南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

数据库系统实验报告

SimpleDB Lab2

王茂增

年级：2021级

专业：计算机科学与技术

指导教师：袁晓洁

2023 年 4 月 10 日

目录

一、 实验简介

本次实验的主要内容是接着上次Lab 1中实现的部分，完成Lab 2代码的编写,并通过单元测试来验证代码的正确性。下面列出了每一个Exercise实现的主要功能，具体的思路以及代码实现将在实验详解中介绍。

- **Exercise 1：**实现谓词、过滤、连接操作
- **Exercise 2：**实现聚合操作
- **Exercise 3：**在HeapPage和HeapFie类中实现插入和删除元组功能
- **Exercise 4：**实现Insert和Delete类
- **Exercise 5：**实现BufferPool的页面驱逐功能

二、 实验详解

（一） Exercise 1

1. 设计思路

在Exercise 1中，首先要实现Predicate类和JoinPredicate类，对输入数据判断是否满足指定的比较条件。然后要实现利用谓词对数据进行过滤或连接的类，即Filter类和Join类。过滤操作的fetchNext()方法会返回满足某条件的下一个元素，连接操作的fetchNext()方法会将下一个满足某条件的元组集合打包成一个元组并返回。

Predicate类存储着fieldIndex, op, operand三个成员变量，其最主要的filter()方法通过将输入元组的第fieldIndex个Field与operand进行op操作，以此判别输入元组是否满足该谓词。

JoinPredicate类存储着fieldIndex1, op, fieldIndex2三个成员变量，其最主要的filter()方法通过对输入元组的第fieldIndex1和第fieldIndex2个Tuple进行op操作，以此判别输入元组是否满足该谓词。

Filter类包含谓词predicate，迭代器child两个成员变量。其最主要的fetchNext()方法会在调用时不断调用迭代器的next()方法，直到找到一个满足谓词的元组并返回，否则返回null。

Join类主要包含谓词joinPredicate, 以及迭代器child1, child2。其最主要的fetchNext()方法需要根据两个迭代器的笛卡尔乘积，将下一个满足谓词joinPredicate的child1、child2中对应元素打包成一个元组并返回。具体实现方法将在重难点中介绍。

2. 重难点

1. **Predicate.filter()、JoinPredicate.filter()、Filter.fetchNext()方法**

   由于上述方法在设计思路部分已经对实现方法进行了详细的阐述，并且代码部分并没有太大的困难，所以在这里我不再给出具体的代码。

   值得注意的是，为了实现Filter和Join的fetchNext()方法，我新加入了一个方法求出所有满足条件元素构成的迭代器并存储在类中，这样在每次获取下一个结果时只需返回迭代器的下一元素即可，提高了next()和hasNext()方法的效率。

2. **简单嵌套循环连接(SNLJ)**

   该方法只是简单地以child1表为驱动表，遍历child2中的元素并进行谓词的判断，最后返回结果的迭代器。如果两者的元组个数分别为n_1、n_2，则时间复杂度为$O(n_1 * n_2)$。

简单嵌套循环连接(SNLJ)

```
ArrayList<Tuple> joinTuples = new ArrayList<>();
child1.rewind();
while (child1.hasNext()) {
        Tuple left = child1.next();
        child2.rewind();
        while (child2.hasNext()) {
                Tuple right = child2.next();
                if (joinPredicate.filter(left, right)) {
                        joinTuples.add(Tuple.merge(getTupleDesc(), left,
                            right));
                }
        }
}
return new TupleIterator(getTupleDesc(), joinTuples);
```

3. 块嵌套循环连接(BNLJ)

上述简单嵌套循环算法当child2的长度过大时会导致很大的磁盘I/O次数。具体来说，设child1和child2的元组个数分别为n_1、n_2，每个页面分别能存储k_1、k_2个对应元组，当缓冲池不能存储下整个child2时，简单嵌套循环算法需要$\lceil \frac{n_1}{k_1} \rceil$次I/O来读取child1中元素，还需要$n_1 * \lceil \frac{n_2}{k_2} \rceil$次I/O来读取child2中元素，由于硬盘读取时间远大于内存读取，这将导致极大的时间开销。

而块嵌套循环连接则将child1分块逐步缓存到JoinBuffer中，然后以child2作为驱动表，遍历child1中对应的一块，这样可以使内循环中的每个元组只需要一次磁盘I/O操作，极大提高了性能。具体来说，设JoinCache每次能缓存c个child1元组，则此时child2只需要$\lceil \frac{n_1}{c} \rceil * \lceil \frac{n_2}{k_2} \rceil$次磁盘I/O，与之前相比有了c倍的提升，而child1的磁盘I/O次数基本上不受影响。在实现块循环嵌套连接时，我选取了四个Page的大小作为joinBufferSize，即16384个字节。若child1元组大小为40字节，则child2的磁盘I/O数目将减少为原来的$\frac{1}{400}$，这是极大的性能提升。

