Java Minibase Project LAB 2

Introduction

The main objective of this lab is to create differents replacer, such as LRU, FIFO, LIFO and LRU-k methods
We use the minibase project, which is a database management system intended for educational use.

- This project is used as part of the DBSys course, of the Data Science branch of EURECOM
- Contributors : Group C : **Julien THOMAS (julien.thomas@eurecom.fr) and Eliot CALIMEZ (eliot.calimez@eurecom.fr)**
- Repository : https://github.com/korrigans84/DBMS-java-minibase

LRU

In this method, we always update the buffer when we pin a page. For example, if page p was consulted 20 pages ago, and there are 10 pages, and our buffer is of size 20, then page p will not be the victim, because it was used more recently. In the FIFO method, page p would have been the victim (if not pinned). This method was already inplemented here

FIFO

The difference with the LRU method is in the pin method. We update the buffer only if the frame is a new frame :

```
if(!Arrays.stream(this.frames).anyMatch(x -> x == frameNo) )
    update(frameNo);
```

You can access the class here

LIFO

Differences with FIFO method

The only difference is when we pick a victim.
Instead of retrieving the first unpinned page from the buffer, we retrieve the one which was last inserted in the buffer, and which is not pinned.

```
for ( int i = numBuffers - 1; i >=0  ; --i ) {
     frame = frames[i];
    if ( state_bit[frame].state != Pinned ) {
    state_bit[frame].state = Pinned;
    (mgr.frameTable())[frame].pin();
    update(frame);
    return frame;
    }
}
```

You can access the class here

LRUK

Structure:

LRUK object contains 4 variables: - k contain the K value for the algorithm
- frames in an array which is the buffer pool. This length is defined by mgr.getNumBuffers()
- nframes is the number of frames used, between 0 and the length of frames array
- histories contains the histories of the pages already passed in the buffer

HIST object contains the k last references for one frame in the buffer.
References are the timestamp of the moment of the page is pinned

How it works?

PIN method

When we pin a new frame, we update the buffer pool, and the history of its frame. If the page already have an history, we just update this one. If not, the private method getHistOfAPage throws an InvalidFrameNumberException, and we create an new history for this new frame. We add this history to the histories buffer, and don't forget to add a reference to this new frame history

Pick a victim method

To pick a victim, we need to find the oldest reference among frames that aren't pinned. We browse all the histories for each frames, and keep the oldest reference. If all the pages are pinned, we throw a BufferPoolExceededException

You can access the class here

Correlated Reference implementation

To prevent unpinning page from the same transaction, we implement the correlated delay. To implement it, we just define a delay for correlated referces in HIST object :

```
    /**
     * The period in millisecond, where whe consider that references are correlated
     */
    private int CORRELATED_REFERENCE_PERIOD = 100;
```

Next, we need a method to say at pick_victim method of LRU object if a page is correlated or not

```
    public boolean isNotCorrelated(long ref)
    {
        return(ref - getLast() > CORRELATED_REFERENCE_PERIOD);
    }
```

And we update the pick_victim method :

```
            if ( state_bit[frame].state != Pinned && hist.getOldestReference() <= min && hist.isNotCorrelated(System.currentTimeMil
```

Then, when we update the history of a page, we verify if the last reference is correlated, and in this case, we replace this last reference with the new one. In `addReference` method of HIST, we add :

```
if(references.size()> 0  && ref - references.get(references.size()-1) < CORRELATED_REFERENCE_PERIOD) {
            references.set(references.size()-1, ref);
            return;
        }
```

This implementation is visible on the branch correlated_reference_implementation of this repository.

TIPS

> the file Code/copyCode.sh is a script file to update *.java files in Code folder with the class used in src/bufmgr

Problems

EDIT : With the new Test4, there is no problem anymore.

- When we execute the test script, we have an error in the console during the Test 4, which does not depend of our implemented code :

```
bufmgr.HashEntryNotFoundException: BUFMGR: HASH_NOT_FOUND.
    at bufmgr.BufMgr.unpinPage(BufMgr.java:617)
    at tests.BMDriver.test4(BMTest2020.java:690)
    at tests.BMDriver.runAllTests(BMTest2020.java:117)
    at tests.BMDriver.runTests(BMTest2020.java:88)
    at tests.BMTest2020.main(BMTest2020.java:786)
```

We have tried to debug this error, and we have find that the error occurs in the `lookup` method of Bufmgr class, where `ht[hash(pageNo)]` is always null.