6.5830/6.5831 Lab 3: GoDB Transactions

Assigned: Wednesday October 23, 2024 **Checkpoint Due:** Wednesday October 30, 2024 by 11:59 ET **Due:** Wednesday November 6, 2024 by 11:59 PM ET

In this lab, you will implement a simple locking-based transaction system in GoDB. You will need to add lock and unlock calls at the appropriate places in your code, as well as code to track the locks held by each transaction and grant locks to transactions as they are needed.

The remainder of this document describes what is involved in adding transaction support and provides a basic outline of how you might add this support to your database.

As with the previous lab, we recommend that you start as early as possible. Locking and transactions can be quite tricky to debug!

1. Getting started

You should begin with the code you submitted for Lab 2 (if you did not submit code for Lab1 / Lab 2, or your solution didn't work properly, contact us to discuss options). Additionally, we are providing extra test files for this lab that are not in this original code distribution you receive.

1.1. Getting Lab 3

You will need to add these new files to your release. The easiest way to do this is to navigate to your project directory (probably called `go-db-2024`) and pull from the master GitHub repository:

```
$ cd go-db-2024
$ git pull upstream main
```

You should see three new files in `/godb`: `locking_test.go`, `transaction_test.go`, and `deadlock_test.go`. There should be no merge conflicts except for possibly `go.mod` and `go.sum`. If you experience conflicts for `go.mod` and `go.sum`, resolve it by accepting either change, run `go test` in `/godb`, and follow the instructions to install the modules correctly.

1.2. Transaction Locks vs Mutexes

In this lab, we use the term "lock" to refer to a database lock on a part of the database (e.g., a page) that is protected using the two-phase locking protocol. We will use the term "mutex" to refer to the go construct that is used to prevent two threads from concurrently executing a piece of code; note that mutexes can be locked and unlocked. There are several other synchronization primitives that can be used to isolate threads in the go sync package; for example, concurrent maps may be useful in your buffer pool implementation.

2. Transactions, Locking, and Concurrency Control

Before starting, you should make sure you understand what a transaction is and how strict two-phase locking (which you will use to ensure isolation and atomicity of your transactions) works.

In the remainder of this section, we briefly overview these concepts and discuss how they relate to GoDB.

2.1. Transactions

A transaction is a group of database actions (e.g., inserts, deletes, and reads) that are executed *atomically*; that is, either all of the actions complete or none of them do, and it is not apparent to an outside observer of the database that these actions were not completed as a part of a single, indivisible action.

In GoDB, and most database systems, each transaction runs in a separate thread. This means that there can concurrently be multiple threads attempting to invoke methods on the database. You will need to use mutexes or other synchronization primitives to prevent these concurrent threads from experiencing race conditions that may result in indeterminate behavior.

2.2. The ACID Properties

To help you understand how transaction management works in GoDB, we briefly review how it ensures that the ACID properties are satisfied:

- **Atomicity**: Strict two-phase locking and careful buffer management ensure atomicity.
- **Consistency**: The database is transaction consistent by virtue of atomicity. Other consistency issues (e.g., key constraints) are not addressed in GoDB.
- **Isolation**: Strict two-phase locking provides isolation.
- **Durability**: A FORCE buffer management policy ensures durability (see Section 2.3 below).

2.3. Recovery and Buffer Management

For this lab, you should implement a NO STEAL/FORCE buffer management policy.

As we discussed in class, this means that:

- You shouldn't evict dirty (updated) pages from the buffer pool if they are locked by an uncommitted transaction (this is NO STEAL). Your buffer pool implementation already does this.
- On each transaction commit, you should force the relevant dirty pages to disk (e.g., flush the pages out) (this is FORCE).

You may assume that GoDB will not crash while processing a `CommitTransaction` or `AbortTransaction` command. Note that these two points mean that you do not need to implement log-based recovery in this lab since you will never need to undo any work (you never evict dirty pages), and you will never need to redo any work (you force updates on commit and will not crash during commit processing).