下面，我给出了块嵌套循环连接算法的代码。该算法不断调用child1的next方法并将其缓存到cacheBlock中，如果缓存块已满，则以child2为驱动表，不断遍历cacheBlock中的元组，由于cacheBlock位于内存中，所以不会造成磁盘I/O，提高了访问效率。当child1里的元素遍历完成后，缓冲区中可能还有些元素并没有处理，所以需要通过另一段类似的代码来进行连接。

简单嵌套循环连接(SNLJ)

```
ArrayList<Tuple> joinTuples = new ArrayList<>();
int maxCacheTupleNum = joinBufferSize / child1.getTupleDesc().getSize();
Tuple[] cacheBlock = new Tuple[maxCacheTupleNum];
int cacheTupleNum = 0;
child1.rewind();

while (child1.hasNext()) {
        cacheBlock[cacheTupleNum++] = child1.next();
        if (cacheTupleNum >= maxCacheTupleNum) {
                //如果缓冲区满了，就按照简单嵌套循环连接的方法进行连接
                child2.rewind();
                while (child2.hasNext()) {
                        Tuple right = child2.next();
                        for (Tuple left : cacheBlock) {
                                if (joinPredicate.filter(left, right)) {
                                        joinTuples.add(Tuple.merge(
                                            getTupleDesc(), left, right));
                                }
                        }
                }
                cacheTupleNum = 0;//在逻辑上对缓存进行清空
        }
}

//处理缓冲区中剩下的元组
if (cacheTupleNum > 0) {
        child2.rewind();
        while (child2.hasNext()) {
                Tuple right = child2.next();
                for (Tuple left : cacheBlock) {
                        if (left == null) break;//元组为空说明缓存已经遍历完
                        if (joinPredicate.filter(left, right)) {
                                joinTuples.add(Tuple.merge(getTupleDesc(),
                                    left, right));
                        }
                }
        }
}

return new TupleIterator(getTupleDesc(), joinTuples);
```

（二） Exercise 2

1. 设计思路

在Exercise 2中，我们需要实现聚合操作，即根据某个Field对元组进行分组，并返回各组在某个Field上的和、数量、平均值、最大值、最小值等信息。我们首先需要实现整数和字符串的

聚合类，即IntegerAggregator类和StringAggregator类，然后根据这两个类实现Aggregator类。

IntegerAggregator类最主要的方法是mergeTupleIntoGroup()方法，该方法可以通过哈希表存储不同组别对应的答案，并根据需要执行的聚类操作对答案进行更新。值得注意的是，为了实现AVG操作，我还通过一个哈希表建立了一个组别与[个数，总和]数组的映射，因为需要通过总和和个数来计算平均值。如果不需要分组，则将组别设置为null，然后按照上述方法运行即可。

对于StringAggregator类，只需实现计算各组别个数的方法，具体方法与IntegerAggregator类相似，故不再介绍。

对于Aggregator类的实现，只需要根据该类构造函数中传入的元组迭代器的aFieldType选择使用IntegerAggregator类或者StringAggregator类即可。

由于代码实现较为简单，故在本报告中不再给出上述代码。

（三） Exercise 3

1. 设计思路

在Exercise 3中，需要在HeapPage类和HeapFile类中实现插入元组和删除元组的功能，并且需要将改变的表标记为dirty，以判断该页面被驱逐出缓冲区时是否需要写入磁盘。

对于HeapPage类，在实现deleteTuple(Tuple t)时，首先需要得到t在该页面中的tupleNo，然后判断该位置是否有这个元组，如果有则删去。为了实现insertTuple(Tuple t)方法，需要遍历表并找到空位置进行插入，具体实现方法见重难点部分。

对于HeapFile类，可以通过调用对应页面的删除操作来实现HeapFile对元组的删除。对于插入操作，需要遍历找到空缺页进行插入，如果没有空缺的页，则需再创建一个页并插入。

对于上述插入算法的实现，显然通过遍历找到空缺位置的方法并不高效。可以通过对每一个HeapPage和HeapFile都维护一个空值链表，记录未满的位置和页面，实现O(1)的插入。但是由于时间原因，在本次实验中本人并没有实现上述算法。

