Project 4: Logging and Recovery

Due **April 13** at **11:55:00 pm**
Late submissions due on **April 17,** 2017 by 11:55PM (15% flat penalty)

Introduction

It's a sad fact of life: computers crash. One minute everything is working just fine, and the next minute everything that was in volatile memory is gone forever. To deal with this problem on your personal machine, you've probably gotten into the habit of saving frequently. In this project, you will implement an algorithm that database systems commonly use to address the same problem without slowing down the system too much.

Some Basics of Transactions[1]

A *transaction* is any one execution of a user program in a database management system (DBMS). For example, suppose Ada and Bob both have accounts at the same bank. Ada logs onto the bank's website and instructs the bank to make a payment of $100 from her account into Bob's account. If the bank's computer runs normally, it will deduct $100 from Ada's account balance, add $100 to Bob's account, and trigger emails to Ada and Bob confirming the transfer. Those steps—deducting, adding, and confirming that it was done—make up one transaction.

But suppose that the moment after the bank's computer deducts the money from Ada's account balance, it crashes. When it reboots, the instructions about that $100 that were in its memory are gone. We would like to ensure that if this happens, the $100 has not simply vanished—it is back in Ada's account, or it is in Bob's account. Either the whole transaction goes through, or the part of the transaction that happened before the crash gets undone. Another way of saying this is that we want a transaction to be **atomic**.

Suppose the timing of the crash was a little different. The bank's computer read Ada's balance into memory and modified it there, read Bob's balance into memory and modified it there, and triggered the emails. Then, just as it was about to write the updated balances back onto disk, it crashed. We want to make sure that if the system says a transaction completed successfully, the changes the transaction made will persist even through a crash. We call this **durability**.

(In lecture, you also learn about two other properties that transactions should have, called **consistency** and **isolation**. These have to do with concurrency and are important to understand, but you don't need to worry about them for this project.)

[1]Transactions will be covered in lecture soon, but if you just can't wait to learn more, check out chapter 16 of the textbook.

To ensure atomicity and durability while minimizing the harm to performance, DBMSs use **write-ahead logging** and with a recovery algorithm, such as **ARIES**. In this project, you will be implementing a single-threaded version of ARIES.

What you need to do:

Read.

You are strongly encouraged to read sections 16.7 through 16.7.2 and 18.1 through 18.6 in your textbook (Database Management Systems (3rd edition) - by Raghu Ramakrishnan and Johannes Gehrke, McGraw Hill, 2003, ISBN: 0-07-115 110-9). This is important. The rest of this project will not make sense until you have understood this material.

(We assume you already own the textbook or know that it can be checked out for two hours at a time from the Art Architecture & Engineering library's course reserve desk on the second floor. Older editions of the book Database Management Systems by Raghu Ramakrishnan (1998 and 2000) also have most of the same material -- though the chapter numbering is different. Older editions may be available cheaply at Amazon.)

Download.

Download the recovery simulator from Canvas and unzip it. You will see that the directory contains directories called StorageEngine, StudentComponent, output, correct, and testcases. The section on *Understanding the Simulator* below explains what these pieces do. You'll also want to spend some time familiarizing yourself with the header files.

Implement.

Implement ARIES in **LogMgr.cpp**. You will just be submitting this file to the autograder. There is no LogMgr.cpp file provided - you need to create this file yourself. The Makefile is set up to look for it under StudentComponent so you need to put it in the StudentComponent directory.

Do not alter the other files in any way. The autograder will compile and run your LogMgr.cpp with unaltered copies of all of the other files, so it is in your best interest to test with exactly the same files. Please do not add any other header files; if you need helper functions, you may declare them and define them in LogMgr.cpp.

You may use the C++11 STL, **except** that you may **not** use **do file I/O such as iostream or interact with the network**. The autograder may reject such submissions and your submission will still count.

Submission

The autograder is available at https://grader484.eecs.umich.edu. Submit your LogMgr.cpp file to the autograder. No other submission is necessary anywhere else.

You are allowed to work in pairs on this project. **If you are working in a pair,**

please join the same group with your partner **on the auto grader before making any submission**.

