CSE 444: Lab 3 Writeup

Linxing Preston Jiang Winter 2018

February 18, 2018

---

1. Lab 3 focuses on adding Transactions functionality to simpleDB. Specifically, we implemented NO STEAL (never evict dirty pages from the buffer pool if they are locked by an uncommited transaction), and FORCE (on transaction commit, force write pages to disk) for buffer pool management. In order to achieve this, we implemented our own `Lock` and `LockManager` which together handle acquiring/releasing both `SHARED` (read-only) and `EXCLUSIVE` (read-write) locks by different transactions on page granularity. Another important part of transcations is deadlock detection and resolution. For lab3, I implemented time-out limits for `acquire` so that a certain period time of blocking on `acquire` will be considered as deadlock and thus abort the transaction.

Main parts of lab3 include:

- `Lock`: We need this class to represent the two types of locks simpleDB uses: `SHARED` and `EXCLUSIVE`. A shared lock is acquired by read-only transactions, and thus can be shared between many read-only transactions; an exclusive lock is acquired by read-write transactions, and thus can only be acquired by at most one read-write transaction at a time.
- `LockManager`: We need this class as the manager for all locking-related actions in simpleDB. It is created within `BufferPool` and will be called to acquire locks when `getPage` is called and release locks when `transactionComplete` is called. Note that because of the design of simpleDB, `acquire` in `LockManager` should only need calling in `getPage` when simpleDB wants to interact with a page. For more about blocking and deadlock detection/resolution, see "design" part of this writeup.
- `BufferPool.transactionComplete`: We need this method to release all locks acquired by a transaction when it commits. Note that because we choose to implement `NO STEAL`, releasing locks should only happen at this method except aborting transactions.

2. I suggest adding a new unit test for the correctness of `BufferPool.flushPages`. More specifically, when flusing all the pages for committing a transaction, `BufferPool` should only flush pages which are dirtied by ***this*** transaction, not just all dirty pages which are possibly marked dirty by other transactions that are not committing. An extra test on this behavior would be helpful for debugging.

3. Design decisions:

(a) Lock

- Lock has two types: `SHARED` or `EXCLUSIVE`.
- Lock keeps a hash set of `TransactionIds`, mainly for shared locks to keep track of it is shared by which transactions.
- Lock is used to ensure **page** granularity.

(b) LockManager

- LockManager is implemented as Singleton, for there should be only one object of `LockManager` to manage all transactions, just like `BufferPool`.
- Lock keeps a hash map from `TransactionId` to a hash set of `PageIds` to keep track of which pages a certain transaction has locks on. It also keeps a hash map from `PageId` to `Lock`.
- There are a few conditions when `acquire` will not be a blocking call. They are:
  - When the lock on that page is not locked.

- When the lock is locked by this transaction itself (will upgrade a share lock to exclusive if this transaction is the only one which holds the lock)
- When the lock is locked, but it is a shared lock, and the transaction wants a shared lock.

- When `acquire` is a blocking call, the transaction will sleep for `500` milliseconds, and then it will try to acquire for the lock again. If the second try still fails, it will be considered as a deadlock condition, and so `acquire` throws a `TransactionAbortedException`.

(c) BufferPool, race conditions

- To avoid race conditions, `getPage` is synchronized.
- When inserting tuples and `HeapFile` needs to add a new page, it is possible that two transactions both find that a new page needed to be added after finishing scanning for empty slots. The current code will possibly make transactions add a new page each. The fix is to makej `HeapFile.insertTuple` synchronized so that only one transaction will perform scanning for empty slots. This is more efficient than making `BufferPool.insertTuple` synchronized because transactions which are modifying different `HeapFile` should not have this issue thus should be able to perform concurrently.

4. Changes to API: I did not make changes to the APIs given.

5. Missing elements: I believe I finished the entire Lab 3.