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 transaction 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 Prevention
- Task #3 - Concurrent Query Execution

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

Release Date: Nov 15, 2021
Due Date: Dec 05, 2021 @ 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 sh **not** modify the signatures for the pre-defined functions in these classes. If you do this, it will break the test code that we w use to grade your assignment, and you will end up getting no credit for the project. If a class already contains certain mem variables, you should **not** remove them. But you may add private helper functions/member variables to these classes in or to correctly realize the functionality.

The correctness of this project depends on the correctness of your implementation of previous projects; we will **not** provi 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 t locks currently held by active transactions. Transactions then issue lock requests to the LM before they are allowed to acc data item. The LM will either grant the lock to the calling transaction, block that transaction, or abort it.

access/modify a tuple.

This task requires you to implement a tuple-level LM that supports the three common isolation levels : READ_UNCOMMI READ_COMMITTED, and REPEATABLE_READ. The Lock Manager should grant or release locks according to a transactio 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 ar isolation level attribute (i.e., READ_UNCOMMITED, READ_COMMITTED, and REPEATABLE_READ) and information abc 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. 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 avai 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 or 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 blo 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 sho be blocked on waiting and should return true when granted. Return false if transaction is rolled back (aborts). This sho 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 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 th isolation level of the transaction that is making the lock/unlock requests. You have the freedom of adding any necessary da structures in `lock_manager.h`. You should consult with Chapters 15.1-15.2 in the textbook and isolation level concepts co in lectures to make sure your implementation satisfies the requirement.

ADVICE FROM THE TAS

- While your Lock Manager needs to use deadlock prevention, we recommend implementing your lock manager first wit any deadlock handling and then adding the prevention mechanism after you verify it correctly locks and unlocks when deadlocks occur.
- You will need some way to keep track of which transactions are waiting on a lock. Take a look at `LockRequestQueue` clas `lock_manager.h` .
- What does `LockUpgrade` translate to in terms of operations on the `LockRequestQueue` ?
- You will need some way to notify transactions that are waiting when they may be up to grab the lock. We recommend u `std::condition_variable`
- Although some isolation levels are achieved by ensuring the properties of strict two phase locking, your implementatio lock manager is only required to ensure properties of two phase locking. The concept of strict 2PL will be achieved thr the logic in your executors and transaction manager. Take a look at `Commit` and `Abort` methods there.

- 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 you manager is used in the abort process.

TASK #2 - DEADLOCK PREVENTION

If your lock manager is told to use prevention, your lock manager should use the **WOUND-WAIT** algorithm for deciding w transactions to abort.

When acquiring a lock, you will need to look at the corresponding `LockRequestQueue` to see which transactions it would b waiting for.

ADVICE FROM THE TAS

- Read through the slides carefully on how Wound-Wait is implemented.
- When you abort a transaction, make sure to properly set its state using `SetState`
- If a transaction is upgrading (waiting to acquire an X-lock), it can still be aborted. You must handle this correctly.
- 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 another thread due to the wound-wait prevention policy. You must l 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 lev specified in the corresponding transaction. To simplify this task, you can ignore concurrent index execution and just focus table tuples.

You will need to update the `Next()` methods of some executors (sequential scans, inserts, updates, deletes, nested loop jc and aggregations) implemented in Project 3. Note that transactions would abort when lock/unlock fails. Although there is requirement of concurrent index execution, we still need to undo all previous write operations on both table tuples and inc appropriately on transaction abort. To achieve this, you will need to maintain the write sets in transactions, which is requir by the `Abort()` method of transaction manager.

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

INSTRUCTIONS

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 v
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. `
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 shoulc
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 lock_manager_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/lock_manager_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
```

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

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

LATE POLICY

See the late policy in the syllabus.

SUBMISSION

After completing the assignment, you can submit your implementation 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' **Policy on Academic Integrity** for additional information.

Last Updated: Jan 07,