2. 重难点

1. **HeapPage.insertTuple(Tuple t)方法**

   在实现insertTuple()时，需要首先检测输入Tuple是否和该表的TupleDesc相符合，如果不符合则抛出异常。然后要顺序访问Page中的每个位置，判断该位置是否为空，如果为空则插入并将页面标记为dirty。可通过以字节为单位判断header是否为0x11111111来加快查找速度，实现更高的插入效率。如果最后没有找到空缺位置，则抛出异常。实现代码如下：

HeapPage.insertTuple(Tuple t)方法

```
if (!td.equals(t.getTupleDesc())) {
        throw new DbException();
}

int headerIndex = -1;
int numTuples = getNumTuples();
while (header[++headerIndex] == (byte) 0x11111111) ;

for (int i = 8 * headerIndex; i < numTuples; i++) {
        if (!isSlotUsed(i)) {
                t.setRecordId(new RecordId(pid, i));
                tuples[i] = t;
                markSlotUsed(i, true);
                return;
        }
}
throw new DbException();
```

（四） Exercise 4

1. 设计思路

在Exercise 4中需要实现Insert类和Delete类，他们对输入的迭代器执行插入或删除操作。

Insert类和Delete类的关键方法为Tuple fetchNext()方法。该方法通过对类的构造函数中输入的迭代器遍历调用BufferPool中的insertTuple和deleteTuple方法实现功能，并返回一个包含插入删除个数的单元素元组。fetchNext()方法只能调用一次，所以需要在类中维护一个called成员变量，当不是第一次调用时直接返回null。

由于上述方法的实现较为简单，故在本报告中不给出具体的实现代码。

（五） Exercise 5

1. 实现思路

在Exercise 5中需要实现BufferPool中的缓存驱逐机制，当缓存页面已满但还要读取新的页面时就要根据某种规则选择驱逐的页面。

缓存驱逐策略一般有三种，LRU、CLOCK和LFU。LRU（Least-Recently Used 最久未使用）通过双向链表和哈希表换出最久未使用的页面；CLOCK（时钟置换）通过给每个节点增加一个REF属性并使用clock指针将长时间未使用的页面换出。上述两种算法都有一个共同的局限性，就是当大型数据库执行scan、select *等操作时会访问非常多的数据，可能会把访问频次很高的页面换出。LFU（Least-Frequently Used 最不常用）通过两个双向链表+两个哈希表驱逐使用次数最少的页面。该算法也有其局限性，由于使用次数具有累加性质，新加入的页面很有可能就是下一个被换出的页面，这无疑是不公平的。

MySQL的缓存驱逐机制是在LRU的基础上加以改进形成的，根据“局部性原理”，通过“预读”异步读取还没访问的页，以减少磁盘IO的次数；另外，该方法还将双向链表划分为新生代和老生代，预读的页先加入老生代，如果在设定的等待时间后再次被访问就加入新生代。通过上述方法，即便采用不带where的select语句进行数据访问也不会将新生代中保存的访问频

次很高的页面换出，并且新加入的页面也会停留一段时间才被逐渐移到双向链表末尾并换出，解决了LRU、CLOCK和LFU的问题。结构示意图如图1所示：

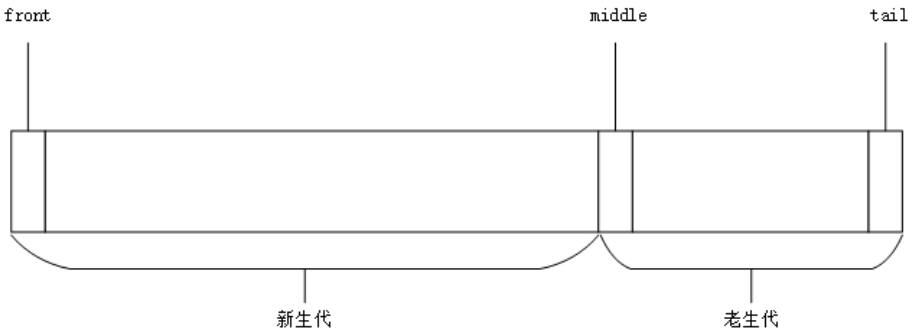

图 1: BufferPoolList结构示意图

通过上述分析，本人决定采用MySQL的缓存驱逐机制，但是由于实验还没要求实现数据库的线程安全部分，而MySQL中的预读机制需要异步读取，所以我暂时去掉了预读机制，实现了简化版的 MySQL缓存驱逐机制，预读机制将在后续实现数据库的线程安全之后加入其中。

2. 重难点

1. BufferPoolList类

为了实现上述缓存驱逐机制，我新建了一个名为BufferPoolList的双向链表类，链表的节点为BufferPoolListNode，存储着前向节点、后向节点、PageId以及缓存开始时间，该时间用于判断该节点是否应该移动到新生代。BufferPoolList类最主要的方法是push方法，该方法将页面插入到链表中，如果要驱逐节点，则会驱逐tail节点。

