DM556 - Principle of Database Systems
Project 1: Buffer Manager

Jonas I. Sørensen <joso216>
Simon D. Jørgensen <simjo16>
Jeff Gyldenbrand <jegyl16>
Supervisor: Jacob A. Mikkelsen

July 20, 2018

Abstract

The goal of this project is to implement one of the bottom layers of a typical Database Management System (DBMS). The goal is to develop the buffer manager and its clock policy, frame descriptor and replacer such that a DBMS will run. The underlying disk manager and framework are already provided, so it's only the buffer manager to be developed. This buffer manager is to be implemented in the java programming language.

Contents

1 Overall status

1.1 Overview

We managed to implement all the required functions in BufMgr.java, Replacer.java and Clock.java. However, there was a single function in Clock.java that became redundant and therefore not made. In addition, we have removed some redundant code in BufMgr.java from an already provided function. This and other major components will be explained in greater details in section 1.2. Functions not mentioned here is elaborated as comments in the source code.

1.2 Details of implementation

No changes were made to the **BufMgr** function. The provided code initialises our buffer pool and frame table with a size determined by the number of pages given as argument. For efficient lookup of allocated pages, a hashmap is used.

In the **newPage** function a redundancy was discovered, where the provided code would deallocate the first page in a loop in the range of the number of new pages it tried to allocate. See code-table below. The loop was completely removed and the deallocate function is now invoked instead, to deallocate the page. Because the deallocate function points to the first page, and then deallocates it based on the number of its elements (run_size), it will have the same outcome.

```
1            for (int i = 0; i < run_size; i++) {
2                firstid.pid += 1;
3                Minibase.DiskManager.deallocate_page(firstid);
4            }
```

The goal of the **freePage** function is to deallocate a single page from the disk. This is archived by looking at a given page's pin count. If its pin count is greater than zero, it means that it is referred to by requesters, and therefore must not be removed. If the pin count is zero, thus have no references, the page is marked invalid and the replacer is notified. Lastly, the page is deallocated from memory.

The goal of the **pinPage** function is to pin a page meaning increment a page's pin count. In order to do this, the function first checks if the page is in main memory, and checks whether the page is already allocated, thus allocating it again would be redundant, in this case skipRead throws an exception. If the page is not already allocated the frame is copied to the output page, subsequently the frame's pin count is incremented. If the page is not in main memory, room has to be made for the new page, hence the **pickVictim** function is invoked. This functions looks through the buffer pool for a free page, if pickVictim returns nothing, it means no free page exists, hence all pages are pinned and the buffer is full. If a free page is found it will, if necessary, be released and written to disk. Once a free frame has been requisitioned, the page is written to it, be it from the disk or an input parameter.

To go a little more in depth with how the **pickVictim** function works: The function starts by going through the frame table array, and if a page has no reference we tag the page as free. It

goes through the table twice to catch pages that are changed from no_reference to free and therefor wasn't caught in the first run.

When a reference to a page no longer exists, the given page's pin counter has to be decremented. To do this, the function **unpinPage** is invoked. **unpinPage** will, if a pincount of zero is reached, free the page. Given that the page would be dirty, **unpinPage** also writes the page to disk. In any given case, **unpinPage** also notfies the replacer of the decrementation.

The **flushPage** function simply checks a given page if its dirty, and if so, writes it to the disk. The **flushAllPages** function invokes the flushPage function for each page in the buffer pool, which then writes all the pages with a dirty bit to the disk.

To get the count of unpinned buffer frames we invoke the **getNumUnpinned** function which converts the frame table array to a stream, then filters all the pages which has a reference, meaning all pages with a pin count greater than zero, and counts them.

2 File description

Of the four given pages, none were added, as these provided sufficient abstraction to achieve the goal of the project.

3 Division of Labor

We have primarily worked together on both the code-, test- and the report section. Jonas is the person in the group with most programming experience, and has primarily come up with how the code should be implemented, while Simon and Jeff have made suggestions for this, but the code is written together. The report is divided among us. Jeff has written the abstract. Jeff and Jonas have written section 1 whereas Simon has written sections 3 and 4. Then we have joined our sections and in unity read the complete report to fix small misunderstandings or errors to finally write the conclusion together.

