Project 3: Concurrency Control

Logistics

Due date: Tuesday 12/6/16, 11:59:59 PM

Grading

This project is worth **15%** of your overall course grade. Your project grade will be out of 100 points:

- 90 points for passing all of the tests provided for you. All tests are in `src/test/java/edu/berkeley/cs186/database`, so you can run them as you write code and inspect the tests to debug. Our testing provides extensive unit testing and some integration (end-to-end) testing.
- 10 points for writing your own, valid tests (10 tests total, 1 point each). The tests **must** pass both your implementation and the staff solution to be considered valid tests.

Extra Credit

You can earn up to 10 points in extra credit:

- 5 points for submitting **one week** before the due date (Tuesday 11/29/16, 11:59:59 PM).
- Up to 5 points for how effective your tests are. The way we will determine the effectiveness of your tests is based on how many edge cases you identify that the rest of the class did not identify. You **will not** be graded on how many other students' tests you pass.

Background

This project builds on top of the code and functionality that you should be familiar with from Project 1 and 2. In this project, you will be implementing a concurrency control manager with deadlock prevention, while using table-level shared and exclusive locking. Strict Two-Phase Locking has already been set up for you, and you can find this code in the Transaction inner class of `src/main/java/edu/berkeley/cs186/database/Database.java`. Essentially, when a transaction tries to add, update, or delete a record to/from a table, it attempts to acquire an exclusive lock on that table, and when it tries to get a specific record from a table, it tries to acquire a shared lock on that table. When a transaction ends, it releases all of its locks (the release phase of Strict 2PL).

We have released the staff solutions for Project 1 and 2 to a private repository (link can be found on Piazza), and you are free to integrate those into your codebase. Though if you feel confident about your own solutions to Project 1 and 2, you may stick with those as well (and we would really like it if you did since the whole point of these projects is to build your own implementation of a database!). Regardless, a correct working implementation of Project 1 and 2 is **necessary** in order to complete this project.

Also, please note that once you pull the starter code and Project 1 and 2 solutions (if you choose to use them), many of the tests for the earlier projects will not pass initially. This is because these tests require a working implementation of locking, and there used to be simple database-level locking in the codebase for the earlier projects, but that has now been replaced with table-level locking which you guys will implement in this project. We recommend running only the tests in the `concurrency` package as you complete this project and once they all pass, you should pass all the tests for this database system as well!

Getting Started

As usual, you can get the assignment from the Github course repository (assuming you've set things up as we asked you to in homework 0) by running `git pull course master`. This will get you all of the starter code we've provided, this project spec, and all of the tests that we've written for you.

We've also generated all of the API documentation of the code as webpages viewable here.

Setup

All of the Java, Maven, and IntelliJ setup from Project 1 and 2 is still applicable for this project. Assuming that you were able to complete Project 1 and 2, you should not need to do any additional setup for this project. Refer to the Project 1 spec for setup details.

Starter Code

Package Overview

There's a bunch of code that we've provided for you, both from Project 1 and 2, plus some new additions in this project. In the previous projects, we gave you a `LockManager` class that allowed for a single transaction to operate on the database at any given time, and until that transaction released its lock on the database, the whole database was locked from any other transactions. For this project, you will be modifying this lock manager so that multiple transactions can operate on the same database. The relevant parts for this project are all in the `concurrency` package. Briefly, it consists of a `LockManager` class that represents the actual lock manager, a `Lock` class which holds lock information for a specific table, and a `WaitsForGraph` class that will be useful for preventing deadlock. You'll be implementing table-level locking (no intent locks) and deadlock prevention in these three classes.

General Notes about Concurrency

You will need to learn about the `notifyAll()` and `wait()` methods. These methods are part of the Object superclass, and the basic idea is that calling an object's `wait()` method will cause the current thread to wait until another thread invokes `notifyAll()` on the same object. In our tests for this project, we run each lock acquisition/release operation for a transaction in its own thread, so these methods can be useful for forcing the thread on a lock object (for a specific table) to wait until another thread on that same lock object (so on the same table) has the transaction corresponding to it release its lock on that table. At this point, `notifyAll()` will be called on the lock, and the other threads on the lock can wake up and their transactions can try to get the lock (based on a thread-safe FIFO queue we will maintain). By doing this threading logic inside the lock class, we can guarantee that only the transactions on a specific table will be blocked if they cannot get their lock on that table rather than all the transactions in the lock manager (aka table-level locking).

