Database Systems

Instructor: Hao-Hua Chu

Fall Semester, 2004

Assignment 10: Transaction Simulation & Crash Recovery

Deadline: 23:59 Jan. 5 (Wednesday), 2005
This is a group assignment, and at most 2 students per group are allowed.

Cheating Policy: If you are caught cheating, your grade is 0.
Late Policy: You may hand in your late assignment before 23:59 on Thursady (1/6/2005) for 80% of original grade, or before 23:59 on Friday (1/7/2005) for 70%. We will not accept any assignment submissions after Friday (1/7/2005).
Demo Time: 1/12 (Wednesday) 10:00 ~ 11:30,
1/14 (Friday) 13:30~14:30
If you are not available on those time period, please let me know. I will announce another time period. The deadline of this demo will be on 1/17 before 17:00.

1. Introduction

In this assignment you will complete certain parts of a database engine. The database is modeled very simply, and runs by executing a series of transaction operations given to it by the user. Each transaction performs a series of reads and writes on the pages of the database, and can commit or abort at any time.

The simulator keeps a recovery manager module that keeps a log of the database's activity. At any point in time, the database may crash (specifically, when it encounters a special command to induce a crash). The recovery manager then performs a restart to restore the database to a correct state, using the Aries recovery algorithm.

2. The Transaction Simulator

The transaction simulator works just like a mini-database. There is a series of pages stored in a file on disk, accessed through a buffer. Transactions are modeled as a sequence of reads and writes to the pages of the database. A log is kept of all changes to database pages, with WAL used to insure that committed transactions are fully represented by the log as written to stable storage. Checkpoints can be inserted at will.

This assignment uses an application called MARS which is a GUI for testing the simulator. The input comes from .mars files that can be found in the folder Tests. These files can be edited using any simple text editor. The syntax for commands to the simulator is discussed in Getting Started with the Simulator.

A breakdown of the major components follows:

- BufMgr - This module operates much like the buffer you implemented for project 1. When a page is to be read or written to, a pinPage() request is made, and the page is, if not already present, loaded into the buffer pool. On request, one or all pages can be flushed. Pages are released with an unpinPage() call. The buffer for this project is somewhat small, being intended to illustrate the role of the buffer in crash recovery.
- Xaction_table - This keeps track of all the transactions currently active in the database. It also keeps information on each active transaction, namely the oldest and most recent log sequence numbers (LSN's) for each transaction.
- The log subsystem:
    - log - This is the log that is used in WAL while the transactions are executing. It provides simple sequential access to the log files for reading purposes, and can append new records to the end of the log. **The log records are of uniform size**, and the log does not concern itself with the type of log being written.
    - logrecord - This represents a single log entry. A special field is kept in each record to identify the variety of log record (commit, update, etc.) that it is. It contains a generic data buffer, which holds the information specific to each type of log record.
    - masterlog - This keeps track of the checkpointing, so that the recovery process will not need to go back too far to reconstruct the database at the time of the crash.

- LogData structures - These classes represent each type of update, and the information associated with it like prevLSN, xaction_id, and the page affected. Each Log Data structure is stored in the data field of a log record, and should be accessed by typecasting that data to the appropriate LogData type (UPD, CLR, ABORT).
- Recovery Manager - consists of several related modules for logging & recovery procedures. While the database is running, each transaction has its own recovery manager that is responsible for logging its actions. Only one recovery module, though, is necessary for performing crash recovery. The pieces of the manager are:
    - The logging functionality (logfunc.cpp) - This translates write requests by transactions into Update records that are written to the log, and creates CLR records whenever a process aborts on its own.
    - rollback.cpp - This performs a rollback on a process that has just aborted on its own, undoing the changes and making sure that none of them persist even after a crash.
    - restart.cpp - This is the part responsible for bringing the database up to a consistent state following a crash. It performs the three phases of the Aries recovery algorithm based on the information written to the log.
    - Checkpoint - This generates checkpoints and writes them to the log, extracting the information from the Xaction table and getting the Dirty Page Table from the Buffer.
    - Recovery DirtyPageTable - this is the list of possibly dirty pages that is built during the analysis phase of recovery.
    - Recovery XactTable - this is the list of active transactions that is built during the analysis phase of recovery.

