6.830 Lab 6: Rollback and Recovery

Assigned: Wednesday, November 26, 2014

Due: Wednesday, December 10, 2014 11:59 PM EST

0. Introduction

In this lab you will implement log-based rollback for aborts and log-based crash recovery. We supply you with the code that defines the log format and appends records to a log file at appropriate times during transactions. You will implement rollback and recovery using the contents of the log file.

The logging code we provide generates records intended for physical whole-page undo and redo. When a page is first read in, our code remembers the original content of the page as a before-image. When a transaction updates a page, the corresponding log record contains that remembered before-image as well as the content of the page after modification as an after-image. You'll use the before-image to roll back during aborts and to undo loser transactions during recovery, and the after-image to redo winners during recovery.

We are able to get away with doing whole-page physical UNDO (while ARIES must do logical UNDO) because we are doing page level locking and because we have no indices which may have a different structure at UNDO time than when the log was initially written. The reason page-level locking simplifies things is that if a transaction modified a page, it must have had an exclusive lock on it, which means no other transaction was concurrently modifying it, so we can UNDO changes to it by just overwriting the whole page.

Your BufferPool already implements abort by deleting dirty pages, and pretends to implement atomic commit by forcing dirty pages to disk only at commit time. Logging allows more flexible buffer management (STEAL and NO-FORCE), and our test code calls

`BufferPool.flushAllPages()`
at certain points in order to exercise that flexibility.

1. Getting started

You should begin with the code you submitted for Lab 5 (if you did not submit code for Lab 5, or your solution didn't work properly, contact us to discuss options.)

You'll need to modify some of your existing source and add a few new files. Here's what to do:

- First change to your project directory (probably called simple-db-hw) and pull from the master GitHub repository:

```
$ cd simple-db-hw
$ git pull upstream master
```

- Now make the following changes to your existing code:
  1. Insert the following lines into `BufferPool.flushPage()` before your call to `writePage(p)`, where `p` is a reference to the page being written:

```
// append an update record to the log, with
// a before-image and after-image.
TransactionId dirtier = p.isDirty();
if (dirtier != null){
  Database.getLogFile().logWrite(dirtier, p.getBeforeImage(), p);
  Database.getLogFile().force();
}
```

  This causes the logging system to write an update to the log. We force the log to ensure the log record is on disk before the page is written to disk.
  2. Your `BufferPool.transactionComplete()` calls `flushPage()` for each page that a committed transaction dirtied. For each such page, add a call to `p.setBeforeImage()` after you have flushed the page:

```
// use current page contents as the before-image
// for the next transaction that modifies this page.
p.setBeforeImage();
```

```
After an update is committed, a page's before-image needs to be updated
```

so that later transactions that abort rollback to this committed version of the page.
(Note: We can't just call `setBeforeImage()` in `flushPage()`, since `flushPage()` might be called even if a transaction isn't committing. Our test case actually does that! If you implemented `transactionComplete()` by calling `flushPages()` instead, you may need to pass an additional argument to `flushPages()` to tell it whether the flush is being done for a committing transaction or not. However, we strongly suggest in this case you simply rewrite `transactionComplete()` to use `flushPage()`.)

- After you have made these changes, do a clean build (`ant clean; ant` from the command line, or a "Clean" from the "Project" menu in Eclipse.)
- At this point your code should pass the first three sub-tests of the `LogTest` systemtest, and fail the rest:

```
% ant runsystest -Dtest=LogTest
  ...
  [junit] Running simpledb.systemtest.LogTest
  [junit] Testsuite: simpledb.systemtest.LogTest
  [junit] Tests run: 10, Failures: 0, Errors: 7, Time elapsed: 0.42 sec
  [junit] Tests run: 10, Failures: 0, Errors: 7, Time elapsed: 0.42 sec
  [junit]
  [junit] Testcase: PatchTest took 0.057 sec
  [junit] Testcase: TestFlushAll took 0.022 sec
  [junit] Testcase: TestCommitCrash took 0.018 sec
  [junit] Testcase: TestAbort took 0.03 sec
  [junit]     Caused an ERROR
  [junit] LogTest: tuple present but shouldn't be
  ...
```

- If you don't see the above output from `ant runsystest -Dtest=LogTest`, something has gone wrong with pulling the new files, or the changes you made are somehow incompatible with your existing code. You should figure out and fix the problem before proceeding; ask us for help if necessary.

2. Rollback

Read the comments in `LogFile.java` for a description of the log file format.
You should see in `LogFile.java` a set of functions, such as `logCommit()`, that generate each kind of log record and append it to the log.

Your first job is to implement the `rollback()` function in `LogFile.java`. This function is called when a transaction aborts, before the transaction releases its locks. Its job is to un-do any changes the transaction may have made to the database.

Your `rollback()` should read the log file, find all update records associated with the aborting transaction, extract the before-image from each, and write the before-image to the table file. Use `raf.seek()` to move around in the log file, and `raf.readInt()` etc. to examine it. Use `readPageData()` to read each of the before- and after-images. You can use the map `tidToFirstLogRecord` (which maps from a transaction id to an offset in the heap file) to determine where to start reading the log file for a particular transaction.
You will need to make sure that you discard any page from the buffer pool whose before-image you write back to the table file.

As you develop your code, you may find the `Logfile.print()` method useful for displaying the current contents of the log.

Exercise 1: LogFile.rollback()

Implement LogFile.rollback().

After completing this exercise, you should be able to pass the TestAbort and TestAbortCommitInterleaved sub-tests of the LogTest system test.

3. Recovery

If the database crashes and then reboots, `LogFile.recover()` will be
called before any new transactions start. Your implementation should:

1. Read the last checkpoint, if any.
2. Scan forward from the checkpoint (or start of log file, if no checkpoint) to build the set of loser
   transactions. Re-do updates during this pass. You can safely start re-do at the checkpoint because `LogFile.logCheckpoint()` flushes all dirty buffers to disk.
3. Un-do the updates of loser transactions.

