

⚠ **NOTICE:** This is an archived version of the course. [Click here](#) to view the latest offering.



PROJECT #4 - CONCURRENCY CONTROL

⚠ **Do not post your project on a public GitHub repository.**

OVERVIEW

The fourth programming project is to implement a **lock manager** in your database system and then use it to support concurrent query execution. A lock manager is responsible for keeping track of the **tuple-level locks** issued to transactions and supporting shared & exclusive locks granted and released appropriately based on the isolation levels.

The project is comprised of the following three tasks:

- [Task #1 - Lock Manager](#)
- [Task #2 - Deadlock Detection](#)
- [Task #3 - Concurrent Query Execution](#)

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

Release Date: Nov 18, 2020

Due Date: Dec 13, 2020 @ 11:59pm

PROJECT SPECIFICATION

Like the previous projects, 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, it will break the test code that we will use to grade your assignment, and you will 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 except the concurrent B+ tree; we will **not** provide solutions or binary files.

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 structure about the locks currently held by active transactions. Transactions then issue lock requests to the LM before they are allowed to access a data item. The LM will either grant the lock to the calling transaction, block 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** and **Executor** classes will use your LM to acquire locks on tuple records (by record id **RID**) whenever a transaction wants to access/modify a tuple.

isolation level. Please refer to the [lecture slides](#) about the details.

In the repository, we are providing you with a **Transaction** context handle (`include/concurrency/transaction.h`) with an isolation level attribute (i.e., `READ_UNCOMMITTED`, `READ_COMMITTED`, and `REPEATABLE_READ`) and information about its acquired locks. The LM will need to check the isolation level of transaction and expose correct behavior on lock/unlock requests. Any failed lock operation should lead to an `ABORTED` transaction state (implicit abort) and throw an exception. The transaction manager would further catch this exception and rollback write operations executed by the transaction.

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). This should also abort the transaction and return false if another transaction is already waiting to upgrade their lock.
- **Unlock(Transaction, RID)**: Unlock the record identified by the given record id that is held by the transaction.

The specific locking mechanism taken by the lock manager depends on the transaction isolation level. 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. We also recommend to review the isolation level concepts since the implementation of these functions shall be compatible with the isolation level of the transaction that is making the lock/unlock requests. 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 isolation level concepts covered in lectures to make sure your implementation satisfies the requirement.

ADVICE FROM THE TAS

- While your Lock Manager needs to use deadlock detection, we recommend implementing your lock manager first without any deadlock handling and then adding the detection mechanism after you verify it correctly locks and unlocks when no deadlocks occur.
- You will need some way to keep track of which transactions are waiting on a lock. Take a look at **LockRequestQueue** class in `lock_manager.h`.
- You will need some way to notify transactions that are waiting when they may be up to grab the lock. We recommend using `std::condition_variable`.
- Although some isolation levels are achieved by ensuring the properties of strict two phase locking, your implementation of lock manager is only required to ensure properties of two phase locking. The concept of strict 2PL will be achieved through the logic in your executors and transaction manager. Take a look at **Commit** and **Abort** methods there.
- 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 (Hint: Look at the methods in `transaction.h`).
- You should also keep track of the shared/exclusive lock acquired by a transaction using `shared_lock_set_` and `exclusive_lock_set_` so that when the **TransactionManager** wants to commit/abort a transaction, the LM can release them properly.
- Setting a transaction's state to `ABORTED` implicitly aborts it, but it is not explicitly aborted until **TransactionManager::Abort** is called. You should read through this function to understand what it does, and how your lock manager is used in the abort process.

TASK #2 - DEADLOCK DETECTION

More precisely, this means that a background thread should periodically build a waits for graph **on the fly** and break any cycles.

The graph API you must implement and use for your cycle detection along with testing is the following:

- `AddEdge(txn_id_t t1, txn_id_t t2)`: Adds an edge in your graph from t1 to t2. If the edge already exists, you don't have to do anything.
- `RemoveEdge(txn_id_t t1, txn_id_t t2)`: Removes edge t1 to t2 from your graph. If no such edge exists, you don't have to do anything.
- `HasCycle(txn_id_t& txn_id)`: Looks for a cycle by using the **Depth First Search (DFS)** algorithm. If it finds a cycle, `HasCycle` should store the transaction id of the **youngest** transaction in the cycle in `txn_id` and return true. Your function should return the first cycle it finds. If your graph has no cycles, `HasCycle` should return false.
- `GetEdgeList()`: Returns a list of tuples representing the edges in your graph. We will use this to test correctness of your graph. A pair (t1,t2) corresponds to an edge from t1 to t2.
- `RunCycleDetection()`: Contains skeleton code for running cycle detection in the background. You should implement your cycle detection logic in here.

