SimpleDB Lab2 实验报告

2011269 王楠舟

1. 过滤 Filter 与连接 Join 运算符的实现：通过检验元组是否满足判断条件，再对元组进行过滤/连接，并返回过滤/连接的结果，为了实现这个运算符，我们必须先实现谓词 Predicate 和连接谓词 JoinPredicate，实现判断元组是否满足过滤/连接条件。

<table>
<tr><td colspan="2">Predicate：谓词
通过构造时确定的的比较操作符，在一个特定的属性上比较不同元组和一个确定的 Field 类型。
维护的变量：
int fieldNum：记录元组中需要进行比较的字段在序号。
Op op：谓词的操作符。
Field operand：用于与元组中字段比较的操作数。</td></tr>
<tr><td colspan="2">构造函数：public Predicate(int field, Op op, Field operand)</td></tr>
<tr><td colspan="2">部分重要函数的实现</td></tr>
<tr><td>public boolean filter(Tuple t)</td><td>return t.getField(fieldNum).compare(op,operand);

返回传入的元组 t 的第 fieldNum 个字段与 operand，在操作符 op 下比较的结果，实现了过滤的具体功能。</td></tr>
</table>

<table>
<tr><td colspan="2">JoinPredicate：连接谓词
通过谓词比较两个元组的某两个字段
维护的变量：
int fieldIndex1：第一个参与比较的元组（左操作数）的字段的序号
int fieldIndex2：第二个参与比较的元组（右操作数）的字段的序号
Predicate.Op op：用于比较的操作符</td></tr>
<tr><td colspan="2">构造函数：public JoinPredicate(int field1, Predicate.Op op, int field2)</td></tr>
<tr><td colspan="2">部分重要函数的实现</td></tr>
<tr><td>public boolean filter(Tuple t1, Tuple t2)</td><td>return t1.getField(fieldIndex1).compare(op,t2.getField(fieldIndex2));

通过特点的字段和确定的比较操作符来比较两个元组，返回比较的结果。实现判读两个元组是否满足连接条件而进行过滤的功能。</td></tr>
</table>

<table>
<tr><td colspan="2">Filter
过滤器 Filter，是一个实现关系选择的操作符。
通过构造时传入的谓词比较，通过一个子迭代器读取需要过滤(比较)的元组。
维护的变量：
Predicate predicate:用于过滤元组的谓词
OpIterator childOpIterator:子迭代器，用于读入需要过滤的元组</td></tr>
<tr><td colspan="2">构造函数：public Filter(Predicate p, OpIterator child)</td></tr>
<tr><td colspan="2">部分重要函数的实现</td></tr>
<tr><td>open、close 函数</td><td>Filter 对象内部通过 open、close、rewind 函数控制子迭代器的打开、关闭、重启，需要注意的是在 Filter 内的 open、close 函数内部需要调用上一级对象的 open、close 函数，即 super.open()或 super.close()。
原因是 Filter 类是 Operator 类的子类，在调用过程中可能是通过父级对象调用 Filter，所以需要调用上一级的 open、close 函数，否则会出现空指针的现象。</td></tr>
<tr><td>protected Tuple fetchNext()</td><td>实现过滤阅读下一个元组的功能。通过迭代子运算符中的元组，将谓词应用于元组，并返回能被谓词接收的元组（没有被过滤的元组），即 predicate.filter(tuple)返回值为 true 的元组。如果没有合适的元组会返回空值 null。</td></tr>
</table>

<table>
<tr><td colspan="2">Join：
Join 是 Operator 的子类，实现连接运算符类实现了关系直接的连接操作。
通过连接谓词将符合条件的元组连接起来返回。
维护的变量：
JoinPredicate：连接谓词，用于判断两个元组是否满足连接条件
OpIterator opIterator1：连接操作中，第一个元组（左操作数）的迭代器
OpIterator opIterator2：连接操作中，第二个元组（右操作数）的迭代器
Tuple t1：记录左操作数遍历到的元组</td></tr>
<tr><td colspan="2">构造函数：public Join(JoinPredicate p, OpIterator child1, OpIterator child2)</td></tr>
<tr><td colspan="2">部分重要函数的实现</td></tr>
<tr><td>open、close 函数</td><td>Join 对象内部通过 open、close、rewind 函数控制子迭代器的打开、关闭、重启，需要注意的是在 Join 内的 open、close 函数内部需要调用上一级对象的 open、close 函数，即 super.open()或 super.close()。
原因是 Join 类是 Operator 类的子类，在调用过程中可能是通过父级对象调用 Join，所以需要调用上一级的 open、close 函数，否则会出现空指针的现象。</td></tr>
<tr><td>public TupleDesc getTupleDesc()</td><td>return TupleDesc.merge(opIterator1.getTupleDesc(),opIterator2.getTupleDesc());
由于 JoinPredicate 将两个类型的元组连接起，所以他的 tupledesc 通过调用 TupleDesc 中的静态方法 merge，将左操作数和右操作数的</td></tr>
</table>

