



PROJECT #3 - CONCURRENCY CONTROL

OVERVIEW

The third programming project is to implement a **concurrent index** and **lock manager** in your database system. The first task is to implement a lock manager which is responsible for keeping track of the **tuple-level locks** issued to transactions and supporting shared & exclusive lock grant and release. The second task is an extension of [project #2](#) where you will enable your B+tree index to support multi-threaded updates.

The project is comprised of the following two tasks:

- [Task #1 - Lock Manager](#)
- [Task #2 - Concurrent Index](#)

This is a single-person project that will be completed individually (i.e., no groups).

? Please post all of your questions about this project on Canvas. Do **not** email the TAs directly with questions. The instructor and TAs will **not** teach you how to debug your code.

Release Date: Oct 25, 2017

Due Date: Nov 15, 2017 @ 11:59pm

PROJECT SPECIFICATION

Like the first project, we are providing you with stub classes that contain the API that you need to implement. You should **not** modify the signatures for the pre-defined functions in these classes. If you do this, then it will break the test code that we will use to grade your assignment you end up getting no credit for the project. If a class already contains certain member variables, you should **not** remove them. But you may add private helper functions/member variables to these classes in order to correctly realize the functionality.

The correctness of this project depends on the correctness of your implementation of previous projects, we will **not** provide solutions or binary file. Since the [second task](#) is closely related to the second project, we recommend that you get your index working correctly before you proceed with this project. You will be allowed to submit your second project to AutoLab after the second project deadline.

TASK #1 - LOCK MANAGER

To ensure correct interleaving of transactions' operations, the DBMS will use a lock manager (LM) to control when transactions are allowed to access data items. The basic idea of a LM is that it maintains an internal data

that transaction, or abort it.

In your implementation, there will be a global LM for the entire system (similar to your buffer pool manager). The **TableHeap** class will use your LM to acquire locks on tuple records (by record id **RID**) whenever a transaction wants to access/modify a tuple. The DBMS will only support **REPEATABLE READ** isolation level (i.e., you do not need to support range-locks so it cannot support **SERIALIZABLE** isolation).

This task requires you to implement a tuple-level LM that supports both two-phase locking (2PL) and strict two-phase locking (S2PL). The system will pass in a flag to the LM to say what version of 2PL it will use during its initialization. Your LM will implement the **WAIT-DIE** policy to prevent deadlocks. In the repository, we are providing you with a **Transaction** context handle (`include/concurrency/transaction.h`) that maintains the current state of the transaction (i.e., GROWING, SHRINKING, ABORTED, COMMITTED) and information about its acquired locks. The LM will need to check/update the state of transaction correctly. Any failed lock operation should return false and change the transaction state to ABORTED (implicit abort).

We recommend you to read [this article](#) to refresh your C++ concurrency knowledge. More detailed documentation is [available here](#).

REQUIREMENTS AND HINTS

The only file you need to modify for this task is the **LockManager** class (`concurrency/lock_manager.cpp` and `concurrency/lock_manager.h`). You will need to implement the following functions:

- **LockShared(Transaction, RID)** : Transaction **txn** tries to take a shared lock on record id **rid**. This should be blocked on waiting and should return true when granted. Return false if transaction is rolled back (aborts).
- **LockExclusive(Transaction, RID)** : Transaction **txn** tries to take an exclusive lock on record id **rid**. This should be blocked on waiting and should return true when granted. Return false if transaction is rolled back (aborts).
- **LockUpgrade(Transaction, RID)** : Transaction **txn** tries to upgrade a shared to exclusive lock on record id **rid**. This should be blocked on waiting and should return true when granted. Return false if transaction is rolled back (aborts).
- **Unlock(Transaction, RID)** : Unlock the record identified by the given record id that is held by the transaction. There is only a tiny corner case you will fail on **Unlock** method (return false). (Hint: Think about the differences between 2PL and strict 2PL)

You should first take a look at the `transaction.h` and `lock_manager.h` to become familiar with the API and member variables we provide. You have the freedom of adding any necessary data structures in `lock_manager.h`. You should consult with Chapters 15.1-15.2 in the textbook and make sure that your implementation follows the 2PL protocol carefully.

COMMON PITFALLS

- The LM's constructor will be told whether it should follow strict 2PL or not. You should also maintain state of transaction. For example, the states of transaction may be changed from **GROWING** phase to **SHRINKING** phase due to **unlock** operation.

the LM can release them properly.

TASK #2 -CONCURRENT INDEX

In this part, you need to update your original single-threaded B+Tree index so that it can support concurrent operations. We will use the latch crabbing technique described in class and in the textbook. The thread traversing the index will acquire then release latches on B+Tree pages. A thread can **only** release latch on a parent page if its child page considered "safe". Note that the definition of "safe" can vary based on what kind of operation the thread is executing:

- **Search** : Starting with root page, grab read (**R**) latch on child Then release latch on parent as soon as you land on the child page.
- **Insert** : Starting with root page, grab write (**W**) latch on child. Once child is locked, check if it is safe, in this case, not full. If child is safe, release **all** locks on ancestors.
- **Delete** : Starting with root page, grab write (**W**) latch on child. Once child is locked, check if it is safe, in this case, at least half-full. (NOTE: for root page, we need to check with different standards) If child is safe, release **all** locks on ancestors.