The recmgr_tab.c module is responsible for parsing the input. It is computer generated, and not very fit for modification. Don't worry too much about what it does; just know that it takes one command at a time from the standard input (which we might have redirected to point to a file) and converts it into a database operation.

The handle.cpp module contains the standalone functions for performing the commands. There's one for each operation, and are the "top-level" functions that are first called when an operation is done. The functions in handle.cpp are the functions that will call the Recovery Manager functions that you will be finishing, and will pass in the relevant data about the operations.

The complete code is available in the \\goose\cs432-fall99\A7 and it consists of the complete project file. You can open it by double clicking the Mars.dsw file. All you need to do is fill in gaps inside two source files (logfunc.cpp and restart.cpp) in the project.

When you have included your modification and are ready to test your code, you can run the Mars.exe file generated by visual studio when you compile the project. When you are generating the Mars.exe file it is recommended that you build the release version of the project. This can be done by opening the Build menu, selecting "Set Active Configuration", and choose "Release". The .exe file can then be found in the "Release" folder. Simply double click it to start the transaction manager.

3. Your Task

You will implement various pieces of the recovery module. Specifically, you will implement the functionality to handle the most basic and common of transactions, the write (WriteUpdateLog()). You must add code to the logging mechanism so that writes (also called *updates*) to the pages of the DB are reflected in the log, and then implement those parts of the restart mechanism that deal specifically with those log entries to restore the database to a consistent state.

The code you will write belongs in these modules:

- logfunc.cpp - You should complete this function:
    - WriteUpdateLog() - Given the information about a given update, generate an update log record, and update all affected information in the rest of the database.

- restart.cpp - You should complete these four main functions:
    - restart_analysis - This scans the log record forward from the last checkpoint, building up information about the database at the time of the crash. Fill in the code executed when the record being looked at is an update record, updating the Recovery Xaction table and Recovery Dirty Page Table being rebuilt.
    - redo_update - This is the function for redoing a single action. Add the code for redoing an update (UPD) record, extracting the necessary information from the log record data and performing the update.

- restart_redo - This is the second phase of recovery, and restores the database to its pre-crash state. You should implement the code that handles each update record, calling redo_update if the action needs to be retaken. You should consider the pageLSN stored with each page to determine the necessity of repeating the action, and update the Recovery Dirty Page Table where necesary.
- restart_undo - This third phase of recovery aborts all transactions active at the time of the crash, scanning the log backwards and undoing the actions of that transaction. Implement the code that handles the case for undoing an update record. You'll need to work with the Recovery Xaction table, and you should generate a CLR to be written to the log.
  Hint: This is essentially a rollback of the transaction, so look at the code that normally handles a rollback when a transaction aborts.
- You should also finish three small methods concerning the control of the recovery process:
- findRedoLsn() - identifying the earliest lsn from which the redo process should start.
- keepPerformingUndo() - determining when to stop the undo process.
- findNextUndoLsn() - identifying the next log record to be undone in the undo process.

4. Reference

You could find the executable file in Exe_4_Test.

The following pages provide more detailed explanations about the classes and types that will be useful for this assignment.

- Class log
- Class lsn_t
- Class logRecord
- type LOG_TYPE
- Class UpdateLogRec
- Class CLRLogRec
- Class CommitLogRec
- Class AbortLogRec

- Class CkptBeginLogRec
- Class CkptEndLogRec
- Class RecoveryMgr
- Class Xaction_table

5. Submission

Submit your program (***logfunc.cpp and restart.cpp***) per group via email to TA, yfhuang@csie.ntu.edu.tw. The email subject must be [DBMS] ASS10 – members' ID. You can submit your program many times, and TA will use that latest one to grade. In addition, each group has to hand in a one-page report to illustrate how you implement this assignment.