You may submit up to **three** times per day per group with autograder feedback. Please take advantage of this, and start submitting early and often.

We will keep your best score from various submissions. If you submit before the regular deadline and also during the late days, we will take the best of your scores, max(regular score without late penalty, late score with late penalty). So, there is no disincentive of submitting late.

Grading

Your grade will be based on your code's performance on the autograder test cases. We will diff your log file and your recovered db file against our known correct files for each test case.

There are hidden test cases on the autograder that are different from the public test cases we have provided you. Passing the public test cases means that you are on the right path, but doesn't ensure full marks on the private test case. The autograder test cases may also cover corner cases the public test cases do not. Thus, make sure you think carefully about all the cases that the algorithm needs to handle. However, your submission will not be checked against test cases not included in autograder. Hence, the final grade on autograder is your grade.

Understanding the Simulator

The simulator has two main pieces: the StorageEngine and the LogMgr. The StorageEngine handles all disk reads and writes. The LogMgr handles logging and recovery. If the LogMgr needs to write something to disk, it must go through the StorageEngine. **Your implementation of LogMgr.cpp must not include any static variables or functions, and it must not write directly to disk. If LogMgr wants to write to disk, it must use StorageEngine's public methods (interface).**

The StorageEngine keeps track of Pages. Each Page has a page_id, a pageLSN, a dirty bit, and a string of data.[2] All Pages are kept on disk. If a page needs to be read or written, StorageEngine adds it to the in-memory page buffer. If the page buffer is full, the StorageEngine chooses a page to flush to disk, informs the LogMgr of the flush through LogMgr's pageFlushed function (Hint: that probably means LogMgr should do something when the page is flushed), and flushes it, freeing up memory for the next page. StorageEngine's constructor declares the MEMORY_SIZE that it uses as its buffer pool:

```
StorageEngine::StorageEngine() : MEMORY_SIZE(10) {
   page_writes_permitted = 0;
}
```

[2] In a real database, of course, pages contain records, but here we've abstracted them away, so you don't need to worry about them.

Thus, in the above, it has 10 pages. If it ever needs to bring 11th page to memory, it selects a victim page and calls LogMgr:flushPage(page_id). See the StorageEngine:findPage(int page_id) method. For more thorough testing, you may want to change 10 to a smaller or larger value (no other change to StorageEngine should be required and you are not submitting StorageEngine.cpp). That will cause more stealing or less stealing, respectively. It will also change the output disk and log and you will have to manually check whether the resulting disk and log is correct.

Sometimes LogMgr will need to write Pages; for example, recovering from a crash and aborting a transaction both require LogMgr to alter Pages. When this is necessary, LogMgr can call StorageEngine::pageWrite. You should always try to minimize the number of Pages LogMgr writes. To force you to do this, StorageEngine limits the number of pageWrites you are permitted.[3] If you try to exceed the limit, pageWrites will stop writing pages and return false. If this happens, LogMgr should stop what it is doing.

The LogMgr also needs to keep the log. A log is made up of LogRecords of various types. You can keep the most recent part of the log (called the log tail) in memory, but sometimes you will need to write the log to disk. LogMgr can call a LogRecord's toString method to transform the LogRecord into a string, and then pass a string to StorageEngine::updateLog to append a string to the log on disk. The log on disk will have one record per line; you can append multi-line strings to it if you want to add more than one record at once. If LogMgr wants to read old log entries, it can call StorageEngine::getLog(), which will return a (multi-line) string. LogRecord::stringToRecordPtr can parse a line of that string and give you a pointer to a LogRecord of that line.

StorageEngine provides a few other utilities LogMgr might find useful:

- StorageEngine::nextLSN provides a unique id number assigned in monotonically increasing order
- StorageEngine::store_master and StorageEngine::get_master write an int to stable storage and retrieve it.
- StorageEngine::getLSN(int page_id) returns the page LSN of the specified page.

