PROJECT #4 - LOGGING & RECOVERY

OVERVIEW

The fourth programming project is to implement **Logging** and **Recovery** mechanism in your database system. The first task is to implement write ahead logging (WAL) under **No-Force/Steal** buffering policy and log every single page-level write operation and transaction command.

The project is comprised of the following two tasks:

- Task #1 - Log Manager
- Task #2 - System Recovery

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: Nov 15, 2017
Due Date: Dec 06, 2017 @ 11:59pm

PROJECT SPECIFICATION

Like the previous 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** give solutions or binary files. However, this project may differ from the last ones in several aspects. Instead of implementing a stand-alone component, it requires you to explore a code base that we have already provided and find the **right place** to add your logic. It focuses on the insert/delete/search operations on table heap (`include/table/table_heap.h`) as well as interactions between log manager, buffer manager, and transaction manager.

TASK #1 - LOG MANAGER

performed by committed transactions are reflected in the database (perhaps during the course of recovery actions after a crash). It can also help us ensure that no modifications made by an aborted or crashed transaction persist in the database. The most widely used structure for recording database modifications is the log. The log is a sequence of log records, recording all the update activities in the database. We have provided with you the basic structure of log record(`include/logging/log_record.h`) and corresponding helper methods.

In your implementation, there will be a global `LogManager` object for the entire system (similar to your `BufferPoolManager`). The `TablePage` class will explicitly create a log record (before any update operations) and invoke `SerializeLogRecord` method of `LogManager` to write it into log buffer when the global variable `ENABLE_LOGGING` (`include/common/config.h`) flag is set to be true. We recommend you to read this article to refresh your C++ concurrency knowledge. More detailed documentation about condition variable is available here.

CHANGES SINCE PROJECT #3

This list contains classes we have changed since the previous project. You may wanna read them carefully and adjust your original implementation based on the updates in order to pass test cases.

- Global Variables: We have added a global variable called `ENABLE_LOGGING` within config.h file. Only when the variable is set to be true can you enable all the logging functionalities. You may need to explicitly set it to be false in order to pass some of the previous test cases. The `LOG_TIMEOUT` is a constant variable defined within config.h file, for every `LOG_TIMEOUT` seconds, your Log Manager needs to execute a flush operation (more details int the next section).
- Buffer Pool Manager: We modified the constructor to take in references to the system's DiskManager and LogManager as input parameters. The default value for log_manager is nullptr, and log_manager is only valid when `ENABLE_LOGGING` is set to be true. We have also removed the `FlushAllPage()` function interface.
- Disk Manager: The DiskManager now creates the database file as well as log file inside its constructor. We have also provided separate helper functions `WriteLog` and `ReadLog` to support accessing the log file.
- BPlusTreePage: We added a member variable `lsn_` to record page log sequence number. You do not need to update lsn within your index implementation, we only test logging and recovery functionalities on table page level. But you may need to double check your init method for both `BPlusTreeInternalPage` and `BPlusTreeLeafPage` if you are hard-coded metadata size.
- Page: We added `GetLSN` and `SetLSN` helper functions. Remember log sequence number is a 4 byte int32_t variable that is stored within page data_ array.
- Transaction: We added `GetPrevLSN` and `SetPrevLSN` helper functions. Each transaction is responsible for maintaining previous log sequence number for undo purpose in recovery (detailed information, please consult with textbook Chapters 16.8).
- LogRecord: We have provided the implementation of physiological LogRecord that support different type of write operations within database system. Each log type corresponds to a write operation within `TablePage` class (`page/table_page.h`, please make sure you understand the record structure before implementation.

REQUIREMENTS AND HINTS

include/logging/table_page.h) plus the TransactionManager class (concurrency/transaction_manager.cpp and include/concurrency/transaction_manager.h) plus your original implementation of BufferPoolManager class(table/table_page.cpp and include/logging/table_page.h). You will need to complete the following functionalities:

- Inside RunFlushThread function in Log manager, you need to start a **separate background** thread which is responsible for flushing the logs into disk file. The thread is triggered every LOG_TIMEOUT seconds or when the log buffer is full. Since your Log Manager need to perform asynchronous I/O operations, you will maintain **two** log buffers, one for flushing (We will call it as flush_buffer) one for concurrently appending log records (We will call it as log_buffer). And you may consider swap buffers under following **three** situations. (1) When log_buffer is full. (2) When LOG_TIMEOUT is triggered. (3) When buffer pool is going to evict a dirty page from LRU replacer.
- Your Log Manager will integrate the group commit feature. Motivation behind group commit is to amortize the costs of each fsync() over multiple commits from multiple parallel transactions. If there are say 10 transactions in parallel trying to commit, we can force all of them to disk at once with a single fsync() call, rather than do one fsync() for each. This can greatly reduce the need for fsync() calls, and consequently greatly improve the commits-per-second throughput. Within TransactionManager , whenever you call Commit or Abort method, you need to make sure your log records are permanently stored on disk file before release the locks. But instead of **forcing** flush, you need to wait for LOG_TIMEOUT or other operations to implicitly trigger the flush operations.
- Before your buffer pool manager evicts a dirty page from LRU replacer and write this page back to db file, it needs to flush logs up to pageLSN . You need to compare persistent_lsn_ (a member variable maintains by Log Manager) with your pageLSN . However unlike group commit, buffer pool can force log manager to flush log buffer, but still needs to wait for logs to be permanently stored before continue.
- Add corresponding logics within TablePage class methods to deal with run-time WAL logging. You need to (1) explicitly create a log record (include/logging/log_record.h) (2) invoke SerializeLogRecord method of Log Manager to write it into log_buffer when the global variable ENABLE_LOGGING (include/common/config.h) is set to be true. (3) Update prevLSN for current transaction. (4) Update LSN for current page

Important: You should first take a look at the files we mention in this and previous sections to become familiar with the APIs and member variables we provide. You have the freedom of adding any necessary data structures in log_manager.h and log_record.h. You should consult with Chapters 16.8 in the textbook and make sure that your implementation follows the ARIES.(Except for the check-pointing part). Since Log Manager need to deal with background thread and thread synchronization stuff, we recommend you to take a look at Future and Promise.

TASK #2 - SYSTEM RECOVERY

The next part of the project is to implement the ability for the DBMS to recover its state from the log file.

REQUIREMENTS AND HINTS

The recovery process for our database system is pretty straightforward. Since we do not enable check-pointing, there is no need for analysis pass. The only file you need to modify for this task is the LogRecovery

- `DeserializeLogRecord` : Deserialize and reconstruct a log record from log buffer. If the return value is true then Deserialization is successful, otherwise log buffer may contain incomplete log record.
- `Redo` : Redo pass on TABLE PAGE level(`include/table/table_page.h`). Read the log file from the beginning to end (you must prefetch log records into buffer to reduce unnecessary I/O operations), for each log record, redo the action **unless** page is already more up-to-date than this record. Also build `active_txn_ table` and `lsn_mapping_ table` along the way.
- `Undo` : Undo pass on TABLE PAGE level(`include/table/table_page.h`). Iterate through `active_txn_ table` and undo every operations within each transaction. You **DON NOT** need to worry about crash during recovery process, thus no complementary logging is needed.

Important: Our suggestion is to first consult with Chapters 16.8 in the textbook to make sure that you understand the whole process of recovery and what is redo/undo pass trying to do. Then figure out each write operation within class `TablePage` and what is the corresponding complementary operation. (e.g To undo Insert operation, you need to call ApplyDelete function instead of MarkDelete)

SETTING UP YOUR DEVELOPMENT ENVIRONMENT

Download the project source code here. This version has additional files that were added after project #3 so you need to update your local copy.

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

```
Created: Nov 21 2017 @ 00:28:55
Last Commit: be0f6dd92b4c3eaa52a6456bd8847773ab65b3ed
```

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 log_manager_test
./test/log_manager_test
```

In the log_manager_test, we will first start the system, create a table, populate several tuples and then shut down the system. When the system restarts and has completed the recovery phases, it should be back to the consistent state before crash.

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("# RID: %s", rid->ToString().c_str());
LOG_DEBUG("Evict page %d", page_id);
```

```
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 `LOG_LEVEL_INFO` (e.g., `LOG_INFO`, `LOG_WARN`, `LOG_ERROR`) will emit logging information.

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 logging & recovery 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)
- Every file for Project 3 (2 in total)
- `src/logging/log_manager.cpp`
- `src/include/logging/log_manager.h`
- `src/logging/log_recovery.cpp`
- `src/include/logging/log_recovery.h`
- `src/include/logging/log_record.h`
- `src/page/table_page.cpp`
- `src/concurrency/transaction_manager.cpp`

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: Mar 06, 2018

CARNEGIE MELLON
DATABASE GROUP