Exercise 2: LogFile.recover()

Implement LogFile.recover().

After completing this exercise, you should be able to pass all
of the LogTest system test.

4. Logistics

You must submit your code (see below) as well as a short (1 page, maximum) writeup describing your approach. This writeup should:

- Describe any design decisions you made, including anything that was difficult or unexpected.
- Discuss and justify any changes you made outside of LogFile.java.

4.1. Collaboration

This lab should be manageable for a single person, but if you prefer
to work with a partner, this is also OK. Larger groups are not allowed.
Please indicate clearly who you worked with, if anyone, on your writeup.

4.2. Submitting your assignment

You may submit your code multiple times; we will use the latest version you submit that arrives before the deadline (before 11:59 PM on the due date). Place the write-up in a file called `answers.txt` or `answers.pdf` in the top level of your `simple-db-hw` directory. **Important:** In order for your write-up to be added to the git repo, you need to explicitly add it:

```
$ git add answers.txt
```

You also need to explicitly add any other files you create, such as new *.java files.

The criteria for your lab being submitted on time is that your code must be **tagged** and **pushed** by the date and time. This means that if one of the TAs or the instructor were to open up GitHub, they would be able to see your solutions on the GitHub web page.

Just because your code has been commited on your local machine does not mean that it has been **submitted**; it needs to be on GitHub.

There is a bash script `turnInLab6.sh` in the root level directory of simple-db-hw that commits your changes, deletes any prior tag for the current lab, tags the current commit, and pushes the branch and tag to GitHub. If you are using Linux or Mac OSX, you should be able to run the following:

```
$ ./turnInLab6.sh
```

You should see something like the following output:

```
$ ./turnInLab6.sh
[master b155ba0] Lab 6
1 file changed, 1 insertion(+)
Deleted tag 'lab6' (was b26abd0)
To git@github.com:MIT-DB-Class/hw-answers-becca.git
- [deleted]         lab6
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 448 bytes | 0 bytes/s, done.
Total 6 (delta 3), reused 0 (delta 0)
To git@github.com:MIT-DB-Class/hw-answers-becca.git
  ae31bce..b155ba0  master -> master
Counting objects: 1, done.
Writing objects: 100% (1/1), 152 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:MIT-DB-Class/hw-answers-becca.git
* [new tag]         lab6 -> lab6
```

If the above command worked for you, you can skip to item 6 below. If not, submit your solutions for Lab 6 as follows:

1. Look at your current repository status.

   ```
   $ git status
   ```

2. Add and commit your code changes (if they aren't already added and commited).

   ```
   $ git commit -a -m 'Lab 6'
   ```

3. Delete any prior local and remote tag (*this will return an error if you have not tagged previously; this allows you to submit multiple times*)

   ```
   $ git tag -d lab6
   $ git push origin :refs/tags/lab6
   ```

4. Tag your last commit as the lab to be graded

   ```
   $ git tag -a lab6 -m 'lab6'
   ```

5. This is the most important part: **push** your solutions to GitHub.

   ```
   $ git push origin master
   $ git push origin lab6
   ```

6. The last thing that we strongly recommend you do is to go to the [MIT-DB-Class] organization page on GitHub to

make sure that we can see your solutions.
Just navigate to your repository and check that your latest commits are on GitHub. You should also be able to check
`https://github.com/MIT-DB-Class/hw-answers-(your student name)/tree/lab6`

Word of Caution

Git is a distributed version control system. This means everything operates offline until you run `git pull` or `git push`. This is a great feature.

The bad thing is that you may forget to `git push` your changes. This is why we strongly, **strongly** suggest that you check GitHub to be sure that what you want us to see matches up with what you expect.

4.3. Submitting a bug

SimpleDB is a relatively complex piece of code. It is very possible you are going to find bugs, inconsistencies, and bad, outdated, or incorrect documentation, etc.

We ask you, therefore, to do this lab with an adventurous mindset. Don't get mad if something is not clear, or even wrong; rather, try to figure it out
yourself or send us a friendly email.
Please submit (friendly!) bug reports to 6.830-staff@mit.edu.
When you do, please try to include:

- A description of the bug.
- A `.java` file we can drop in the `src/simpledb/test` directory, compile, and run.
- A `.txt` file with the data that reproduces the bug. We should be able to convert it to a `.dat` file using `PageEncoder`.

You can also post on the class page on Piazza if you feel you have run into a bug.

4.4 Grading

50% of your grade will be based on whether or not your code passes the system test suite we will run over it. These tests will be a superset of the tests we have provided; the tests we provide are to help guide your implementation, but they do not define correctness.
Before handing in your code, you should

make sure it produces no errors (passes all of the tests) from both `ant test` and `ant systemtest`.

Important: Before testing, we will replace your `build.xml`, `HeapFileEncoder.java`, `BPlusTreeFileEncoder.java`, and the entire contents of the `test/` directory with our version of these files. This means you cannot change the format of the `.dat` files! You should also be careful when changing APIs and make sure that any changes you make are backwards compatible. In other words, we will pull your repo, replace the files mentioned above, compile it, and then grade it. It will look roughly like this:

```
$ git pull
[replace build.xml, HeapFileEncoder.java, BPlusTreeFileEncoder.java and test]
$ ant test
$ ant systemtest
[additional tests]
```

If any of these commands fail, we'll be unhappy, and, therefore, so will your grade.

An additional 50% of your grade will be based on the quality of your writeup and our subjective evaluation of your code.

We've had a lot of fun designing this assignment, and we hope you enjoy hacking on it!