Advanced Computer Systems

Programming Assignment 2

University of Copenhagen

Julian Sonne Westh Wulff <bkx655@alumni.ku.dk>

Toke Emil Heldbo Reines <bkx591@alumni.ku.dk>

December 16, 2021

1 Implementation and Test

1.a Before-or-after Atomicity

`SingleLockConcurrentCertainBookStore`

We implemented this locking protocol by adding a global `ReadWriteLock` to the entire server. In this case, before-or-after atomicity is achieved by getting a shared/exclusive lock on the global lock (db lock) prior to reading/writing from the data, and unlocking it afterwards, as to ensure that no write operations are executed in parallel / interleaved.

`TwoLevelLockingConcurrentCertainBookStore`

We implemented this locking protocol using one top-level lock similar to the one described in above. Though in this case this lock was only used as an exclusive lock when new books were added (`addBooks()`) or books were deleted (`removeBooks()` or `removeAllBooks()`), otherwise the lock was used as a shared lock. This ensured that the books database stayed static/consistent for our locking schemes, which is needed to guarantee correctness of a two-phase locking scheme. Besides utilizing the top level lock we implemented a "lock map" that held individual `ReadWriteLock`'s for each book in the database. Thus the strategy for using these locks to achieve before-or-after atomicity was similar to the strategy described for the `SingleLockConcurrentCertainBookStore` where a shared/exclusive lock is set (in this case on the individual books) prior to reading/writing from the data, and unlocking it afterwards. The protocol is described in further detail in section 3.

1.b Testing the correctness of our implementation

Our tests are located in the ConcurrencyTest.java file.

We wanted to test for the three anomalies, namely dirty reads (WR Conflicts), unrepeatable reads (RW Conflicts), and dirty writes (WW Conflicts). As we have no direct control over when the locks are released (i.e. the transactions are committed), we have written tests that mimicks this as best as we could. Finally we wanted a test to provoke a deadlock or rather, test that no such deadlock will occur, by running all exposed methods a large number of times, concurrently and in random order. The four tests are identical in their setup, in that they run multiple threads who each run a set of function calls that tests for each anomaly and for the before-or-after atomicity.

For the WR conflicts, dirty reads, no endpoint in the TwoLevelLockingConcurrentCertainBookStore aborts after having performed write operations, but our Test 3, unaware of the implementation details, tests for this anomaly.

For the WW conflicts, dirty writes, any two threads concurrently writing (buying, adding copies etc.) to two or more books will at some point run in a interleaved schedule (if not for our locks). Therefore, both Test 1

2 Correctness of the locking protocols

`SingleLockConcurrentCertainBookStore`

This implementation resembles the 2PL variant refereed to as *Solution 1* in Traytel (2021) as it uses a global `ReadWriteLock` on the entire database as described in subsection 1.a. The only difference here being that our implementation uses a `ReadLock`/shared lock for operations that does not modify the data, whilst *Solution 1* only uses an exclusive lock. Thus, our implementation guarantees atomicity as exceution is done in serial, even for read operations as a locked `ReadLock` or `WriteLock` blocks any other transaction to acquire a lock of the opposite type until the lock has been unlocked. As transaction are performed in serial this also means that the set of objects is guaranteed to be fixed doing the execution of a transaction.

`TwoLevelLockingConcurrentCertainBookStore`

As we did not know how to implement deadlock prevention or deadlock detection within a reasonable time-frame we chose to implement the CS2PL scheme, where we get exclusive locks on all the data items that are modified upfront, and get shared locks on alt the data items that are only read upfront, execute the transaction, and realease all the locks. The locking mechanism was implemented using a "lock map" (`BookStoreLockMap`) as described in subsection 1.a, that maps the books in the database to a `ReadWriteLock`. To ensure that a single transaction could acquire all locks needed for the transaction without deadlocking with another transaction we implemented `BookStoreLockMap` to internally use its own `ReadWriteLock` that makes sure that setting the individual locks is done atomically. Thus by following the CS2PL protocol atomicity is guaranteed, even for predicate reads, as we guarantee the database's consistency due to the top-level lock described in subsection 1.a.

3 Deadlocks

`SingleLockConcurrentCertainBookStore`

Since all functions requires a a single global `ReadWriteLock`, all write functions are executed in serial. Therefore we will not experience any deadlocks.

`TwoLevelLockingConcurrentCertainBookStore`

Since we have implemented the CS2PL scheme, we are guaranteed that no deadlocks can occur. To make sure that this actually holds, we also created a test that checks for this by having 4 clients concurrently making 100 semi random API calls each.

4 Scalability/Bottlenecks

As the Book store can be scaled in the application-/proxy-layer by deploying this layer on more nodes, the scalability issues/bottlenecks resides within the server-layer's database, as the database can not be deployed on multiple nodes without loosing its consistency. Due to the locking protocol used in *SingleLockConcurrentCertainBookStore* this version of the book scales the least compared to the *TwoLevelLockingConcurrentCertainBookStore*, as write operations are forced to be performed in serial regardless of the accessed data items for the former. If we suspect that our book store clients will make lots of write operations to the database the *SingleLockConcurrentCertainBookStore* will scale increasingly bad compared to the *TwoLevelLockingConcurrentCertainBookStore* as the number of clients performing write operations increase. Thus *TwoLevelLockingConcurrentCertainBookStore* as has a greater amount of concurrency this will scale implementation will scale best especially with respect to write operations.

When looking at the scalability of the *TwoLevelLockingConcurrentCertainBookStore*, one also has to consider the size of the database and the number of clients and therefore locks - the number of locks scales with the number of books. *TwoLevelLockingConcurrentCertainBookStore* can handle more requests, but will also have to keep more locks in the memory.

5 Locking protocol overhead

The overhead is smaller in *SingleLockConcurrentCertainBookStore* (and constant/independent of the number of books being read/update), but since *TwoLevelLockingConcurrentCertainBookStore* allows for concurrent writes (with non-overlapping sets), *TwoLevelLockingConcurrentCertainBookStore* allows for a much higher degree of concurrency. The overhead being paid in the locking protocol in *TwoLevelLockingConcurrentCertainBookStore*, is dependent on the number of books being read/updated and the constant of acquiring the writelock on the `BookStoreLockMap`. For delete/insert operations, a global write-lock is required, which only adds the overhead of acquiring and releasing this. For only read operations, the overhead of acquiring a read-lock is insignificant compared to the potentially infinite concurrency we can now allow.

References

[1] D. Traytel. Concurrency control: Two-phase locking (2pl), introduction to schedules and serializability. Lecture slides, 2021.