Important: The write up only describe the basic concepts behind latch crabbing, before you start your implementation, please consult with lecture and textbook Chapter 15.10.

REQUIREMENTS AND HINTS

- You are **not allowed** to use a global scope latch to protect your data structure, that is to say, you can't lock the whole index and only unlock the latch when operations are done. We will check grammatically and manually to make sure you are doing the latch crabbing in the right way.
- We have provided the implementation of read-write latch (`src/include/common/rwmutex.h`). And have already added helper functions under page header file to acquire and release latch (`src/include/page/page.h`).
- We will not add any **mandatory** interfaces in the B+Tree index. You can write any private/helper functions in your implementation as long as you keep all the original public interfaces intact for test purpose.
- For this task, you have to use the passed in pointer parameter called **transaction** (`src/include/concurrency/transaction.h`). It provides methods to store the page on which you have acquired latch while traversing through B+ tree and also methods to store the page which you have deleted during **Remove** operation. Our suggestion is to look closely at the **FindLeafPage()** method within B+ tree, you may wanna twist your previous implementation(You may need to change to **return value** for this method) and add the logic of latch crabbing within this particular method.
- The return value for **FetchPage()** in buffer pool manager is a pointer that points to a Page instance(`src/include/page/page.h`). You can grab a latch on **Page** , but you cannot grab a latch on B+Tree node (neither internal node nor leaf node).

COMMON PITFALLS

- Think carefully about the order and relationship between **UnpinPage(page_id, is_dirty)** method from buffer pool manager class and **UnLock()** methods from page class. You have to release the latch on that page **BEFORE** you unpin the same page from the buffer pool.

bottom) to avoid possible deadlock situation.

- One of the corner case is that when insert and delete, the member variable **root_page_id** (`src/include/index/b_plus_tree.h`) will also be updated. It is your responsibility to protect from concurrent update of this shared variable(hint: add an abstract layer in B+ tree index, you can use `std::mutex` to protect this variable)

SETTING UP YOUR DEVELOPMENT ENVIRONMENT

Download the project source code [here](#). This version has additional files that were added after project #2 so you need to update your local copy.

Make sure that your source code has the following **VERSION.txt** file:

```
Created: Oct 26 2017 @ 18:01:53
Last Commit: fc181c8abb34ddb964c7f75cf70a16d994a2c70a
```

TESTING

You can test the individual components of this assignment using our testing framework. We use [GTest](#) for unit test cases. You can compile and run each test individually from the command-line:

```
cd build
make b_plus_tree_concurrent_test
./test/b_plus_tree_concurrent_test
```

In the `b_plus_tree_concurrent_test`, we will first test separately with insert/remove operations and then spawn threads that execute insert and remove at the same time.

```
cd build
make lock_manager_test
./test/lock_manager_test
```

Important: These tests are only a subset of the all the tests that we will use to evaluate and grade your project. You should write additional test cases on your own to check the complete functionality of your implementation.

DEVELOPMENT HINTS

Instead of using `printf` statements for debugging, use the `LOG_*` macros for logging information like this:

```
LOG_INFO("# Leaf Page: %s", leaf_page->ToString().c_str());
LOG_DEBUG("Fetching page %d", page_id);
```

To enable logging in your project, you will need to reconfigure it like this:

```
cd build
cmake -DCMAKE_BUILD_TYPE=DEBUG ..
make
```

The different logging levels are defined in `src/include/common/logger.h`. After enabling logging, the logging level defaults to `LOG_LEVEL_INFO`. Any logging method with a level that is equal to or higher than

Using [assert](#) to force check the correctness of your implementation. For example, when you try to delete a page, the `page_count` must be equal to 0. And when you try to unpin a page, the `page_count` must be larger than 0.

Using a relatively small value of page size at the beginning test stage, it would be easier for you to check whether you have done the delete and insert in the correct way. You can change the page size in configuration file (`src/include/common/config.h`).

GRADING RUBRIC

Each project submission will be graded based on the following criteria:

Does the submission successfully execute all of the test cases and produce the correct answer?

Does the submission execute without any memory leaks?

Note that we will use additional test cases that are more complex and go beyond the sample test cases that we provide you.

LATE POLICY

See the [late policy](#) in the syllabus.

SUBMISSION

After completing the assignment, you can submit your implementation of to Autolab:

<https://autolab.andrew.cmu.edu/courses/15445-f17>.

You only need to include the following files:

- Every file for Project 1 (6 in total)
- Every file for Project 2 (10 in total)
- `src/concurrency/lock_manager.cpp`
- `src/include/concurrency/lock_manager.h`

You can submit your answers as many times as you like and get immediate feedback. Your score will be sent via email to your Andrew account within a few minutes after your submission.


COLLABORATION POLICY

Every student has to work individually on this assignment.

Students are allowed to discuss high-level details about the project with others.

Students are **not** allowed to copy the contents of a white-board after a group meeting with other students.

Students are **not** allowed to copy the solutions from another colleague.

 **WARNING:** All of the code for this project must be your own. You may not copy source code from other students or other sources that you find on the web. Plagiarism **will not** be tolerated. See CMU's [Policy on Academic Integrity](#) for additional information.

