min(map.get(f),aValue));
                    tally.put(f,tally.get(f)+1);
                }
                break;
            case MAX:
                if(!map.containsKey(f)){
                    map.put(f,aValue);
                    tally.put(f,1);
                }
                else{
                    map.put(f,Math.max(map.get(f),aValue));
                    tally.put(f,tally.get(f)+1);
                }
                break;
            case SUM:
                if(!map.containsKey(f)){
                    map.put(f,aValue);
                    tally.put(f,1);
                }
                else{
                    map.put(f,map.get(f)+aValue);
                    tally.put(f,tally.get(f)+1);
                }
                break;
            case COUNT:
                if(!map.containsKey(f)){
                    map.put(f,1);
                    tally.put(f,1);
                }
                else{
```

```
                    map.put(f,map.get(f)+1);
                    tally.put(f,tally.get(f)+1);
                }
                break;
            case AVG:
                if(!map.containsKey(f)){
                    map.put(f,aValue);
                    tally.put(f,1);
                    Sum.put(f,aValue);
                }
                else{
                    tally.put(f,tally.get(f)+1);
                    Sum.put(f,Sum.get(f)+aValue);
                    int count=tally.get(f);
                    int sum=Sum.get(f);
                    map.put(f,sum/count);//迭代返回均值会增大误差
                }
                break;
        }
    }
```

改进过程：一开始使用数组分组存储 tuple，通过 int 型变量记录个数等。但思考发现，无须存储 tuple，只需读取后提取相应字段信息，更新聚合结果即可。另外，哈希表有更简单的遍历方法查找键值对，比循环遍历查找更高效，而且减少了所需变量个数。

细节点注意：在实现求平均值的操作时，每一次的平均值都四舍五入，会产生误差。因而不能通过迭代求取前一次平均值来计算当前的平均值，需要建立哈希表 Sum，记录对应组当前的求和，除以对应 tally，才能避免迭代误差的产生。

实现 iterator 时，想到 HeapFile 中 iterator 的实现，采取构造内部类的方法，返回由分组字段和聚合值构成的 tuple 的迭代器。

内部类 IntegerIterator 的实现有如下考虑：创建 Tuple 类的 Iterator，通过遍历存储聚合结果的哈希表 map，实现结果 tuple 的构造和在迭代器中的装载，以及迭代器相应函数。

IntegerIterator 中的 open 函数，实现了结果 tuple 的构造和装载。首先创建 Tuple 型的 ArrayList，通过 map 的迭代器遍历 map 并通过 ArrayList 的 add 函数装载结果 tuple。最终将 ArrayList 的迭代器赋值给 IntegerIterator 类的迭代器。函数体如下所示：

open() 函数的实现

```
public void open() throws DbException, TransactionAbortedException {
        //所有结果元组全部装入it中
        //创建数组
        ArrayList<Tuple> res=new ArrayList<Tuple>();
        //遍历哈希表，加入tuple（gp,agg）
        Iterator<Map.Entry<Field, Integer>>mapIt = map.entrySet().
            iterator();

        while (mapIt.hasNext()) {
            Map.Entry<Field, Integer> entry =mapIt.next();
            Field gpf= entry.getKey();
```