4 Test Output

For testing purposes, a generalised test was done by using **Gradle**, to check whether the program did as it was intended to:

```
1    $ ./gradlew runBmTests
```

with the output seen in the *Appendix* in Figure 1.
There were 3 tests in total. The first test *allocates* and *write* on a bunch of pages, to then *read* them all, and lastly *frees* up the pages.
The second test checks for illegal buffer manager operations, such as: pinning more pages than frames, freeing a page that's already pinned more than once, and lastly, attempting to unping a page that's not in the bufferpool.
The third test simply allocated and edited new pages and left some pinned, lastly it read the pages.

A check for special corner-cases were made, like looking up **TABLES** that do not exist.

We want to know whether or not, the **TABLE** do exists, and avoid crashing if that's the case. This can be seen in Figure 3.

Using the examples from **sample MiniSQL queries.txt**, testing whether the **BufMgr** works correctly could be monitored, using *queries* such as:

```
1    SELECT name, depname FROM Emp, Works, Dept WHERE id = eid AND depid = did;
```

Resulting in a table with information of the rows and columns, as seen in Figure 5, if and only if said **TABLES** do exist.
Furthermore, in the same Figure, we show how **STATS** provide information over: *reads*, *writes*, *allocs* and *pinned*. This information proved quite efficient when checking whether the functions had been implemented correctly.

5 Conclusion

It has been a success to implement the lower layer in a database, more specifically called the buffer manager. All requirements have been met: both the buffer manager, the clock policy and the replacement work. This statement is supported by the test's output.

6 Appendix

```
jegyl16@imada-106311:~/Desktop/fourth_semester/DM556-DB/assignments/project1$ ./gradlew runBmTests
:compileJava UP-TO-DATE
:processResources NO-SOURCE
:classes UP-TO-DATE
:runBmTests
Creating database...
Replacer: Clock

Running buffer manager tests...

  Test 1 does a simple test of normal buffer manager operations:
  - Allocate a bunch of new pages
  - Write something on each one
  - Read that something back from each one
   (because we're buffering, this is where most of the writes happen)
  - Free the pages again
  Test 1 completed successfully.

  Test 2 exercises some illegal buffer manager operations:
  - Try to pin more pages than there are frames
  --> Failed as expected

  - Try to free a doubly-pinned page
  --> Failed as expected

  - Try to unpin a page not in the buffer pool
  --> Failed as expected

  Test 2 completed successfully.

  Test 3 exercises some of the internals of the buffer manager
  - Allocate and dirty some new pages, one at a time, and leave some pinned
  - Read the pages
  Test 3 completed successfully.

All buffer manager tests completed successfully!

BUILD SUCCESSFUL

Total time: 1.607 secs
```

Figure 1: ./gradlew runBmTest

```
jegyl16@imada-106311:~/Desktop/fourth_semester/DM556-DB/assignments/project1$ ./gradlew -q run
Minibase SQL Utility 1.0
Loading database...

MSQL>
```

Figure 2: ./gradlew -q run

```
jegyl16@imada-106311:~/Desktop/fourth_semester/DM556-DB/assignments/project1$ ./gradlew -q run
Minibase SQL Utility 1.0
Loading database...

MSQL> SELECT * FROM Emp;

ERROR: table 'Emp' doesn't exist

MSQL> SELECT name,depname FROM Emp, Works,Dept WHERE id = eid AND depid = did;

ERROR: table 'Emp' doesn't exist

MSQL> STATS;

reads  = 2
writes = 0
allocs = 0
pinned = 0

MSQL>
ERROR: Encountered ";" at line 3, column 6.
Was expecting one of:
    "CREATE" ...
    "DELETE" ...
    "DESCRIBE" ...
    "DROP" ...
    "EXPLAIN" ...
    "HELP" ...
    "INSERT" ...
    "QUIT" ...
    "SELECT" ...
    "STATS" ...
    "UPDATE" ...

MSQL> DROP TABLE Emp;

ERROR: table 'Emp' doesn't exist

MSQL> QUIT;

Closing database...
jegyl16@imada-106311:~/Desktop/fourth_semester/DM556-DB/assignments/project1$
```