How you choose to implement the graph is up to you as long as your implementation supports the above API. Those functions are what we will use to test your code.

REQUIREMENTS AND HINTS

- Your background thread should build the graph on the fly every time it wakes up. You should **not** be maintaining a graph, it should be built and destroyed every time the thread wakes up.
- Your DFS Cycle detection algorithm **must** be deterministic. In order to do achieve this, you must always choose to explore the lowest transaction id first. This means when choosing which **unexplored** node to run DFS from, always choose the node with the lowest transaction id. This also means when exploring neighbors, explore them in sorted order from lowest to highest.
- When you find a cycle, you should abort the **youngest** transaction to break the cycle by setting that transactions state to ABORTED.
- When your detection thread wakes up, it is responsible for breaking **all** cycles that exist. If you follow the above requirements, you will always find the cycles in a **deterministic** order. This also means that when you are building your graph, you should **not** add nodes for aborted transactions or draw edges to aborted transactions.
- Your background cycle detection algorithm may need to get a pointer to a transaction using a `txn_id`. We have added a static method `Transaction* GetTransaction(txn_id_t txn_id)` to let you do that.
- You can use `std::this_thread::sleep_for` to order threads to write test cases. You can also tweak `CYCLE_DETECTION_INTERVAL` in `common/config.h` in your test cases.

ADVICE FROM THE TAS

- Remember that a waits for graph is a **directed** graph.
- A waits for graph draws edges when a transaction is waiting for another transaction. Remember that if multiple transactions hold a **shared** lock, a single transaction may be waiting on multiple transactions.
- When a transaction is aborted, make sure to set the transaction's state to **ABORTED** and throw an exception in your lock manager. The transaction manager will take care of the explicit abort and rollback changes.
- A transaction waiting for a lock may be aborted by the background cycle detection thread. You must have a way to notify waiting transactions that they've been aborted.

TASK #3 - CONCURRENT QUERY EXECUTION

During concurrent query execution, executors are required to lock/unlock tuples appropriately to achieve the isolation level specified in the corresponding transaction. To simplify this task, you can ignore concurrent index execution and just focus on table tuples.

You will need to update the `Next()` methods of some executors (sequential scans, inserts, updates, deletes, nested loop joins, and aggregations) implemented in Project 3. Note that transactions would abort when lock/unlock fails. Although there is no requirement of concurrent index execution, we still need to undo all previous write operations on both table tuples and indexes

You should not assume that a transaction only consists of one query. Specifically, this means a tuple might be accessed by different queries more than once in a transaction. Think about how you should handle this under different isolation levels.


More specifically, you will need to add support for concurrent query execution in the following executors:

- `src/execution/aggregation_executor.cpp`
- `src/execution/delete_executor.cpp`
- `src/execution/insert_executor.cpp`
- `src/execution/nested_loop_executor.cpp`
- `src/execution/seq_scan_executor.cpp`
- `src/execution/update_executor.cpp`

INSTRUCTIONS

SETTING UP YOUR DEVELOPMENT ENVIRONMENT

See the [Project #0 instructions](#) on how to create your private repository and setup your development environment.

 You must pull the latest changes on our BusTub repo for test files and other supplementary files we have provided for you. Run ``git pull``.

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 lock_manager_test
make transaction_test
./test/lock_manager_test
./test/transaction_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

To ensure your implementation does not have memory leak, you can run the test with [Valgrind](#).

```
cd build
make concurrency_control_test
valgrind --trace-children=yes \
  --leak-check=full \
  --track-origins=yes \
  --soname-synonyms=somalloc=jemalloc \
  --error-exitcode=1 \
  --suppressions=../third_party/valgrind/valgrind.supp \
  ./test/concurrency_control_test
```

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
```

`LOG_ERROR`) will emit logging information.

Use `assert` to force check the correctness of your implementation.

? Please post all of your questions about this project on Piazza. Do **not** email the TAs directly with questions. The instructor and TAs will **not** debug your code. Particularly with this project, we will not debug deadlocks.

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 Gradescope.

You must include every file for Project #1, #2, and #3 in your submission zip file, as well as the following files:

- `src/concurrency/lock_manager.cpp`
- `src/include/concurrency/lock_manager.h`

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.

Last Updated: Dec 06, 2020