```
                IntField agg=new IntField(entry.getValue());
                Tuple tp=new Tuple(getTupleDesc());
                if(gbfield!=NO_GROUPING){
                    tp.setField(0,gpf);
                    tp.setField(1,agg);
                }
                else{
                    tp.setField(0,agg);
                }
                res.add(tp);
            }
            it= res.iterator();
        }
```

getTupleDesc() 函数返回结果表头，该函数实现过程中需要对 gbfield 是否为空进行讨论，然后构造相应的 TypeAr[ ] 和 FieldAr[ ]，然后通过这两个数组创建表头并返回。

内部类实现后，IntegerAggregator 中的 iterator() 函数只需通过返回一个 new 出的 IntegerIterator 即可。

(二) StringAggregator

StringAggregator 与 IntegerAggregator 基本完全一致，但由于聚合字段是 String 类型，因而在聚合方式中，只实现对该字段的计数操作即可。

(三) Aggregate

Aggregate 类实现了对 IntegerAggregator 和 StringAggregator 的封装。在类中，除了有给定分组字段的数组下标 gfield 和给定聚合字段的数组下标 afield 外，还需要一个提供读取待分组聚合数组的 OpIterator child，以及给定的聚合操作 aop，和聚合操作器 agg（Aggregator 型，是 IntegerAggregator 和 StringAggregator 的父类）。

在聚合前，需要先判断 agg 的类型，然后使用聚合器的 mergeTupleIntoGroup 函数进行分组聚合操作。

其中 open 函数实现把所有 tuple 读入并分组聚合，调用 mergeTupleIntoGroup 函数，如下所示：

open() 函数的实现

```
public void open() throws NoSuchElementException, DbException,
        TransactionAbortedException {
    super.open();
    child.open();
    //把所有tuple读入并分组聚合
    while(child.hasNext()){
        Tuple temp=child.next();
        agg.mergeTupleIntoGroup(temp);
    }
    agg.iterator().open();
}
```

getTupleDesc() 函数按要求返回表头，注意分情况考虑是否有 gfield 的分组操作。

三、 exercise3

(一) HeapPage

首先实现 markSlotUsed 函数，标记某行被用过。只需要找到该行在 header 中的对应字节位，然设为 1 即可，函数的实现如下所示:

markSlotUsed() 函数的实现

```
private void markSlotUsed(int i, boolean value) {
    int index=i/8;
    int pos=i%8;
    if(value) {// 为真，已使用，置1
        header[index]=(byte)(header[index]|(0x1<<pos));
    }
    else {//置0
        header[index]=(byte)(header[index]&(~(0x1<<pos)));
    }
}
```

实现 insertTuple 函数时，首先检查表头结构是否对应，然后判断该页是否有已满，若有空位，则遍历找到空位，插入 tuple，调用 markSlotUsed 标记该行已用，并为该 tuple 创建 RecordId。函数体如下所示:

insertTuple() 函数的实现

```
public void insertTuple(Tuple t) throws DbException {
    //先检查表头结构是否对应
    TupleDesc td2=t.getTupleDesc();
    if(!td.equals(td2))
        throw new DbException("mismatch");
    //再检查是否全满
    if(getNumEmptySlots()==0)
        throw new DbException("全满");
    //找空位
    int key=-1;
    for(int i=0;i<numSlots;i++){
        if(!isSlotUsed(i)){
            key=i;
            break;
        }
    }
    //添加
    RecordId newid=new RecordId(pid,key);
    t.setRecordId(newid);
    tuples[key]=t;
    //标志位
    markSlotUsed(key,true);
```

```
    }
```

实现 deleteTuple 函数时，不用像 insert 一样逐行遍历，只需要读取 tuple 的 RecordId，直接定位其所在行即可，如下所示：

deleteTuple() 函数的实现

```
public void deleteTuple(Tuple t) throws DbException {
    int index=t.getRecordId().getTupleNumber();
    if(tuples[index]==null||!isSlotUsed(index))
        throw new DbException("不匹配");
    if(!tuples[index].equals(t))
        throw new DbException("不匹配");
    tuples[index]=null;
    markSlotUsed(index,false);
}
```

markDirty 函数标记该页为脏，表明下次读取数据前应先刷新页面。

(二) HeapFile

针对该类中的 insertTuple，需要遍历所有页，查找空位，添加 tuple；若该 file 中的所有 page 都满，则需通过输入流 RandomAccessFile 类创建新 page，并将 tuple 加入其中。最终返回由插入 page 构成的列表。

针对该类中的 deleteTuple 函数，同 HeapPage 的删除操作，无须逐页逐行遍历查找，只需通过读取需要删除的 tuple 的 RecordId，定位到该页该行后删除即可。deleteTuple 函数的实现如下所示：

deleteTuple() 函数的实现