Figure 3: Checking for cornercases

```
jegyl16@imada-106311:~/Desktop/fourth_semester/DM556-DB/assignments/project1$ ./gradlew -q run
Minibase SQL Utility 1.0
Loading database...

MSQL> CREATE TABLE Emp (name STRING(50), id INTEGER, age INTEGER);

Table created.

MSQL> CREATE TABLE Works (eid INTEGER, depid INTEGER);

Table created.

MSQL> CREATE TABLE Dept (did INTEGER, budget INTEGER, depname STRING(50));

Table created.

MSQL> INSERT INTO Emp VALUES ('Yongluan', 1 , 28);

1 row affected.

MSQL> INSERT INTO Emp VALUES ('Jacob', 2 , 32);

1 row affected.

MSQL> INSERT INTO Emp VALUES ('Claus', 3 , 42);

1 row affected.

MSQL> INSERT INTO Works VALUES (1, 1);

1 row affected.

MSQL> INSERT INTO Works VALUES (1, 2);

1 row affected.

MSQL> INSERT INTO Works VALUES (2, 1);

1 row affected.

MSQL> INSERT INTO Works VALUES (3, 2);

1 row affected.

MSQL> INSERT INTO Dept VALUES (1, 42 , 'IMADA');

1 row affected.

MSQL> INSERT INTO Dept VALUES (2, 2000 , 'ADMINISTRATION');

1 row affected.

MSQL>
```

Figure 4: Creating and inserting DB

```
MSQL> SELECT * FROM Emp;

1 row affected.

MSQL>
name                                                    id        age
-------------------------------------------------------------------
Yongluan                                                1         28
Jacob                                                   2         32
Claus                                                   3         42

3 rows affected.

MSQL> SELECT name,depname FROM Emp, Works,Dept WHERE id = eid AND depid = did;

name                                                    depname
-------------------------------------------------------------------------------------------------------
Yongluan                                                IMADA
Yongluan                                                ADMINISTRATION
Jacob                                                   IMADA
Claus                                                   ADMINISTRATION

4 rows affected.

MSQL> SELECT name FROM Emp, Works,Dept WHERE id = eid AND depid = did AND budget > 100;

name
-------------------------------------------------
Yongluan
Claus

2 rows affected.

MSQL> STATS

reads  = 6
writes = 1
allocs = 7
pinned = 0

MSQL> DROP TABLE Emp;

Table dropped.

MSQL> STATS

reads  = 0
writes = 6
allocs = -2
pinned = 0
```

