ACS-A3

Rikke Tophøj Heinemann (DTZ190)
Marie Elkjær Rødsgaard (DCK495)
Rasmus Pallisgaard (DWP992)

December 14, 2023

1

(a)

We have followed the strict 2PL protocol to achieve before-or-after atomicity to prevent two transactions' `READ` and `WRITE` to run concurrently. This goes for both the single lock book store and the two level lock book store.

(b)

We tested for this atomicity using the strategy of allowing multithreaded access to the book store, through both a client and a store manager accessing the book store simultaneously, as well as two clients accessing the book store simultaneously. We test to see if dirty writes occur by setting an initial setting, then performing concurrent work that, if dirty reads or writes were to happen, we would get an error throughout the test as the work was performed. For some tests, a final check is performed to see that some concurrent but ultimately negating work results in the same or some predetermined status.

(c)

We did not use different testing strategies for the two book store locking strategies, and instead relied on a single `BookStoreConcurrencyTest` file to contain all tests for both the classes. Two different test files however, wouldn't necessarily break with modularity, as they could ultimately have different expectations with regards to concurrency, much like how we create a separate test file for concurrent tests along side the standard tests of functionality.

2

Yes, because the 2PL we use is the strict 2PL that finishes the thread it started with first before doing the next thread making sure it respect the queue for the threads. This is true for both the `SingleLockConcurrentCertainBookStore` and the `TwoLevelLockingConcurrentCertainBookStore` in our implementation.

3

Strict 2PL does not inherently prevent deadlocks, but we have implemented some additional functionality which helps prevents deadlocks, which intended to impelemtn the strict 2PL protocol. Originally our code would upgrade a `READ` lock to a `WRITE` lock, but this was done by unlocking the `READ` lock first before the same transaction initialized the `WRITE` lock; thus *not* following the strict 2PL protocol as a transaction then would be able to start more locks after unlocking. To our knowledge no method of upgrading a `READ`- to a `WRITE` lock exists, so with the original implementation, if multiple transactions tried to upgrade from a `READ`- to a `WRITE` lock, this could lead to deadlock. To prevent this we initially use a `WRITE` lock instead of a `READ` lock if at *any* point in the transaction there exist a `WRITE` lock, thus no need for updating a lock and this enables the strict 2PL protocol and prevents deadlocks in this case. There is no additional deadlock detection implemented.

4

As mentioned in section 3, we initialise a `WRITE` lock instead of a `READ` lock if at *any* point in the transaction there exist a `WRITE` lock. The use of too many `WRITE` locks may limit the scalability of a system if used too frequently. Especially if the write locks the whole database and not just individual objects. Too many `WRITE` locks also prevents `READ` locks from different transactions to read an object concurrently thus creating more wait time on the locks, limiting the scalability of the service.

5

As mentioned in section 3, we initialise a `WRITE` lock instead of a `READ` lock if at *any* point in the transaction there exist a `WRITE` lock. Because of this we don't deadlock but this harms the performance, as the general introduction of locks contributes to additional computational costs.