Database Systems- Project

Lab 3: SimpleDB Transcations

Chirag Shivakumar 1004996, Devanshi Joshi 1005162, Suthar Prachi Jayesh 1005421

# Design Decisions made during the Project

## Modifying BufferPool.java – Transactions, Locking and Concurrency Control

The goal of this lab is to create a simple locking-based transaction system in SimpleDB. The lab requires the addition of lock and release methods in the code, as well as code to monitor the holds held by each transaction. Students must use two-phase locking to ensure transaction separation and atomicity. The lab will demonstrate how to implement transaction support in a database and how to use locking to manage concurrency. The job entails adding lock and release methods to the code as well as monitoring the locks held by each transaction to ensure that transactions are executed consistently and that the database remains consistent even when multiple transactions are executed concurrently. The concepts of transactions, locking, and concurrency control are explained, with a focus on the use of strict two-phase locking.

### Exercise 1 and 2- Transactions, The ACID Properties, Recovery and Buffer Management. Granting Locks

### A transaction in a database management system is a set of activities that are performed atomically, either all of them or none of them. Transaction atomicity, consistency, isolation, and durability are all guaranteed by the ACID properties. SimpleDB preserves these properties through the use of strict two-phase locking and buffer management rules. It is recommended to lock at the page level, and a LockManager class keeps track of locks held by transactions and provides locks when requested. Strict two-phase locking ensures that appropriate locks are obtained before handling objects and that locks are not relinquished until the transaction is completed. BufferPool in SimpleDB obtains and releases locks using the getPage(), unsafeReleasePage(), and holdsLock() methods, whereas HeapFile.insertTuple() and HeapFile.deleteTuple() methods use the BufferPool.getPage() method to handle pages.

### Exercise 2- Lock Lifetime

### The second exercise demonstrates how to use lock acquisition and release across SimpleDB. The job entails making sure that reading, writing, and removing tuples work properly, as well as paying close attention to obtaining and releasing locks when adding a new page to a HeapFile and searching for an empty space. The task entails taking into account potential race conditions with other transactions and ensuring that the code passes the LockingTest unit tests.

### Exercise 3- Implementing NO STEAL

### The exercise enforces a NO STEAL policy, which states that dirty pages may not be removed from the buffer pool and must be written to storage only after the transaction has completed. If the eviction policy requires the removal of a soiled page, another page must be selected. If all of the pages in the buffer pool are dirty, a DbException must be thrown. Be wary of any locks that transactions may already have on the expelled page and properly manage them.

### Exercise 4- Transactions

You must implement the purpose of TransactionId objects in SimpleDB as well as how the transactionComplete() method in BufferPool is used to perform or terminate transactions in this Exercise. On commit, use the transactionComplete() function to remove dirty pages associated with the transaction to disk, and on abort, use the transactionComplete() function to restore any changes made by the transaction to the on-disk state.

### Exercise 5- Deadlocks and Aborts

### The task implements deadlock detection and avoidance in SimpleDB's BufferPool.java. Deadlocks can occur when transactions wait for locks to be released, causing transactions to become trapped indefinitely. A simple timeout strategy may not be sufficient to break a deadlock. Cycle spotting in a wait-for graph before each lock request is one approach. When a deadlock is discovered, the system must decide how to resolve it, such as by aborting or restarting operations. To indicate a deadlock, the code should throw a TransactionAbortedException and pass the provided DeadlockTest.java to ensure that the system is working properly.

# Non-Trivial parts of code

## Additional Class Functions to Support the BufferPool.java

### LockManager.java and PageLock.java

The LockManager class manages locks on database buffer pool pages, allowing multiple transactions to view the same pages without interfering with each other. If another transaction is holding a write lock, the method waits and adds the waiting transaction to the waitMap, whereas the isDeadLock() function checks for deadlocks. If the page does not have a write lock, the procedure obtains a read lock for READ\_ONLY rights and a write lock for READ\_WRITE rights. It also converts a read lock to a write lock if the current operation is the only read lock on the page.

The LockManager class manages locks in a database buffer pool to ensure that multiple operations can access the same pages without interfering. It stores lock and event information in three hash maps and can obtain read or write locks based on permissions.

#### These methods are associated with the buffer pool because they are invoked whenever a transaction attempts to hold a page that is currently in the buffer pool. To ensure that the desired lock can be obtained without interfering with other transactions, the buffer pool consults with the LockManager classIf the buffer pool cannot obtain a lock quickly, it will wait until the lock is released before returning the desired page to the transaction.

#### holdsLock(TransactionId tid, PageId pid)

#### This method determines whether a transaction has a page lock. It first checks to see if the page ID exists in the pgLock database. If it is, it checks to see if the transaction has a lock on the page using the holdsLock() function of the PageLock object associated with the page ID.

#### acquireLock(TransactionId tid, PageId pid, Permissions perm)

#### This method obtains a page lock for the specified transaction ID and rights. If the page ID is not found in the pgLock map, a new PageLock object is created and appended to the existing one. The page ID is then appended to the transaction's collection of pages in the transacMap database.

#### releaseLock(TransactionId tid, PageId pid)

#### This method makes use of a transaction-held page lock. If the pgLock map contains the page ID, the lock is released by calling the releaseLock() function of the PageLock object associated with the page ID. In addition, the technique removes the page ID from the transaction's collection of pages in the transacMap map and notifies other transactions that are waiting for the lock.

# Justify any changes made to the API- None

# Any missing or incomplete elements in your code- None