If you want more information about these methods and how to use them, check out this link regarding coordinating different threads.

Within `Lock.java`, you'll find synchronized methods, which are in turn called by your `LockManager`. Basically, we want to synchronize any methods that modify the lock object (such as `Lock#acquire` and `Lock#release`) because otherwise the thread for the transaction requesting or releasing a lock will not have ownership of the object's monitor, a construct that allows the thread to wait or be notified. We need to synchronize these methods to handle transaction blocking and lock promotion from the waiting queue.

Your Assignment

Let's get started with the last project! You will be responsible for implementing various methods in the `LockManager`, `Lock`, and `WaitsForGraph` classes.

NOTE: Please do not change any of the interfaces that we've given you. It's also a good idea to always check the course repo for updates on the project.

Part 1: Locking

The first part of the project involves completing logic to acquire and release locks as well as checking to see if a transaction holds a lock. In order to do this, you will need to implement **queueing logic** within `Lock` so you can keep track of locks that have been granted to each table, as well as locks that were blocked for each table.

Also, your `LockManager` has a `ConcurrentHashMap` that maps a table name to a `lockType`. This data structure is thread-safe. We have already implemented for you the logic of mapping tables to lock objects in `LockManager#acquireLock`. The lock object keeps track of two things:

> 1) Which transactions own the lock on that table (the "granted" group of transactions) and what the lock mode is (shared or exclusive).
>
> 2) A thread-safe queue of `LockRequests` (which consist of transactions and the desired lock types) that are waiting to be granted a lock on that table

To reiterate, we will only implement two lock modes in this project: shared and exclusive. Note that we have defined a bunch of potentially useful methods in `Lock.java` so make sure you take a look at them!

1.1 Acquire Lock

For the first part of this project you'll need to implement queueing logic in the `Lock#acquire` method, which is used in `LockManager#acquireLock`.

When a transaction requests a lock in a mode that conflicts with the current holder of the lock, the requesting transaction should be placed on a FIFO queue of transactions waiting for that lock. It should then `wait()` until it can successfully acquire the lock which we have defined the conditions for below. At this point, it can be removed from the queue, and you must modify the lock's owners/type as appropriate.

We recommend you have a helper method that checks to see if the specified lock request is **compatible for promotion** initially or whenever another lock is released. A lock request is compatible for promotion (to receive the lock type it requested) if it is compatible with the transactions that own this lock object and the type of the lock object (based on the compatibility matrix between shared and exclusive locks defined in lecture) **and ONE** of the following conditions is true: * the request is an upgrade (from shared to exclusive) * the request is in front of the queue * the request is for a shared lock and there are only other shared locks in front of it on the queue

Note that we prioritize lock upgrades meaning that if a transaction releasing a lock on the table allows for a lock upgrade for another transaction that owns a shared lock on that table, we prioritize the upgrade over what is in front of the queue.

As a note, the client is issuing requests in a synchronous fashion, which means that if a request gets blocked (because the proper lock cannot be granted), the client cannot issue the next request until it gets back the response for the current request. In addition, a transaction cannot appear in a wait queue more than once. The following situation won't happen for a given key:

> granted group: (T1(X))
>
> wait queue: (T2(S), T2(X))

This is impossible because T2 requested an S lock, then requested an X lock before the first request could finish.

1.2 Release Lock

In this section you'll need to complete `Lock#release`, which is called by `LockManager#releaseLock`. The `releaseLock` method releases the lock for a given `transNum` and `tableName`.

When a transaction leaves the system, all mutually compatible transactions at the head of the FIFO queue should be granted the lock, but keep in mind that we prioritize lock upgrades if possible. After the lock has been released, this method should also make use of `notifyAll()` to notify all the other threads waiting to obtain their desired lock type on this table to see if they are now eligible for promotion (this logic you have just implemented in `Lock#acquire`).

1.3 Holds Lock

For the last part of this section you'll need to implement `Lock#holds`, which is called by `LockManager#holdsLock`. This method should be short -- it simply checks to see if a given transaction holds a lock of a given type.

1.4 Testing

Once you've implemented all of these methods, you should be passing all of the tests in `TestLockManager.java`. We strongly recommend you start writing tests once you've wrapped your head around the code to try to catch some of the edge cases that you might have missed. It's generally a good idea to write your own tests as you go along due to the fact that the given tests don't cover all edge cases. Please look at the tests in `TestLockManager.java` for reference on how to format your tests.