2.4. Granting Locks

You will need to add calls to GoDB (in `buffer_pool.go`, for example), that allow a caller to request or release a (shared or exclusive) lock on a specific object on behalf of a specific transaction.

We recommend locking at *page* granularity; please do not implement table-level locking (even though it is possible). The rest of this document and our unit tests assume page-level locking.

You will need to create data structures that keep track of which locks each transaction holds and check to see if a lock should be granted to a transaction when it is requested.

You will need to implement shared and exclusive locks; recall that these work as follows:

- Before a transaction can read an object, it must have a shared lock on it.
- Before a transaction can write an object, it must have an exclusive lock on it.
- Multiple transactions can have a shared lock on an object.
- Only one transaction may have an exclusive lock on an object.
- If transaction *t* is the only transaction holding a shared lock on an object *o*, *t* may *upgrade* its lock on *o* to an exclusive lock.

If a transaction requests a lock that cannot be immediately granted, your code should *block*, waiting for that lock to become available (i.e., be released by another transaction running in a different thread). Recall that multiple threads will be running concurrently. Be careful about race conditions in your lock implementation --- think about how concurrent invocations to your lock may affect the behavior. To block a thread, you can simply call `time.Sleep` for a few milliseconds.

2.5. Transactions

In GoDB, a `TransactionID` variable is created at the beginning of each query. This object is passed to each of the operators involved in the query. When the query is finished, the appropriate `BufferPool` method, `CommitTransaction()` or `AbortTransaction()`, is called.

The test cases we have provided you with create the appropriate `TransactionID` objects, pass them to your operators in the appropriate way and invoke `CommitTransaction()` or `AbortTransaction()` when a query is finished. We have also implemented `TransactionID` and `NewTID` for you.

2.6. Lock Lifetime

You will need to implement strict two-phase locking. This means that transactions should acquire the appropriate type of lock on any object before accessing that object and shouldn't release any locks until after the transaction is committed.

Fortunately, the GoDB design is such that it is possible to obtain locks on pages in `BufferPool.GetPage()` before you read or modify them. So, rather than adding calls to locking routines in each of your operators, you should acquire locks in `GetPage()`. You will implement releasing of locks when you implement `CommitTransaction()` and `AbortTransaction()` below.

You will need to acquire a *shared* lock on any page (or tuple) before you read it, and you will need to acquire an *exclusive* lock on any page (or tuple) before you write it. You will notice that we are already passing around `RWPerm` variables in the BufferPool; these variables indicate the type of lock that the caller would like to have on the object being accessed (we have given you code for the `RWPerm` type.)

Our testing system relies on the godb implementation returning an error when the transaction is aborted (due to deadlocks etc.). If the transaction is aborted, your implementation is not responsible for restarting the transaction. Simply return an error and the test suite (called/user of the system) will restarte the transaction. Before you start implementing anything for this lab, check that `TestTransactionTid` passes. This test relies on the implementation we have provided you for `NewTID()` which the rest of the system and the tests depend on. If it does not pass, contact the course staff. At this point, `TestTransaction` will not terminate since `insertTuple` returns an error for a buffer pool full with dirty pages, not an aborted transaction (see `readXaction` and `writeXaction` in `transaction_test.go`). This part will execute normally after you implement exercises 1 and 2.

Exercise 1.

Write the methods that acquire transactional locks in BufferPool. Assuming you are using page-level locking, you will need to modify `GetPage` to block and acquire the desired lock (specified by `RWPerm`) before returning a page. `GetPage` receives a `TransactionID` that is attempting to acquire the lock. You will want to allocate data structures that keep track of the shared and exclusive locks each transaction is currently holding.