	tupledesc 组合。
protected Tuple fetchNext()	Fetchnext 方法用于返回连接生成的下一个元组。由于只要 opIterator1、opIterator2 中的下一个元组满足连接谓词的条件，就能连接生成新的元组并返回。所以函数只需要简单的循环连接即可。 逻辑为：1.用一个变量 t1 记录当前 opIterator1 左操作数目前的 Tuple。当 t1 非空，说明左操作数有值，只需要迭代右操作数 opIterator2 找到与 t1 满足连接谓词的 Tuple t2，将 t1 与 t2 连接起来组成新的 Tuple res_t 返回即可； 2.当 opIterator2.hasnext()为 FALSE，说明 t1 作为左操作数的情况已经遍历完了，将 t1 设为 null； 3.当 t1 为 null，说明 fetchnext 需要对左操作数进行遍历，将 t1 设为 opIterator1.next()。 循环 1.2.3.步的操作直至左右子运算符都迭代结束，如此便可实现循环嵌套连接。

2. 聚合操作的实现。为了实现数据库中对表的聚合，我们将实现整数聚合器 IntegerAggregator 和字符串聚合器 StringAggregator。最后利用实现好的聚合器实现聚合操作 Aggregate 类。

IntegerAggregator： 对一组整数字段 IntField 计算聚合。 维护的变量： int gbfield：按照哪一个 field 进行分组 Group BY，这是那个 field 在元组中的序号 Type gbfieldtype：按照哪一个 field 进行分组 Group BY，这是那个 field 的类型 int afield：被聚合操作的 field 在元组中的序号 Op what：按照什么聚合操作类型 ArrayList<Integer>list：在 AVG 聚合操作中记录各个值 HashMap<Field,Integer>hashMap：保存聚合操作的结果 HashMap<Field,ArrayList<Integer>>avgHashmap：在 AVG 聚合操作中保存的临时结果 String[]fieldNames：字段名，fieldname[0]是 gbfield 的名称，fieldname[1]是 afield 的名称	
构造函数：public IntegerAggregator(int gbfield, Type gbfieldtype, int afield, Op what)	
部分重要函数的实现	
public void mergeTupleIntoGroup(Tuple tup)	实现将新传入的 Tuple tup 进行聚合操作，并根据 gbfield 进行分组。 首先需要对 gbfield 进行判断，如果 gbfield==NO_GROUPING，说明这个聚合没有发生分组，不需要记录 fieldname[0]，但无论如何都需要记录 fieldname[1]。 //分别取到聚合的字段、分组的字段和聚合字段的值 Field aField=tup.getField(this.afield); Field

	gbField=gbfield==NO_GROUPING?null:tup.getField(this.gbfield);
	//整型字段的值
	int value=((IntField)aField).getValue();
	随后需要根据 what 操作类型来通过 switch 分支语句选择具体的操作：
	switch (this.what){
	case MIN:
	if(!hashMap.containsKey(gbField)){
	hashMap.put(gbField,value);
	}
	else
	hashMap.replace(gbField,Math.min(hashMap.get(gbField),value));
	break;
	case MAX:
	if(!hashMap.containsKey(gbField)){
	hashMap.put(gbField,value);
	}
	else
	hashMap.replace(gbField,Math.max(hashMap.get(gbField),value))
	;
	break;
	case COUNT:
	if(hashMap.containsKey(gbField)){
	hashMap.replace(gbField,hashMap.get(gbField)+1);
	}
	else
	hashMap.put(gbField,1);
	break;
	case SUM:
	if(!hashMap.containsKey(gbField)){
	hashMap.put(gbField,value);
	}
	else {
	int sum=hashMap.get(gbField);
	sum+=value;
	hashMap.replace(gbField, sum);
	}
	break;
	case AVG:
	if(!hashMap.containsKey(gbField)){
	list=new ArrayList<Integer>();
	list.add(value);
	avgHashmap.put(gbField,list);
	hashMap.put(gbField,value);