In the StorageEngine directory, you will also find main.cpp. Main takes the relative path to a testcase as a command line argument. For example, once you have compiled the whole project, you might call `./main.o testcases/test00` from the directory to run testcase 0. Main calls the runTestcase function, which will read in the testcase line by line and run a simulation.

[3] The limit at any given time is set by the testcase. Sometimes we give you enough page writes to finish an operation. Sometimes, to simulate a crash in the middle of an operation, you will run out of pageWrites early, and the next event in the testcase will be a crash.

Test Cases

Public testcases are given to you in the `testcases` folder. The first line of the testcase specifies a "database" file (db) to use for this run. For this simulation, we are using an oversimplified format, so that you can see the effects of writing to pages: a database file is a text file, and each line of text represents a page of data on disk. If the first line of a database file looks like this:

```
-1 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

it means that the page with page_id 1 (because it is the first line of the file) has pageLSN -1, and the data currently written to that page is 50 consecutive x's.

The remaining lines of the testcase are instructions to write, abort, commit, make a checkpoint, crash, or end the simulation. Each of these is described in more detail below, with examples.

- write (e.g., `1 write 34 27 ABC`): An instruction from a particular transaction (1) to write an update (ABC) to a particular page (34) at a particular offset (27). Main's runTestcase function calls StorageEngine's write function, which in turn calls LogMgr's write function, giving it the transaction id, page id, offset, input string, and what was already written at that page-offset location. StorageEngine expects to get back a pageLSN for the write; once it receives this, it updates the page.
- abort (e.g., `1 abort 5`): An instruction to abort a particular transaction (1), and a cap on the number of page writes the LogMgr is permitted (5). RunTestcase calls StorageEngine's abort function; StorageEngine sets the number of page writes permitted to five and then calls LogMgr::abort(1).
- commit (e.g., `1 commit`): An instruction to commit a transaction (1). RunTestcase calls LogMgr::commit(1).
- checkpoint (`checkpoint`): An instruction to create a checkpoint. RunTestcase calls LogMgr::checkpoint().
- crash *{v} (*e.g., `crash {2 5}`): where v is a list of integers. An instruction to simulate n crashes and recovery phases, where n is the number of values in the list v. During crash i, v[i] number of writes are permitted during each recovery phase. In this example, the first crash happens, and the LogMgr gets two writes before another crash happens; for the second crash, LogMgr gets five writes to recover. Sometimes the number of writes will allow a full recovery before the next crash; sometimes it will not, and the second crash will begin with StorageEngine no longer writing when LogMgr requests it. When crashing, RunTestcase will destroy the current instance of LogMgr and create a new one, then call StorageEngine::crash(). StorageEngine will destroy all of the Pages it has in memory, then call LogMgr's recover method. LogMgr is responsible for recovering according to the ARIES protocol.
- end (`end`): An instruction to end the simulation. Saves the final state of the on-disk pages and exits the program.

At the end of the testcase, the simulator will have generated two files: a log and a modified version of the input db file. These can be found in the `output/log/` and `output/dbs/`, respectively. Each filename includes the number of the corresponding testcase.

NOTE: If you already have a file in the log folder with the same name as the new one, the new log entries will be appended to the file, rather than overwriting it. You might want to rename or delete old log files to avoid some confusing bugs.

Reading the Log

The log file is saved as a multi-line string, with one line for each log entry, so that you can easily review it. The format is loosely based on Figure 18.2 in the textbook. For example, a log might have content like this (heading and table borders added for clarity):

LSN	prevLSN	tx_id	type	page_id	offset	before image	after image
2	-1	1	update	5	0	xxxxxxxx	update10
3	-1	2	update	3	0	xxxxxxxx	update20
4	3	2	commit				

End checkpoint log records also include a record of the transaction table and the dirty page table. Each table is enclosed in curly braces, and each entry in the table is enclosed in square braces. For example,

LSN	prevLSN	tx_id	type	TX_table	Dirty Page Table
5	4	-1	end_checkpoint	{[1 2 U] [2 3 U]}	{[3 3] [5 2]}

This transaction table has two entries, and so does the dirty page table.

At times, you have to read the log from the disk, which is in string form (see the sample logs under the `correct/logs` folder). You will have to add a function to LogMgr.cpp, which is declared in LogMgr.h:

```
vector<LogRecord *> LogMgr::stringToLRVector(string logstring)
```

The above function converts a log in the string form to reconstruct a vector of log records. To help do the conversion, a helper function is available in LogRecord.cpp that converts one line to a LogRecord of appropriate type, allocating the record using new().