Please note that unlike in previous tests, there will be multiple threads concurrently calling `GetPage()` during this test. Use `sync.Mutex` or the `sync.Map` construct to prevent race conditions. Think about what happens if two threads simultaneously try to read or evict a page. The simplest approach (which we recommend) is:

- Associate a `Mutex` with your buffer pool.
    - Acquire this mutex before you access any of the data structures you used to keep track of which pages are locked; this will ensure only one thread is trying to use the data structures in the buffer pool to acquire a page lock at a time.
        - If you successfully acquire the page lock, you should release the buffer pool mutex after lock acquisition.
        - If you fail to acquire the lock, you will block.
            - You will need to release the mutex before blocking (to allow another thread/transaction to attempt to acquire the lock)
            - Attempt to re-acquire the mutex before trying to re-acquire the lock.

Exercise 2.

Implement the `BeginTransaction()`, `CommitTransaction()` and `AbortTransaction()` methods in `BufferPool`.

`BeginTransaction()` may or may not need to do anything depending on your design choices -- you may want to store the transaction id in a list of running transactions.

When you commit, you should flush dirty pages associated with the transaction to disk. When you abort, you should revert any changes made by the transaction by restoring the page to its on-disk state (which can be done simply by discarding the page from memory since we never write dirty pages back to disk).

Whether the transaction commits or aborts, you should also release any state the `BufferPool` keeps regarding the transaction, including releasing any locks that the transaction held.

As with previous methods, you will need to use mutexes or other synchronization to ensure correctness when two transactions simultaneously attempt to abort or

commit. In our implementation, we used the `Mutex` associated with our buffer pool to protect the entire body of each of these three methods.

At this point, your code should pass the tests in `locking_test.go`, `TestTransactionTwice`, and `TestTransaction{Commit, Abort}` unit tests and the `TestAbortEviction` system test. You may find the `Test{One, Two, Five}Threads` and `TestAllDirtyFails` system tests illustrative, but they will likely fail until you complete the next exercises.

2.7. Changes to Methods Outside of Buffer Pool

Double check that your implementation of `HeapFile.insertTuple()` and `HeapFile.deleteTuple()`, as well as the implementation of the iterator returned by `HeapFile.Iterator()` access pages using `BufferPool.GetPage()`. Double check that these different uses of `GetPage()` pass the correct permissions object (e.g., `WritePerm` or `ReadPerm`). You may also wish to double check that your implementation of `HeapFile.insertTuple()` and `HeapFile.deleteTuple()` call `setDirty()` on any of the pages they access (you should have done this when you implemented this code in lab 1.)

You will also need to ensure that your methods behave properly under concurrency. Transactional locking will prevent methods like `insertTuple` or `deleteTuple` from being called on the same `HeapPage` object by two different transactions (and hence two different threads), but your `HeapFile` itself may have shared variables that need to be protected with mutexes. For example, your heap file implementation may use a variable to track the number of pages or the next page to insert into; you will want to ensure that threads are isolated from each other when one or both of them are updating these variables. There also may be race conditions that you will need to think through. For example, in your implementation, you will want to ensure that two threads do not simultaneously try to insert a new tuple that adds a new page to the HeapFile (e.g. because two transactions try to do an insert into a heap file with no empty slots on any pages).

Exercise 3.

Add synchronization primitives like mutexes throughout GoDB. For most implementations, the primary code to be concerned about is in HeapFile. Some (but not necessarily all) actions that you should verify work properly:

- Reading tuples off of pages during your Iterator. Note that it is okay for two threads to read the same variable at the same time -- its concurrent modification by both threads or modification by one and reading by another that is problematic. Also, recall that transactional locking will prevent one transaction from inserting into a page while another is reading from it.
- Inserting and deleting tuples through HeapFile methods.
- Adding a new page to a `HeapFile`. When do you physically write the page to disk? Are there race conditions with other transactions (on other threads) that might need special attention at the HeapFile level, regardless of page-level locking?
- Looking for an empty slot into which you can insert tuples.