	}
	else{
	ArrayList<Integer>list=avgHashmap.get(gbField);
	list.add(value);
	int avg=0;
	for(int i=0;i<list.size();i++)
	avg+=list.get(i);
	avg/=list.size();
	hashMap.replace(gbField,avg);
	}
	break;
	}
	如果 hashMap 里不存在为 gbField 的键，说明这是一个全新的分组，将 hashMap 中 gbField 对应的键值按照操作类型 what 进行初始化；如果 hashMap 中存在 gbField 键，说明已经有前面元组与该元组是同一种分组，则根据操作类型 what 对 value 值进行更新。
	但与其他操作情况不同的是，在 what==AVG 求平均值时，需要用到一个 ArrayList 保存每个需要聚合的值，在更新时插入新的 value 并将所有记录值取出重新计算平均值。但这样在计算大规模数据时效率缓慢浪费内存空间，改进方法是保存一个 Integer 型数组保存在 countHashMap 中，arr[0]保存累加和，arr[1]保存参与聚合的元组数量。

public OpIterator iterator()	return new IntegerAggregatorIterator(); 函数需要返回一个聚合操作完成后的元组的迭代器，用于访问这些元组。由于在 mergeIntoGroup 函数中将聚合结果都以 gbField 为键保存在 hashMap 中，所以 IntegerAggregatorIterator 构造阶段需要通过 IntegerAggregator 中的 hashMap 进行初始化，用一个 private ArrayList<Tuple> TupleList 保存聚合结果的元组，首先需要通过 fieldname 和 gbField 的类型初始化 TupleDesc，随后根据初始化的 TupleDesc 构造我们需要的 Tuple 对象，并通过 tuple.setField 设置元组的字段数值，随后插入到数组当中。所以最后得到的迭代器实际上是 TupleList 的迭代器。 IntegerAggregatorIterator 的具体实现如下。

```
//实现 IntegerAggregatorIterator
private class IntegerAggregatorIterator implements OpIterator{
    private HashMap<Field,Integer> hashMap;
    private Iterator<Tuple> iterator;
    private TupleDesc tupleDesc;
    private ArrayList<Tuple> TupleList;
    public IntegerAggregatorIterator(){
        hashMap=IntegerAggregator.this.hashMap;
        Type[] type=new Type[2];
        type[0]=gbfieldtype;
        type[1]=Type.INT_TYPE;
```

```
            String[] fieldName=new String[2];
            if(gbfield!=NO_GROUPING)
                fieldName[0]=fieldNames[0];
            fieldName[1]=fieldNames[1];
            if(gbfield==NO_GROUPING)
                tupleDesc=new TupleDesc(new Type[]{type[1]},new String[]{fieldNames[1]});
            else
                tupleDesc=new TupleDesc(type,fieldName);
            TupleList=new ArrayList<>();
            for(Field field:hashMap.keySet()){
                Tuple tuple=new Tuple(tupleDesc);
                if(gbfield!=NO_GROUPING) {
                    tuple.setField(0, field);
                    tuple.setField(1, new IntField(hashMap.get(field)));
                }
                else
                    tuple.setField(0,new IntField(hashMap.get(field)));
                TupleList.add(tuple);
            }
        }
        @Override
        public void open() throws DbException, TransactionAbortedException {
            iterator=TupleList.iterator();
        }
        @Override
        public boolean hasNext() throws DbException, TransactionAbortedException {
            return iterator.hasNext();
        }
        @Override
        public Tuple next() throws DbException, TransactionAbortedException,
NoSuchElementException {
            return iterator.next();
        }
        @Override
        public void rewind() throws DbException, TransactionAbortedException {
            iterator=TupleList.iterator();
        }
        @Override
        public TupleDesc getTupleDesc() {return tupleDesc;}
        @Override
        public void close() {iterator=null;}
```