Part 2: Deadlock Prevention

The second part of the project is focused on deadlock prevention.

As an overview, we are going to prevent deadlocks by checking to see if a certain lock request by a transaction would cause a deadlock and throwing a `DeadlockException` if so. We've provided code to build a waits-for graph in the `WaitsForGraph` class. This class also has methods that you can use to modify the nodes and edges in the graph as you see fit.

2.1 Cycle Detection

For the first part of deadlock prevention, you need to implement `WaitsForGraph#edgeCausesCycle`. This method doesn't modify the waits-for graph; it checks to see if the edge in question will create a cycle in the graph, which can be achieved through an extension of DFS. Since cycle detection was not explicitly presented in class, if you find an implementation online that works for you, you may use it, but please site your sources in the comments for this function. Also, make sure that you are not changing the structure of the graph with this method. You can add edges if you need, but make sure to remove them later.

2.2 Handling Potential Deadlocks

You will now implement `LockManager#handlePotentialDeadlock`. In this function, you should add any nodes/edges that need to be added to the `WaitsForGraph` corresponding to this `LockManager` due to this lock request. However, if adding the transaction to the waits-for graph causes a deadlock due to an edge that would be created by the request, throw a `DeadlockException`.

2.3 Testing

If you've completed all the sections up to this point, you should now be passing **all** of the tests that we've given you. Again, we strongly encourage you to write tests as you go to try to catch any relevant bugs in your implementation. To test for deadlock, please see the tests in `TestDeadlockPrevention.java` for reference. Note that instead of just using the Java `Thread` class to run our transaction lock acquire/release operations, we are using the `AsyncDeadlockTesterThread` class thread we have defined. This class essentially serves to report any errors that occur inside the `run()` method of the `Runnable` argument of a thread back to the main thread so that JUnit can detect the error (in this case a `DeadlockException`). Please make use of this class in your own deadlock tests.

Part 3: Testing

We can't emphasize enough how important it is to test your code! Like we said earlier, writing valid tests that **test actual code** (i.e., don't write `assertEquals(true, true);`, or we'll be mad at you) is worth 10% of your project grade.

CS 186 is a design course, and validating the reasonability and the functionality of the code you've designed and implemented is very important. We suggest you try to find the trickiest edge cases you can and expose them with your tests. Testing that your code does exactly what you expect in the simplest case is good for sanity's sake, but it's often not going to be where the bugs are.

To encourage you to try to find interesting edge cases, we're going to give you some extra credit. If you find edge cases that most other students didn't find, you can get up to 5 points of extra credit. Keep in mind that if some other people find the same edge cases as you, that doesn't mean you won't get extra credit -- so don't treat this as a competition!

Writing Tests

In the `src/test` directory you'll notice we've included several tests for you already. You should take a look at these to get a sense of how to write tests. You should write your tests in one of the existing files according to the functionality you're trying to test.

All test methods you write should have both the `@Test` and `@Category(StudentTestP3.class)` annotations. Note that this is different from the `@Category(StudentTest.class)` and `@Category(StudentTestP2.class)` annotations you used in Project 1 and 2 respectively! This is important in making sure that your Project 3 tests are not mixed up with your Project 1 and 2 tests. We have included an example test in the `TestLockManager` class:

```
@Test
@Category(StudentTestP3.class)
public void testSample() {
  assertEquals(true, true); // Do not actually write a test like this!
}
```

Then whenever you run `mvn test`, your tests will be run as well. To run only the tests that you wrote for this project, you may run `mvn test -Dtest=StudentTestSuiteP3`. You now also have the ability to run only the tests in a specific package. For example, you may run `mvn test -Dtest="edu.berkeley.cs186.database.concurrency.*"` to run all of the tests in the `concurrency` package. This may be helpful for debugging to save you time from running the whole test suite.

Part 4: Feedback

Congratulations, you guys have finished all the projects, and (ideally) you have implemented a working database system complete with an interface to work with tables and records, perform optimized queries, and deal with concurrency control issues!

We've been working really hard to give you guys the best experience possible with these new projects. We'd love to improve on them and make sure we're giving reasonable assignments that are helping you learn. In that vein, please fill out this Google Form to help us understand how we're doing!