In the staff implementation, we only needed a mutex in two places. We added a `Mutex m` to our HeapFile. We then locked `m` on entry to `insertTuple` and released it before any return. We also locked `m` on entry to `deleteTuple` and released before any return. This is because we want to avoid two inserts adding a page at the same time and because we have some shared heap file variables that keep track of the last page inserted into and the total number of pages. We didn't need to acquire the mutex during our iterator because we know that no other transaction will modify a page while we are scanning it.

There are no test cases for this exercise because the places where synchronization needs to be added are dependent on your implementation.

2.8. Implementing NO STEAL

Modifications from a transaction are written to disk only after it commits. This means we can abort a transaction by discarding the dirty pages and rereading them from the disk. Thus, we must not evict dirty pages. This policy is called NO STEAL.

Exercise 4.

Double-check that you don't evict dirty pages from the buffer pool. We will test this later in `TestAllDirtyFails` but you probably cannot pass this test case yet.

2.9. Deadlocks and Aborts

It is possible for transactions in GoDB to deadlock -- if you do not understand why, we recommend reading about deadlocks in the reading on Concurrency Control and Recovery (i.e., the reading for Lecture 10 and 11). You will need to detect this situation and return an error.

There are many possible ways to detect a deadlock. A simple method would be to implement a timeout policy that aborts a transaction if it has not been completed after a given period of time. For a better solution, you may implement cycle-detection in a dependency graph data structure as shown in lecture. In this scheme, you would check for cycles in a dependency graph periodically or whenever you attempt to grant a new lock, and abort something if a cycle exists. After you have detected that a deadlock exists, you must decide how to improve the situation. Assume you have detected a deadlock while transaction *t* is waiting for a lock. In theory, you could abort **all** transactions that *t* is waiting for; this may result in a large amount of work being undone, but you can guarantee that *t* will make progress. Alternately, you may decide to abort *t* to give other transactions a chance to make progress. This means that the end-user will have to retry transaction *t*.

Another approach is to use global orderings of transactions to avoid building the wait-for graph. This is sometimes preferred for performance reasons, but transactions that could have succeeded can be aborted by mistake under this scheme. Examples include the WAIT-DIE and WOUND-WAIT schemes.

Exercise 5.

Implement deadlock detection or prevention in `BufferPool.GetPage()`. You have many design decisions for your deadlock handling system, but it is not necessary to do something highly sophisticated. We expect you to do better than a simple timeout on each transaction. A good starting point will be to implement cycle-detection in a wait-for graph before every lock request, and you will receive full credit for such an implementation. Please describe your choices in the lab writeup and list the pros and cons of your choice compared to the alternatives.

You should ensure that your code aborts transactions properly when a deadlock occurs, which means calling `AbortTransaction()` and returning an error. You are not expected to automatically restart a transaction which fails due to a deadlock -- you can assume that higher-level code will take care of this.

You will need to be careful about acquiring and releasing mutexes here -- if `AbortTransaction` also acquires the buffer pool mutex, your `GetPage` will need to release the mutex before calling `AbortTransaction.`

We have provided some (not-so-unit) tests in `deadlock_test.go`. They are a bit involved, so they may take more than a few seconds to run (depending on your policy). If they seem to hang indefinitely, then you probably have an unresolved deadlock. These tests construct simple deadlock situations that your code should be able to escape.

Note that there are two timing parameters near the top of `deadlock_test.go`; these determine the frequency at which the test checks if locks have been acquired and the waiting time before an aborted transaction is restarted. You may observe different performance characteristics by tweaking these parameters if you use a timeout-based detection method.

Your code should now should pass the `Test{One, Two, Five}Threads` and `TestAllDirtyFails` tests (which may also run for quite a long time depending on your implementation).

At this point, you should have a recoverable database, in the sense that if the database system crashes (at a point other than `CommitTransaction()` or `AbortTransaction()`) or if the user explicitly aborts a transaction, the effects of any running transaction will not be visible after the system restarts (or the transaction aborts.) You may wish to verify this by running some transactions and explicitly killing the database server.