```
public ArrayList<Page> deleteTuple(TransactionId tid, Tuple t) throws
    DbException,
        TransactionAbortedException {
    //创建返回列表
    ArrayList<Page> res=new ArrayList<Page>();
    //直接根据tuple的RecordId找到对应页
    HeapPageId pid=(HeapPageId) t.getRecordId().getPageId();
    HeapPage page=(HeapPage) Database.getBufferPool().getPage(tid,pid,
        Permissions.READ_WRITE);
    //调用该页的delete函数
    page.deleteTuple(t);
    res.add(0,page);
    return res;
}
```

(三) BufferPool

BufferPool 中的 insertTuple 将给定 tuple 插入到给定 table 中,通过 getDatabaseFile(tableId) 找到表对应的 HeapFile，调用此 File 中的 insertTuple 函数，并使用 markDirty 标记该页为脏，实现如下所示：

insertTuple() 函数的实现

```
public void insertTuple(TransactionId tid, int tableId, Tuple t)
        throws DbException, IOException, TransactionAbortedException {
    //确定heapFile
    HeapFile heapFile = (HeapFile) Database.getCatalog().getDatabaseFile(
        tableId);
    //调用file.insert
    ArrayList<Page> page = heapFile.insertTuple(tid, t);// 调用heapFile插
        入，返回修改过的page
    for(Page p:page){
        p.markDirty(true,tid);// 插入了 标记为脏数据
        pageStore.put(p.getId().hashCode(),p);
    }
}
```

DeleteTuple 函数依然需要通过 Tuple 的 RecordId 定位到文件、页和行，并调用 file 中的 deleteTuple 函数删除 tuple，标记页为脏即可，实现与 insertTuple 相似。

四、 exercise4

(一) Insert

Insert 将给定的 tuple 插入给定 table 中，需要调用 BufferPool 中的 insertTuple 函数。在 fetchNext 中，循环读取 child 中的 tuple。并设置变量 count 用来计数，并将最终的计数结果包装成 tuple 返回。函数实现如下所示：

fetchNext() 函数的实现

```
public void insertTuple(TransactionId tid, int tableId, Tuple t)
        throws DbException, IOException, TransactionAbortedException {
    //确定heapFile
    HeapFile heapFile = (HeapFile) Database.getCatalog().getDatabaseFile(
        tableId);
    //调用file.insert
    ArrayList<Page> page = heapFile.insertTuple(tid, t);// 调用heapFile插
        入，返回修改过的page
    for(Page p:page){
        p.markDirty(true,tid);// 插入了 标记为脏数据
        pageStore.put(p.getId().hashCode(),p);
    }
}
```

(二) Delete

Delete 将给定的 tuple 删除，需要调用 BufferPool 中的 deleteTuple 函数。实现方式与 Insert 类似。

五、 exercise5

(一) BufferPool

在 exercise5 中需要实现 BufferPool 中的页面置换策略。当 BufferPool 满了之后，可使用 LRU（最近最少使用）原则，淘汰最佳不常用的页面，并将页面标记为脏。实现过程中用链表节点 LinkedNode，存放 pageID 和 page，方便在置换中实现移动。

LinkedNode 的实现

```
    LinkedNode head;
    // 尾节点
    LinkedNode tail;
    private void addToHead(LinkedNode node){
        node.prev = head;
        node.next = head.next;
        head.next.prev = node;
        head.next = node;
    }
    private void remove(LinkedNode node){
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }
    private void moveToHead(LinkedNode node){
        remove(node);
        addToHead(node);
    }
    private LinkedNode removeTail(){
        LinkedNode node = tail.prev;
        remove(node);
        return node;
    }
```


六、 总结

在本次实验中，实现了分类聚合操作的底层代码，以及插入删除、页面更新等操作。对数据库的理解进一步加深。