```
LogRecord *LogRecord::stringToRecordPtr(rec_string)
```

Testing

We have given you some public test cases. The correct output for these can be found in the `correct/logs/` and `correct/dbs/` directories.

You will likely need to add your own tests. Testing is challenging as your goal is to write testcases that uncover behavior that is not being tested by existing testcases. You can use a GNU test coverage tool called `gcov` that works with g++. `gcov` will tell you whether your tests are covering every (or at least most) of the line in LogMgr.cpp. Makefile2 will build your project for test coverage analysis. Simply run "make" with the -f argument, specifying Makefile2 instead of Makefile.

% make -f Makefile2

Look at the contents of Makefile2 to see what it does. You may have to do a few edits as you add new test cases. It should run on CAEN and, hopefully, on Macs running recent versions of XCode with command-line tools.

Trouble Debugging?

If you are using gdb on the CAEN computers to debug, you might find that it has some difficulty displaying stl::map and other data structures that your implementation might use. To fix this, you can use Pretty Print:
Make a folder for it wherever you like to keep such things (mine is ~/gdb_printers) and, from inside that folder, run

```
svn co svn://gcc.gnu.org/svn/gcc/branches/gcc-4_6-branch/libstdc++-v3/python
```

Once you have that, create a file `~/.gdbinit` (or append to it if the file already exists) and add the following lines to it:

```
python
import sys
sys.path.insert(0, '/path/to/gdb_printers/python')
from libstdcxx.v6.printers import register_libstdcxx_printers
register_libstdcxx_printers (None)
end
```

Make sure you replace `/path/to/gdb_printers/python` with the actual path.

Now try running `gdb --args ./main.o testcases/test00` again, and you should be able to view the maps properly.

Important Information about the Honor Code

No sharing of code, use of code (or pseudo-code), or sharing of test case files written by others is permitted. We do run cheating detection software against all past and current submissions and will report any suspected violations to the Honor Council. Trying to hack the autograder or trick the autograder into giving you a good grade without doing the assigned work is cheating (though we welcome bug reports with the autograder, including any security vulnerabilities, as long as you don't use them to take advantage). Any violations of these policies caught will receive a zero on the project and will be referred to the Honor Council.

Your assignment is to implement ARIES within the simulator environment. As mentioned above, your LogMgr.cpp file must not attempt to read or write any file or use network calls directly, and it must not include any static functions or variables. They are not needed for this assignment.

A Few Clarifications:

- When you're undoing a transaction, any time you see a log record for that transaction, you need to get the prevLSN from that log record and add it to your toUndo list (or end the transaction if the prevLSN is -1). If the record is a CLR or an update record, you then process it the way the book says; if it's something else (like an abort record), you just move on to the next item in your toUndo list.
- You may have noticed in lecture and some of the problems in the textbook that the log tail is sometimes flushed to disk without a triggering commit or page write or end checkpoint. ARIES permits a regular background flushing of the log tail; for example, a certain amount of space in memory called a log buffer may be set aside, and any time it fills up, the log tail can be flushed to disk. **You don't need to worry about that background flushing for this project.** Any reasonably sized log buffer would be bigger than the log tail ever gets in these test cases.
- The book's description of abort is a little brief. What you need to know is that your abort function should write an abort record, then do basically the same thing as undo, only on a smaller scale. For undo, you need to look at all of the loser transactions and put their most recent LSNs in your toUndo queue. For abort, you just need to find the most recent LSN for one transaction you are aborting. Look at the log record for that LSN. Its prevLSN field will tell you which log record you need to look at next (or if you're done); its type will tell you if you need to process it the way you process records during undo. When you get to a log record with prevLSN = -1, process that record, then write your end log record, and you're done.