Figure 5: Query the database

6.1 Source Code:

6.1.1 BufMgr

```
1  package bufmgr;
2
3  import java.util.HashMap;
4
5  import global.GlobalConst;
6  import global.Minibase;
7  import global.Page;
8  import global.PageId;
9  import java.util.Arrays;
10
11 /**
12  * <h3>Minibase Buffer Manager</h3> The buffer manager reads disk pages into a
13  * main memory page as needed. The collection of main memory pages (called
14  * frames) used by the buffer manager for this purpose is called the buffer
15  * pool. This is just an array of Page objects. The buffer manager is used by
16  * access methods, heap files, and relational operators to read, write,
17  * allocate, and de-allocate pages.
18  */
19 @SuppressWarnings("unused")
20 public class BufMgr implements GlobalConst {
21
22        /** Actual pool of pages (can be viewed as an array of byte arrays). */
23        protected Page[] bufpool;
24
25        /** Array of descriptors, each containing the pin count, dirty status, etc.
              */
26        protected FrameDesc[] frametab;
27
28        /** Maps current page numbers to frames; used for efficient lookups. */
29        protected HashMap<Integer, FrameDesc> pagemap;
30
31        /** The replacement policy to use. */
32        protected Replacer replacer;
33
34        /**
35         * Constructs a buffer manager with the given settings.
36         *
37         * @param numbufs: number of pages in the buffer pool
38         */
39
40         /*
41         *     Here we initialize our buffermanager with the variable numbufs which
              determines
42         *     the size of the pool, and initialize the frame table in the arrays.
43         *     We initialize our HashMap for efficient lookup and our replace policy
              as the clock policy.
44         */
45        public BufMgr(int numbufs) {
46           // initialize the buffer pool and frame table
47           bufpool = new Page[numbufs];
```

```
48            frametab = new FrameDesc[numbufs];
49            for (int i = 0; i < numbufs; i++) {
50              bufpool[i] = new Page();
51              frametab[i] = new FrameDesc(i);
52            }
53
54            // initialize the specialized page map and replacer
55            pagemap = new HashMap<Integer, FrameDesc>(numbufs);
56            replacer = new Clock(this);
57        }
58
59        /**
60         * Allocates a set of new pages, and pins the first one in an appropriate
61         * frame in the buffer pool.
62         *
63         * @param firstpg
64         *          holds the contents of the first page
65         * @param run_size
66         *          number of new pages to allocate
67         * @return page id of the first new page
68         * @throws IllegalArgumentException
69         *           if PIN_MEMCPY and the page is pinned
70         * @throws IllegalStateException
71         *           if all pages are pinned (i.e. pool exceeded)
72         */
73
74         /*
75         *     ('Minibase.DiskManager.deallocate_page(firstid,run_size)': is only
               code we wrote)
76         *     Removed the for-loop because it was made redundant by the
               deallocate_page-function.
77         *     If we somehow cant pin the page, we just call the function
               deallocate_page which points to our
78         *     newly allocaled memory (first page) and then deallocate it based on
               the number of elements (run_size)
79         *
80         *     If we succeed in pinning the page(s) we tell the replacer that it has
               got a new page.
81         */
82        public PageId newPage(Page firstpg, int run_size) {
83               // allocate the run
84               PageId firstid = Minibase.DiskManager.allocate_page(run_size);
85
86               // try to pin the first page
87               try {
88                      pinPage(firstid, firstpg, PIN_MEMCPY);
89               } catch (RuntimeException exc) {
90                 Minibase.DiskManager.deallocate_page(firstid,run_size);
91                 throw exc;
92               }
93               // notify the replacer and return the first new page id
94               replacer.newPage(pagemap.get(firstid.pid));
95               return firstid;
96        }
```

```
97
98          /**
99           * Deallocates a single page from disk, freeing it from the pool if needed.
100          * Call Minibase.DiskManager.deallocate_page(pageno) to deallocate the page
                 before return.
101          *
102          * @param pageno
103          *           identifies the page to remove
104          * @throws IllegalArgumentException
105          *            if the page is pinnedreplacer
106          */
107
108          /*
109          *     We want to free a page whenever it has no reference because it is no
                 longer
110          *     in use and a waste of space.
111          *
112          *     If a page has references we cant remove it and throw an exception.
113          *     Else we remove it and set the page to INVALID, and tell it to the
                 replacer.
114          *     Lastly we deallocate the page.
115          */
116         public void freePage(PageId pageno) throws IllegalArgumentException {
117                 final FrameDesc fd = pagemap.get(pageno.pid);
118
119                 if(null == fd) return;
120                 if(0 < fd.pincnt) throw new IllegalArgumentException("Page(" + pageno.
                        pid + ") is pinned, can not be removed.");
121
122                 fd.pageno.pid = INVALID_PAGEID;
123                 pagemap.remove(pageno.pid);
124                 replacer.freePage(fd);
125
126                 Minibase.DiskManager.deallocate_page(pageno);
127         }
128
129         /**
130          * Pins a disk page into the buffer pool. If the page is already pinned,
131          * this simply increments the pin count. Otherwise, this selects another
132          * page in the pool to replace, flushing the replaced page to disk if
133          * it is dirty.
134          *
135          * (If one needs to copy the page from the memory instead of reading from
136          * the disk, one should set skipRead to PIN_MEMCPY. In this case, the page
137          * shouldn't be in the buffer pool. Throw an IllegalArgumentException if so.
                 )
138          *
139          *
140          * @param pageno
141          *           identifies the page to pin
142          * @param page
143          *           if skipread == PIN_MEMCPY, works as as an input param, holding
                 the contents to be read into the buffer pool
```

```
144         *           if skipread == PIN_DISKIO, works as an output param, holding the
                contents of the pinned page read from the disk
145         * @param skipRead
146         *           PIN_MEMCPY(true) (copy the input page to the buffer pool);
                PIN_DISKIO(false) (read the page from disk)
147         * @throws IllegalArgumentException
148         *            if PIN_MEMCPY and the page is pinned
149         * @throws IllegalStateException
150         *            if all pages are pinned (i.e. pool exceeded)
151         */
152
153         /*
154         *The pinpage function pins a page, given as argument a page and an id, it
                furthermore requires an enum(skipread)
155         * to dictate whether or not it should pin the id or create a new page.
156         */
157         public void pinPage(PageId pageno, Page page, boolean skipRead) {
158                // Check if page is in main memory.
159                if( pagemap.containsKey(pageno.pid) ){
160                       // If the page is already allocated, then allocating it again
                           is redundant, thus skipread throws an argument.
161                       if(skipRead) throw new IllegalArgumentException( "Page(" +
                           pageno.pid + ") PIN_MEMCPY and the page is pinned" );
162
163                       // Copy frame to out-pgae and increment frame pincount.
164                       final FrameDesc fd = pagemap.get( pageno.pid );
165
166                       page.setPage(bufpool[fd.index]);
167                       increment(fd);
168                // If the page is not in main memory, room has to be made for the new
                       page.
169                } else {
170                       // Search for a victim, aka an invalid page.
171                       final int index = replacer.pickVictim();
172
173                       // If the replacer found no victims, there is no room for
                           another page.
174                       if( EMPTY_SLOT == index ) throw new IllegalStateException("All
                            pages are pinned");
175
176                       final FrameDesc fd = frametab[ index ];
177
178                       // If the selected page is valid, it must be removed from main
                            memory and if nessecery(dirty) written to disk.
179                       if( INVALID_PAGEID != fd.pageno.pid ){
180                              pagemap.remove( fd.pageno.pid );
181                              if( fd.dirty ) Minibase.DiskManager.write_page( fd.
                                   pageno, bufpool[ index ]);
182                       }
183                       // If PIN_MEMCPY, copy from the page to the buffer. Else the
                           the page form the disk into the buffer.
184                       if( skipRead ) bufpool[ index ].copyPage( page );
185                       else Minibase.DiskManager.read_page( pageno, bufpool[ index ])
                           ;
```

```
186
187                     page.setPage( bufpool[ index ]);
188                     new_page( fd, pageno );
189             }
190         }
191
192             private void increment (final FrameDesc fd){
193                     fd.pincnt++;
194                     replacer.pinPage(fd);
195             }
196
197             private void new_page(final FrameDesc fd, final PageId pageno){
198                     fd.pincnt = 1;
199                     pagemap.put( pageno.pid, fd );
200                     fd.pageno.pid = pageno.pid;
201                     replacer.pinPage( fd );
202             }
203
204         /**
205          * Unpins a disk page from the buffer pool, decreasing its pin count.
206          *
207          * @param pageno
208          *           identifies the page to unpin
209          * @param dirty
210          *           UNPIN_DIRTY if the page was modified, UNPIN_CLEAN otherrwise
211          * @throws IllegalArgumentException
212          *            if the page is not present or not pinned
213          */
214
215         /*
216          *     We want to remove a reference from a page, by decrementing the pin
                count
217          *     and if the page has been altered we set its status to dirty.
218          *     Lastly we tell it to the replacer.
219          */
220         public void unpinPage(PageId pageno, boolean dirty) throws
                IllegalArgumentException {
221             final FrameDesc fd = pagemap.get(pageno.pid);
222
223             if (null == fd) throw new IllegalArgumentException("Page is not
                    present");
224             if (0 < fd.pincnt){
225                     fd.pincnt--;
226                     fd.dirty = dirty;
227                     replacer.unpinPage(fd);
228             }
229         }
230
231         /**
232          * Immediately writes a page in the buffer pool to disk, if dirty.
233          */
234
235         public void flushPage(PageId pageno) {
236             if (pagemap.get(pageno.pid).dirty)
```

```
237                 Minibase.DiskManager.write_page(pageno, bufpool[pagemap.get(pageno.pid
                        ).index]);
238         }
239
240         /**
241          * Immediately writes all dirty pages in the buffer pool to disk.
242          */
243
244         /*
245         *       With a simple lambda we check each keys if their values are dirty, if
                    so,
246         * we flush them.
247         */
248         public void flushAllPages() {
249                 pagemap.forEach( (k,v) -> flushPage(v.pageno));
250         }
251
252         /**
253          * Gets the total number of buffer frames.
254          */
255         public int getNumBuffers() {
256                 return bufpool.length;
257         }
258
259         /**
260          * Gets the total number of unpinned buffer frames.
261          */
262
263         /*
264         *       We convert the frame table array to a stream, then we filter all pages
265         * which has a reference (!= zero) and counts them.
266         */
267         public int getNumUnpinned() {
268                 return (int)Arrays.stream(frametab).filter(i-> 0 == i.pincnt).count();
269         }
270
271 } // public class BufMgr implements GlobalConst
```