StringAggregator

和 IntegerAggregator 类似，StringAggregator 是对字段为 Sting 类型的元组进行聚合操作

维护的变量：

int gbfield：按照某一个 Field 类型进行分组，gbfield 为这个字段在元组中的序号

Type gbfieldtype：按照某一个 Field 类型进行分组，gbfieldtype 为这个字段的数据类型

int afield：需要聚合的字段在元组中的序号

Op what：聚合的操作方式

HashMap<Field,Integer>hashMap：用于保存聚合结果的 hashMap

String [] fieldNames：属性名，fieldname[0]是 gbfield 的名称，fieldname[1]是 afield 的名称

构造函数：public StringAggregator(int gbfield, Type gbfieldtype, int afield, Op what)

StringAggregator 只能进行 COUNT 操作，构造函数阶段需要判断若传入的 what 不是 COUNT，则抛出异常。

部分重要函数的实现

public void mergeTupleIntoGroup(Tuple tup)	由于 StringAggregator 只能进行 COUNT 操作，所以 mergeIntoGroup 实现比较简单，只需要用哈希表 hashMap 来记录相同分组的元组的数量。
public OpIterator iterator()	return new StringAggregatorIterator(); StringAggregatorIterator 的实现与上面 IntegerAggregatorIterator 相似，不在复述。

Aggregate：

Aggregate 是 Operator 的子类，用于实际的聚合操作，通过构造时传入的子迭代器、聚合操作类型和聚合字段和分组字段在元组中的序号。

维护的变量：

OpIterator child：子迭代器，用于遍历需要聚合的元组

int afield;int gfield; 字段在元组中的序号

Aggregator.Op aop：聚合操作类型

Aggregator aggregator：具体进行聚合操作的聚合器

OpIterator iterator：获取聚合结果的元组列表的迭代器

Type type：聚合字段的数据类型

构造函数：public Aggregate(OpIterator child, int afield, int gfield, Aggregator.Op aop)

对维护的数据初始化，并根据聚合字段的数据类型，将 aggregator 指向子类对象 IntegerAggregator 或 StringAggregator。

部分重要函数的实现

public void open()	对 Aggregate 进行 open 操作意味着需要开始进行聚合操作，所以通过遍历子迭代器读出 tuple，然后利用 aggregator 对读出的元组一个个地进行聚合操作，最后将得到的 aggregator.iterator()赋值给 iterator，并调用 iterator.open();

protected Tuple fetchNext()	读出聚合结果的下一个元组。

3. 元组在 HeapPage 和 HeapFile 中的插入与删除。

HeapPage	
部分重要函数的实现	
private void markSlotUsed(int i, boolean value)	实现改变第 i 个槽的状态的函数 思路： 首先计算出第 i 个槽在 byte 上的偏移量：int byte_va=i/8; 和在 bit 上的偏移量：int bit_va=i%8; 因为在 x86 系统下小端序的存储方式，header[byte_va]中从右往左数第 bit_va 位上的位数据就是第 i 位槽的状态。 如果我们需要将第 i 位槽置为被占用的状态，即将该位上的数据设为 1，我们可以定义一个临时变量为 1，在二进制上其表示为 0000 0001，往左偏移 bit_va，例如 0001 0000（bit_va=4 的情况），而我们将得到的偏移后的临时变量与 header[byte_va]做按位或操作|，因为 0|1=1,1|1=1，这样我们不会改变其他位置槽的状态并将第 i 为槽上的数据就能被置为 1，即被占用； 同理，如果我们需要将第 i 位槽置为未被占用的状态，定义一个临时变量为 1，在二进制上其表示为 0000 0001，往左偏移 bit_va，例如 0001 0000（bit_va=4 的情况），再取反~ ，例如~0001 0000=1110 1111，我们将取反后的临时变量与 header[byte_va]做按位与操作，因为 1&1=1,1&0=0，所以我们可以不改变其他槽的状态将第 i 位槽置为 0，即未被占用。
public void deleteTuple(Tuple t)	实现在当前 page 下删除指定的元组 t 实现思路： 首先通过元组的 RecordId.getTupleNum()找到元组 t 在表中的序号（位置） 然后判断该序号(位置)对应的槽是否为空，如果为空或数组在该位置下保存的元组不等于 t，说明 t 不在这张 page 中，删除失败。 如果该 heappage 对应位置元组等于 t，则删去该元组。需要先调用 markSlotUsed 函数将该序号对应的槽设为空，然后将用于保存元组的数组对应序号下的元组设置为空值，这样便能实现从该 page 中删除特定元组。
public void insertTuple(Tuple t)	实现在该 page 下插入元组 t 实现思路： 如果元组的 getTupleDesc()与 page 保存的 tupledesc 不相等或该 page 下空槽的数量为零，则都是插入失败的情况，抛出异常。