3. Logistics

You must submit your code (see below) as well as a short (2 pages, maximum) writeup describing your approach. This writeup should:

- Describe any design decisions you made, including your choice of deadlock detection.
- Discuss and justify any changes you made to the API.
- Describe any missing or incomplete elements of your code.
- Describe how long you spent on the lab, and whether there was anything you found particularly difficult or confusing.

3.1. 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 individual writeup.

3.2. Submitting your assignment

We will be using Gradescope to autograde all programming assignments. You should have all been invited to the class instance; if not, please check Piazza for an invite code. If you are still having trouble, let us know, and we can help you set up. You may submit your code multiple times before the deadline; we will use the latest version as determined by Gradescope. Place the write-up in a file called `lab3-writeup.txt` with your submission.

If you are working with a partner, only one person needs to submit to Gradescope. However, make sure to add the other person to your group. Also, note that each member must have their own writeup. Please add your Kerberos username to the file name and in the writeup itself (e.g., `lab3-writeup-username1.txt` and `lab3-writeup-username2.txt`).

The easiest way to submit to Gradescope is with `.zip` files containing your code. On Linux/macOS, you can do so by running the following command:

```
$ zip -r submission.zip godb/ lab3-writeup.txt

# If you are working with a partner:
$ zip -r submission.zip godb/ lab3-writeup-username1.txt lab3-writeup-username2.txt
```

3.2.1 Checkpoint Submission

Like in lab 1, we are encouraging you to submit a checkpoint submission. If your checkpoint submission passes 15 points worth of testcases and includes a preliminary writeup describing what parts you find confusing, you will receive a donut the following week in lecture! The following day, we will host a bootcamp for this lab, addressing the questions you mentioned in the preliminary writeup. If you finish the lab before the checkpoint deadline, you will receive a super donut!

3.3. Submitting a bug

Please submit (friendly!) bug reports to 6.5830-staff@mit.edu or as a post on Piazza. When you do, please try to include:

- A description of the bug.
- A `.go` file with test functions that we can drop into the `godb` directory, compile, and run.
- A `.txt` file with the data that reproduces the bug.

If you are the first person to report a particular bug in the code, we will give you a sweet treat!

3.4 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. Before handing in your code, you should make sure it produces no errors (passes all of the tests) when you run `go test` in the `godb` directory.

New:

- Given that this lab will require you to heavily modify your earlier code, we strongly recommend that you make sure your implementation passes the tests from

earlier labs. If you are having trouble making progress because your implementation fails tests from the previous labs, please contact us to discuss options.

- Given that this lab deals with concurrency, we will rerun the autograder on the submission you activated after the due date to discourage trying buggy code until lucky. It is your responsibility to ensure that your code **reliably** passes the tests.
- This lab has a higher percentage of manual grading at 50% compared to previous labs. Specifically, we will be very unhappy if your concurrency handling is bogus (e.g., inserting `time.Sleep(time.Second)` until a race disappears).

Important: Before testing, Gradescope will replace the go test files with our version of these files. This means you should make sure that your code passes the unmodified tests. Since there are many ways to implement these concurrency controls and this is the last lab before lab 4, you can modify the interface as long as you pass the tests as described above, although there are solutions that closely match to what you learned in class without modifying these interfaces. However, you should not modify `catalog.go`, any `.txt`, any `.csv`, (since the test setup code relies on these), `mem_file.go`, `parser.go`. The only parts you might want to modify in `types.go` are the three `interface`s.

You should get immediate feedback and error outputs for failed visible tests (if any) from Gradescope after submission. There may exist several hidden tests (a small percentage) that will not be visible until after the deadline. The score given will be your grade for the auto-graded portion of the assignment. An additional 50% of your grade will be based on the quality of your writeup and our subjective evaluation of your code. This part will also be published on Gradescope after we finish grading your assignment.

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