6.1.2 Clock

```
1  package bufmgr;
2
3  public class Clock extends Replacer{
4
5         // TAGS for each state of a page
6         protected static final int free = 1;
7         protected static final int no_reference = 2;
8         protected static final int pinned = 3;
9
10        // a pointer to keep track of location in the frame table.
11        protected int pointer;
12
13        /*
14        *       We start by initializing the buffermanager and then set all the frames
15                to the tag free.
16        */
17        protected Clock(BufMgr bufmgr) {
18                super(bufmgr);
19                for (int i = 0; i < frametab.length; i++) {
20      frametab[i].state = free;
21     }
22         }
23
24        @Override
25        public void newPage(FrameDesc fdesc) {
26                // There is no need for this function because
27                // we evaluate the need of a new page in the buffermanager.
28        }
29
30        @Override
31        public void freePage(FrameDesc fdesc) {
32                fdesc.state = free;
33        }
34
35        @Override
36        public void pinPage(FrameDesc fdesc) {
37                fdesc.state = pinned;
38        }
39
40        @Override
41        public void unpinPage(FrameDesc fdesc) {
42                if (0 == fdesc.pincnt)
43                fdesc.state = no_reference;
44        }
45
46        /*
47        * We want to pick a victim that is free. Meaning look through the
48        * frame table for a free page. If a page has no reference we set its tag to
49        *       free. We go through the frame table once more to catch the pages which
              are
50        *       changed from no_reference to free.
51        */
```

```
52          @Override
53          public int pickVictim() {
54                  for ( int i = 0 ; i < frametab.length << 1; i++ ) {
55
56                          final FrameDesc fd = frametab[pointer];
57
58                          if(free == fd.state){
59                                  return pointer;
60                          }
61                          if(no_reference == fd.state){
62                                  fd.state = free;
63                          }
64                          pointer = (pointer+1) % frametab.length;
65                  }
66                  return -1;
67          }
68  }
```