	如果有空槽，则遍历到一个可以插入的位置，调用 markSlotUsed 函数将该位置对应的槽设为被占用的状态，然后将数组对应序号下的元组赋值为 t，用 setRecordId 函数记录下元组在 page 中保存的位置和 page 的 Id。
public void markDirty(boolean dirty, TransactionId tid)	将 this.dirty 置为参数传入的 dirty，同时将 tid 保存为 transactionId
public TransactionId isDirty()	如果 dirty 为 true，返回记录的事务 transactionId；如果为 false，则返回 null

HeapFile：	
部分重要函数的实现	

public ArrayList<Page> insertTuple(TransactionId tid, Tuple t)	向表中插入特点元组 t，并返回被修改的页的链表
	实现思路：
	首先通过 Database.getBufferPool().getPage()方法从缓冲池中获取我们需要的 heapPage，如果 heappage 的槽全被占用了，即 heappage.getNumEmptySlots()==0，意味该页下已经不能插入元组，我们继续从缓冲池中获取下一个 heappage，直到有空槽。然后调用 heappage.insertTuple()方法向页中插入元组。
	当然，可能我们从缓冲池中得到的 heappage 都是满的状态，这意味着需要为 heapfile 创建新的 heappage。
	我们可以通过下面的方法在 file 文件下创建出新的页
	BufferedOutputStream bufferedOutputStream=
	new BufferedOutputStream(new FileOutputStream(file,true));
	
	bufferedOutputStream.write(HeapPage.createEmptyPageData());
	bufferedOutputStream.close();
	
	然后通过 HeapPage heapPage=(HeapPage) Database.getBufferPool().getPage(tid,new HeapPageId(getId(),numPages()-1),Permissions.READ) ，得到新插入的页，再向该页插入元组，从而实现元组的插入。

public ArrayList<Page> deleteTuple(TransactionId tid, Tuple t)	实现从表中删除特定元组 思路： 首先通过传入的元组 t，通过 t.getRecordId().getPageId()得到 tuple 所在 page 的 pageId。然后再到缓冲池中通过 Database.getBufferPool().getPage(tid,pageId,Permissions.READ_WRITE);获取 tuple 所在的 heappage，然后调用 heappage.deleteTuple(t)方法就能实现从表中删除特定元组。

public void writePage(Page page)	实现将 page 保存的内容写入内存文件当中
	实现思路：
	通过 page.getId().getPageNumber()方法获取 page 在表中的序号。
	然后使用一个可以向内存文件随机位置进行读写的 RandomAccessFile 对象,初始化如下：
	RandomAccessFile randomAccessFile=new RandomAccessFile(getFile(),"rw");
	在借助 page 序号和 page 的大小计算出 page 在文件中位的偏移量，即为
	pageno*BufferPool.getPageSize();
	然后将文件读写对象跳转到偏移量位置，将 page 内容写入。
	randomAccessFile.seek(st);

	randomAccessFile.write(page.getPageData());

实现在缓冲池中插入、删除元组：

BufferPool：	
部分重要函数的实现	
public void insertTuple(TransactionId tid, int tableId, Tuple t)	实现了向一张表中插入特定元组的功能，并将插入过程访问的页的 dirty 标志置为 true。 实现思路： 通过目录获取表所在文件 Database.getCatalog().getDatabaseFile(tableId); 然后通过 dbfile.insertTuple(tid,t)方法向表中插入元组，并且得到返回的 ArrayList<Page>pages. 通过遍历数组，将插入过程更新的页的 dirty 标志置为 true。然后在缓冲池中保存下该页，pageConcurrentHashMap.put(pages.get(i).getId().hashCode(),pages.get(i));
public void deleteTuple(TransactionId tid, Tuple t)	实现从缓冲池中删除特定元组 t 的功能 实现思路： 通过 int tableId=t.getRecordId().getPageId().getTableId();得到元组 t 所在表的 tableId，然后借助 tableId 在数据库目录中找到表所在的 dbfile DbFile dbFile=Database.getCatalog().getDatabaseFile(tableId); 调用 dbfile.deleteTuple(tid,t)实现从表中删除元组 t，并得到函数的返回值为 ArrayList<Page>pages. 通过遍历数组，将删除过程中更新的页的 dirty 标志置为 true，然后在缓冲池中保存下该页， pageConcurrentHashMap.put(pages.get(i).getId().hashCode(),pages.get(i));

