SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.043 Database Lab 3 Report

Name	Student ID
Matthaeus Choo Zhi Jie (Zhu Zhijie)	1008103
Lee Jing Kang	1008240
Chong Juo Ru Faustina	1007967

50.043 Database System and Big Data ~ Lab 3

Summary and Design decisions

Exercise 1, 2 & 5 ~ Lock acquisition and release in Bufferpool

In bufferpool.java, GetPage() function was modified to acquire a lock from lock manager before returning pages to ensure all page access goes through proper lock acquisition and proper permissions checking. UnsafeReleasePage() was implemented to release both read and write lock. The two-phase locking protocol is implemented through shared and exclusive locks along with a deadlock detection mechanism.

Exercise 3 ~ NO STEAL Buffer management

The NO STEAL policy was implemented by modifying evictPage() to iterate through the bufferpool pages, only evicting clean pages. In the case that no clean pages are found, a DbException is thrown to prevent the eviction of dirty pages from uncommitted transactions. Uses first-found clean page selection for eviction.

Exercise 4 ~ Transaction Completion

TransactionComplete() was implemented with commit/abort logic; On commit, the function calls flushPages(tid) to write all transaction's dirty pages to disk. On abort, it restores pages to before-image state by replacing dirty pages in bufferpool. And always releases all transaction locks in finally block regardless of commit/abort outcome to guarantee lock cleanup, preventing deadlock from unreleased locks

Deadlock Detection and Prevention

Created a seperate Lockmanger class that manges lock state using a concurrentHashMap data structure. LockManager uses four main data structures;

1. PageLocks maps PageId to sets of lock objects
2. TransactionLocks maps TransactionId to sets of Locks held
3. WaitingQueue maintains FIFO queues of lock requests per page
4. Graph is a wait-for graph that tracks transaction dependencies for deadlock detection

The implemented data structures provide efficient lookup and maintenance operations and ensures thread safety through synchronized blocks. The design checks if the lock can be granted immediately, otherwise it adds the transaction to the wait-for graph, performing cycle detection, and enqueues the request.

The cycle detection is done using a depth-first search algorithm with recursion stack to detect cycles in the wait for graph. This method provides more precise detection when detecting a deadlock as it only aborts transactions when actual deadlocks exists, minimizing rollbacks. The FIFO queue ensures transactions acquire locks in request order, preventing starvation.

However, the cost to maintain the graph is high, as the graph must update wait-for relationships after every lock release, requiring iteration through waiting queues. The deadlock detection maintain many data structures to function, causing memory overhead that grows with transaction count.

Missing/Incomplete Elements Of Our Code/Changes Made To The API

All Lab 1, 2 and 3 code have been implemented. No Changes made to the API