6.1.3 Replacer

```
1  package bufmgr;
2
3  import global.GlobalConst;
4
5  /**
6   * Base class for buffer pool replacement policies.
7   */
8  abstract class Replacer implements GlobalConst {
9
10   /** Reference back to the buffer manager's frame table. */
11   protected FrameDesc[] frametab;
12
13   // ------------------------------------------------------------------------
14
15   /**
16    * Constructs the replacer, given the buffer manager.
17    */
18   protected Replacer(BufMgr bufmgr) {
19     this.frametab = bufmgr.frametab;
20   }
21
22   /**
23    * Notifies the replacer of a new page.
24    */
25   public abstract void newPage(FrameDesc fdesc);
26
27   /**
28    * Notifies the replacer of a free page.
29    */
30   public abstract void freePage(FrameDesc fdesc);
31
32   /**
33    * Notifies the replacer of a pined page.
34    */
35   public abstract void pinPage(FrameDesc fdesc);
36
37   /**
38    * Notifies the replacer of an unpinned page.
39    */
40   public abstract void unpinPage(FrameDesc fdesc);
41
42   /**
43    * Selects the best frame to use for pinning a new page.
44    *
45    * @return victim frame number, or -1 if none available
46    */
47   public abstract int pickVictim();
48
49 } // abstract class Replacer implements GlobalConst
```