4. 实现插入 Insert 和删除 Delete 运算符

<table>
<tr><td colspan="2">Insert：
实现了将从子运算符读出的元组插入构造函数中指定的 tableId 对应的表。
维护的变量：
TransactionId transactionId;
OpIterator childOpIterator：子运算符，用于读出需要插入的元组
int tableId：插入的表的 Id
boolean called：记录插入方法是否被调用过</td></tr>
<tr><td colspan="2">构造函数：public Insert(TransactionId t, OpIterator child, int tableId)</td></tr>
<tr><td colspan="2">部分重要函数的实现</td></tr>
<tr><td>open、close 函数</td><td>因为 Insert 类是 Operator 的子类，open、close 需要调用父类的 open、close 函数，即 super.open()、super.close()，这样才能实现完整的打开关闭，不然可能出现空值的现象。</td></tr>
<tr><td>protected Tuple fetchNext()：</td><td>实现从子运算符读取元组插入到构造时指定的 tableId 中，并返回一个包含插入记录数的单字段元组。
实现思路：
如果 called 为 true 说明已经完成插入，直接返回一个空值结束函数。
遍历子运算符读取元组 tuple 并通过缓冲池插入，即 Database.getBufferPool().insertTuple(transactionId,tableId,t);
并且记录插入的记录数量 count。
然后构造
tupleDesc=new TupleDesc(new Type[]{Type.INT_TYPE},new String[]{"Inserted Records"});
Tuple tuple=new Tuple(tupleDesc);
tuple.setField(0,new IntField(count));
返回这个单字段元组。</td></tr>
</table>

<table>
<tr><td colspan="2">Delete：
实现删除操作符，从子运算符读取元组并从他们所属的表中删除。
TransactionId transactionId：事务 ID
OpIterator childOpIterator：子运算符，从中读取需要删除的元组
TupleDesc tupleDesc
boolean called：记录删除运算是否被调用过</td></tr>
<tr><td colspan="2">构造函数：public Delete(TransactionId t, OpIterator child)</td></tr>
<tr><td colspan="2">部分重要函数的实现</td></tr>
<tr><td>open、close 函数</td><td>因为 Delete 类是 Operator 的子类，open、close 需要调用父类的 open、close 函数，即 super.open()、super.close()，这样才能实现完整的打开关闭，不然可能出现空值的现象。</td></tr>
</table>

protected Tuple fetchNext()：	实现从子运算符读取元组并删除的功能 实现思路： 如果 called 为 true，说明删除操作已经进行过，则函数直接返回一个空值。 否则就遍历子运算符 childOpIterator，通过 childOpIterator.next()从中读取出元组，并借助缓冲池实现的删除操作：Database.getBufferPool().deleteTuple(transactionId,tuple) 实现对元组的删除。并记录下删除的元组的数量 count Tuple tuple=new Tuple(tupleDesc); tuple.setField(0,new IntField(count)); 最后函数返回这个单字段元组。

5. 实现从缓冲池进行页的驱逐。

BufferPool：	
部分重要函数的实现	
public synchronized void discardPage(PageId pid)	实现从缓冲池中删除特定页面 id 确定的页，并且保证不会被回滚： pageConcurrentHashMap.remove(pid.hashCode());
private synchronized void flushPage(PageId pid)	实现从内存中驱逐出 pageId 确定的页将其保存当磁盘 int hash=pid.hashCode(); Page page=pageConcurrentHashMap.get(hash); 首先在缓冲池中获取 pid 对应的页 如果 page.isDirty()返回的是空值，说明 page 已经被驱逐；否则我们需要对该 page 进行驱逐操作，通过 DateBase.getCatalog()从目录中调用 getDatabaseFile()方法，获取到 DbFile，调用 dbfile.writePage(page)将该页写入文件中，实现将 page 驱逐进入磁盘。最后将 page 的 dirty 标志置为 true。
private synchronized void evictPage(PageId pid)	实现页的驱逐，在缓存池中找到 dirty 标志为空的 page，再调用 discardPage(page.getId())实现从内存中删除这个 page。