CSC540 – Programming Team Project 2
Systems Project - Buffer & Recovery Management
Due by Nov 29th, 2016, 11.59p.m.

This project covers two main features of databases: Buffer Management and Recovery. In short, you will be making changes to the Buffer Management and Recovery Algorithms currently supported in SimpleDB. Your work will concentrate on the *simpledb.file*, *simpledb.buffer* and *simpledb.tx.recovery* packages. In particular, you should begin by taking a close look at the following classes *File*, *Page*, *FileMgr*, *Buffer*, *BufferMgr*, *LogRecord*, *LogManager* and *LogIterator*. As background for SimpleDB, it might be helpful to begin by reading the publication on SimpleDB and reviewing the slides in on SimpleDB that are on Moodle will also be helpful background.
Currently, we have covered the topic of Buffer Management which is one part of the project. Next lecture we will begin transaction management and recovery which will help you understand the description of the second part of the project.

You may continue with the same teams as before unless there is a major reason to change teams. **But note that, you will only be able to change teams if there are others interested in forming a new team.**

Part A: Buffer Management Background

A buffer pool contains a set of buffer and each buffer can hold a page. Data is stored on disk as blocks and after read into main memory are represented as pages. Reading and writing pages from main memory to disk is an important task of a database system. Main memory is partitioned into collections of pages. Data is stored on disk as blocks and after being read into main memory are represented as pages. Each page can be hold in a buffer and the collection of buffers is called the buffer pool. The buffer manager is responsible for bringing blocks from disk to the buffer pool when they are needed and writing blocks back to the disk when they have been updated. The buffer manager keeps a pin count and dirty flag for each buffer in the buffer pool. The pin count records the number of times a pagehas been requested but not released, and the dirty flag records whether the pagehas been updated or not. As the buffer pool fills, some pages may need to be removed in order to make room for new pages. The buffer manager uses a replacement policy to choose pages to be flushed from the buffer pool. The strategy used can greatly affect the performance of the system. LRU (least recently used), MRU (most recently used) and Clock are different policies that are appropriate to use under different conditions.

SimpleDB Buffer Manager (from Dr.EdwardSciore's notes)

The SimpleDB buffer manager is grossly inefficient in two ways:

- When looking for a buffer to replace, it uses the first unpinned buffer it finds, instead of using some intelligent replacement policy.
- When checking to see if a block is already in a buffer, it does a sequential scan of the buffers, instead of keeping a data structure (such as a map) to more quickly locate the buffer.

Project Tasks Description

Below is a brief description of the tasks that you are required to implement and some additional guidelines that will help you with the implementation:

- **Task 1: Use a data structure to keep track of the buffer pool for more efficient searching**
  This structure will track allocated buffers, keyed on the block they contain. (A buffer is allocated when its contents is not null, and may be pinned or unpinned. A buffer starts out unallocated; it becomes

allocated when it is first assigned to a block, and stays allocated forever after.) Use this map to determine if a block is currently in a buffer. When a buffer is replaced, you must update the data structure -- The mapping for the old block must be removed, and the mapping for the new block must be added. For our convenience, we will be using "**bufferPoolMap**" as the name of the structure.

- **Task 2: First In First Out (FIFO) Buffer Replacement Policy**
  This suggests a page replacement strategy that chooses the page that was least recently replaced i.e. the page that has been sitting in the buffer pool the longest. This differs a little from the least recently used in that FIFO considers when the page was *added* to the pool while LRU considers when the page was *last accessed*.

Part B: Recovery Management Background

The SimpleDB recovery manager is implemented via the package *simpledb.tx.recovery* and in particular the class *RecoveryMgr*. Each transaction creates its own recovery manager which has methods to write the appropriate log records for that transaction. For example, the constructor writes a start log record, the commit and rollback methods write corresponding log records, and setInt and setString extract the old value from the specified buffer and write and update record to the log.

The code for the SimpleDB recovery manager can be divided into three major areas: (i) code to implement the different kinds of log records (implemented by different classes that implement the LogRecord interface), (ii) code to iterate to the log file (*LogIterator*) and (iii) code to implement the rollback and recovery algorithms. In this part of the project, you will be working primarily with the *simpledb.tx.Recovery*. The current implementation of SimpleDB only implements the Undo recovery algorithm. Your task is to modify it to do the Undo-Redo algorithm.

The outline of the undo-redo algorithm is as follows:

//The undo stage

1. For each log record, (reading backwards from the end)"
   a. If the current record is a commit record then: Add that transaction to the commit list
   b. If the current record is a rollback record then: Add that transaction to the rollback list
   c. If the current record is an update record and that transaction is not on the commit or rollback list then: Restore old value at the specified location.

// The Redo Stage:

2. For each log record (reading forwards from the beginning)
   If the current record is an update record and that transaction is on the committed list, then: Restore the new value at the specified location

Below is an outline of the key things that need to be done:

- First, the log manager needs to be modified so that it is possible to read the log forward, starting from any given log record (as well as from the beginning of the log). Currently, the records in a log block are chained backwards. They need to be chained forwards as well.

- Second, log records need to be modified so that they contain both the before and after values of the modified location.
- Third, the method *doRecover* in *RecoveryMgr* needs to be modified to support the redo stage of algorithm.

Tentative Grading Scheme

Code compiles and sample client program runs – 20%
Task 1: Basic working of Buffer Pool and Replacement Policy – 40%
Task 2 – Undo-Redo Recovery 40%

APPENDIX: General Testing guidelines

Start thinking about how to unit test your code (i.e. develop test programs for the different components separately). Try to thoroughly test your code and think about corner cases. Below we provide some guidance and some test scenarios that will be used for grading. **However, do not rely solely only on these because not all test cases are not captured here**.

For Buffer Management: We will be evaluating the proper implementation of the two tasks.

- Basic Buffer manager functionality – pin / unpin / pinNew.
- Test for appropriate changes in availability of buffers after pinning.
- Are all buffers allocated before re-allocating any?
- Pin more blocks than the number of buffers available (Expected: BufferAbortException)
- Test if the contents of the Buffer pool Map is up to date after pinning / unpinning
- Basic FIFO algorithm - Pin / unpin blocks in a random order to test MRM.

Below, we provide some methods that we will use to test your implementation of the Map for the Buffer pool. Copy them into the respective java files. For our convenience, we have used "**bufferPoolMap**" as the name of the data structure.

Code snippets

BasicBufferMgr.java

```
/**
* Determines whether the map has a mapping from
* the block to some buffer.
* @paramblk the block to use as a key
* @return true if there is a mapping; false otherwise
*/
booleancontainsMapping(Block blk) {
returnbufferPoolMap.containsKey(blk);
}
/**
* Returns the buffer that the map maps the specified block to.
* @paramblk the block to use as a key
* @return the buffer mapped to if there is a mapping; null otherwise
*/
Buffer getMapping(Block blk) {
returnbufferPoolMap.get(blk);
}
```

BufferMgr.java

```
/**
* Determines whether the map has a mapping from
* the block to some buffer.
* @paramblk the block to use as a key
* @return true if there is a mapping; false otherwise
*/
publicbooleancontainsMapping(Block blk) {
returnbufferMgr.containsMapping(blk);
}
/**
* Returns the buffer that the map maps the specified block to.
* @paramblk the block to use as a key
* @return the buffer mapped to if there is a mapping; null otherwise
*/
public Buffer getMapping(Block blk) {
returnbufferMgr.getMapping(blk);
}
```

For Recovery Management: We will be evaluating the three main tasks associated with implementing Undo-Redo algorithm. Some additional guidelines and code snippets similar to that given for Buffer Management may be given later.