在push方法中，首先会判断页面是否已满，如果未满且已缓存该节点时则将节点移动到新生代头部，如果未缓存该节点则将其插入新生代头部。当页面第一次满时，会通过不断遍历next节点得到middle和tail。当页面已满时，如果已缓存该页面且达到等待时间则将其移动到新生代头部，如果未达到等待时间则移动到老生代头部；而如果页面已满且未缓存该页面，则将其插入到老生代头部。具体实现代码如下：

BufferPoolList.push(PageId pid)方法

```
if (size < MAX_SIZE) {//当页面未满时
        if (pid2node.containsKey(pid)) {//若存在该页面则移动到新生代头部
                moveNode2Front(pid2node.get(pid));
        } else {//否则在新生代头部插入一个新页面
                BufferPoolListNode newNode = new BufferPoolListNode(null,
                    front, pid);
                if (front != null) {
                        front.prev = newNode;
                }
                front = newNode;
                pid2node.put(pid, newNode);
                ++size;
        }
} else {
        //如果缓冲池刚满，则先计算得到旧生代头部节点和尾部节点
        if (!initial) {
                BufferPoolListNode tempNode = front;
                for (int i = 1; i <= newBlocksSize; i++) {
                        tempNode = tempNode.next;
                }
                middle = tempNode;
                for (int i = newBlocksSize + 1; i < MAX_SIZE; i++) {
                        tempNode = tempNode.next;
                }
                tail = tempNode;
                initial = true;
        }

        if (pid2node.containsKey(pid)) {//如果存在该页面
                BufferPoolListNode oldNode = pid2node.get(pid);
                if (System.currentTimeMillis() - oldNode.initialTime >=
                    OLD_BLOCK_TIME) {
                        moveNode2Front(oldNode);//等待时间已过则移动到新生代头部
                } else {//等待时间未过则移动到老生代头部
                        moveNode2Middle(oldNode);
                }
        } else {//如果不存在该页面，则插入节点到老生代头部
                BufferPoolListNode newNode = new BufferPoolListNode(middle.
                    prev, middle, pid);
                middle.prev.next = newNode;
                middle.prev = newNode;
                middle = newNode;
                pid2node.put(pid, newNode);
                ++size;
        }
}
```

三、 总结

本次实验中，我完成了SimpleDB的Lab 2部分。本次实验在Lab 1的基础上实现了用于表的插入、删除、选择、连接、聚合等操作的类和方法，同时还加入了缓冲池的驱逐机制。

对于表的连接操作，我注意到简单嵌套循环连接在表的数据量过大时会导致极大的磁盘I/O次数，所以我采用了块嵌套循环连接，大大降低了磁盘I/O次数；对于缓冲池的驱逐机制，由于常见的LRU、CLOCK、LFU方法都存在一些问题，所以我根据MySQL中的缓冲驱逐机制编写了BufferPoolList类，对缓冲池中的页面进行管理，解决了常见缓存驱逐算法存在的一些问题，提高了数据库的效率。

四、 提交记录

```
Author: mzwangg <mzwangg@gmail.com>
Date:   Sun Apr 9 21:01:09 2023 +0800

    Lab_2 Update

commit a68dcc14a717c4203b40a6d8f711ec1aedf9e174
Author: mzwangg <mzwangg@gmail.com>
Date:   Fri Apr 7 22:34:39 2023 +0800

    Lab_2 Exercise5

commit 95afe192d9988bc57873415b3c051944178620cd
Author: mzwangg <mzwangg@gmail.com>
Date:   Thu Apr 6 22:27:09 2023 +0800

    Lab_2 Exercise4

commit 7c189c1288c9eadc26a7efd01e9af4739cacfbfa
Author: mzwangg <mzwangg@gmail.com>
Date:   Wed Apr 5 23:57:24 2023 +0800

    Lab_2 Exercise2、3

commit a4995e5e9663b150ee866c2fc2480be3f7be6548
Author: mzwangg <mzwangg@gmail.com>
Date:   Tue Apr 4 23:58:19 2023 +0800

    Lab_2 Exercise2

commit 5ecb6ca06c7e02457Ofc193c501c8cd6959dbf8d
Author: mzwangg <mzwangg@gmail.com>
Date:   Mon Apr 3 20:25:35 2023 +0800

    Lab_2 Exercise1
```

图 2: 提交记录

五、 源代码

https://gitlab.com/mzwangg/SimpleDB