ACS programming assignment 2

ndp689 mnp553

December 2021

1. **Provide a short description of your implementation and tests, focusing on:**

   (a) What strategy have you followed in your implementation to achieve before-or-after atomicity for each of SingleLockConcurrentCertainBookStore and TwoLevelLockingConcurrentCertainBookStore?

   For the SingleLockConcurrentCertainBookStore, we use shared locks for Reads and exclusive locks for Writes on the whole database to achieve before-or-after atomicity.

   For the TwoLevelLockingConcurrentCertainBookStore, we use global exclusive locks for Inserts and Deletes on the database to simulate intention locks, and global shared locks for other read-realted operations. At the bottom level, we also implement locks on the books regarding their ISBN to ensure before-or-after atomicity.

   (b) How did you test for correctness of your concurrent implementation? In particular, what strategies did you use in your tests to verify that anomalies do not occur (e.g., dirty reads or dirty writes)?

   In the test case1, we let a client C1 to call buyBooks to buy a given and fixed collection of books and simultaneously let a client C2 to call addCopies to add the same number of copies of the same books. The test result is that the books have the same number of copies in stock as they started. In the test case4, we let a client C1 to call buyBooks to buy a given and fixed collection of books. C1 then calls addCopies to replenish the stock of exactly the same books bought. C2 calls addBooks to add the same number of copies of the same books. The test result is that the books have the number of copies as they started plus the copies C2 added. **So, the test results means we have achieved the**

concurrent implementation.

In the test case2, we let a client C1 to call buyBooks to buy a given and fixed collection of books. C1 then calls addCopies to replenish the stock of exactly the same books bought. C2 continuously calls getBooks to check the quantities for the books. The result of the test is that these books seem to have just been purchased or replenished. And in the test case3, we let a client C1 to call updateEditorPicks to pick book1 and do not pick book2, and then to call updateEditorPicks to pick book2 and do not pick book1. C2 continuously calls getEditorPicks(1) to check the No.1 picked book. The test result is C2 always get the snapshot returned either the book1 was picked or the book2 was picked. **Then, the test cases verify that anomalies do not occur.**

(c) Did you have to consider different testing strategies for SingleLockConcurrentCertainBookStore and TwoLevelLockingConcurrentCertainBookStore? Since these classes need to provide the same semantics, would the use of different strategies be a violation of modularity?

No, we did not. Because both of them technically function the same and they only differ in the degree of concurrency.

Using different strategies will not violate modularity.

2. Is your locking protocol correct? Why? Argue for the correctness of your protocol by equivalence to a variant of 2PL. Remember that even 2PL is vulnerable when guaranteeing the atomicity of predicate reads, so you must also include an argument for why these reads work in your scheme. Note: Include arguments for each of SingleLockConcurrentCertainBookStore and TwoLevelLockingConcurrentCertainBookStore.

For the SingleLockConcurrentCertainBookStore, it is equivalent to the CS2PL since the locks will be added on the whole database. Such locking mechanism ensures that the Writes from different threads operate sequentially and the clients can read simultaneously.

For TwoLevelLockingConcurrentCertainBookStore, it is equivalent to the S2PL since the local read-write locks will be added on each book when the book's information will be updated and all the locks will be released when transaction finishes. And we also have intention locks(IX and IS), which will be added on the database, to ensure sequential Writes on each book and concurrent Reads. So, the locking protocol of the TwoLevelLockingConcurrentCertainBookStore is correct.

3. **Can your locking protocol lead to deadlocks? Explain why or why not. Note: Include arguments for each of SingleLockConcurrentCertainBookStore and TwoLevelLockingConcurrentCertainBookStore.**

   For SingleLockConcurrentCertainBookStore, there will not be any deadlocks since we ensure Writes are sequential, and Reads must add share locks before functioning.

   For TwoLevelLockingConcurrentCertainBookStore, deadlocks can happen when local exclusive locks are used and both of two threads wait for each other.

4. **Is/are there any scalability bottleneck/s with respect to the number of clients in the bookstore after your implementation? If so, where is/are the bottleneck/s and why? If not, why can we infinitely scale the number of clients accessing this service? Also, discuss how scalability differs between the two variants SingleLockConcurrentCertainBookStore and TwoLevelLockingConcurrentCertainBookStore.**

   For SingleLockConcurrentCertainBookStore, there is the bottleneck for Writes, where the clients in line need to wait due to the exclusive lock on the entire database.

   For TwoLevelLockingConcurrentCertainBookStore, the same bottleneck can arise if there is a client with Insert or Delete operations because it will use a global exclusive lock on the database and other clients need to wait.

   TwoLevelLockingConcurrentCertainBookStore provides the better scalability. It allows clients to add different books' copies simultaneously since we use global shared locks on database as IS locks and use exclusive locks on the specific books. And clients can read in the progress of adding books' copies by others, which cannot be implemented in SingleLockConcurrentCertainBookStore.

5. **Discuss the overhead being paid in the locking protocol in the implementation vs. the degree of concurrency you expect your protocol to achieve. Contrast how this trade-off differs between the two variants SingleLockConcurrentCertainBookStore and TwoLevelLockingConcurrentCertainBookStore.**

   For SingleLockConcurrentCertainBookStore, the cost for concurrency is low because the lock manager only needs to care about the lock on the database. And the degree of concurrency is low as well since a Write will make the whole database locked and all other Writes and Read wait. Such locking mechanism is only suitable for Read-dominant clients.

For TwoLevelLockingConcurrentCertainBookStore, the cost is high as local locks exist but the degree of concurrency is high. Generally, we want to achieve greater concurrency to process requests faster even as the cost of higher overhead.