ACS-A3

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

December 3, 2024

Implementation details and description

Single lock concurrent book store

When a transaction is performed in the book store through calling one of the public methods associated with actions on the book store our locking protocol employs the following rule: If a read is performed during the transaction, then activate a shared lock on the database, and if a write is performed, then activate an exclusive lock. In code these locking mechanisms are implemented through the pattern seen in Listing 1. We first request the lock. Upon obtaining the lock, we execute all transaction code including exception throws inside a try-block, followed by a finally block where we release the lock. Thus if any exceptions are thrown, the finally block will release the locks before returning or throwing. This pattern is used both for shared and exclusive locks.

Our implementation of a single lock concurrent book store utilizes a database wide `ReentrantReadWriteLock` to simulate a shared/exclusive lock using read/write lock behavior. This allows us to use employ a shared (read) lock for transactions that reads from the book store and an exclusive (write) lock for transactions that write to the book store. The lock we utilize has the limitation that we cannot upgrade a lock from a shared lock to an exclusive lock, which becomes relevant when deciding a locking policy to employ. We cannot simply release a read lock and acquire a write, as this would break with 2PL, and thus in the case where we encounter first a read and then a write, we need to initially perform a write lock instead. This lock will be held until the end of the transaction, at which it will be released.

Two lock concurrent book store

The two-locking mechanism employs multiple-granularity locks in two levels: Database wide and across individual items. The database wide lock is supposed to *simulate* an intention lock as described in the assignment, i.e. an exclusive lock when we update or delete from the book store, and an intention-esque lock simulated by a read lock when the database is read from or updated. The locks for individual items are also simulated by read/write locks, one for each item. This is implemented by a nested version of Listing 1, wherein if we wish to read or update items in the database we first initiate a database-wide read lock and then inside the following try-block initiate a read/write lock on the specific items we wish to update. One note is that sometimes we might need to request a lock on items from a collection that might be null, in which we have to check if it is null before requesting locks.

This lower level lock on specific items is implemented again through `ReentrantReadWriteLock`s

for each item, all of which are stored in a `ConcurrentHashMap`, which provides a thread-safe implementation for hash maps that efficiently locks only the parts of the map being used, leading to high efficiency. The same issues of not being able to upgrade locks apply to both the upper and lower level locks here as well, and the policies for locking mimic the method described in the section on single level locking.

Listing 1: implementation of a read lock in java.

```
lock.readLock.lock();
try {
    // All transaction computation including error checking is
       handled here.

    ...
} finally {
    lock.readLock.unlock();
}
```

1 Provide a short description of your implementation and tests

(a) What strategy have you followed in your implementation to achieve before-or-after atomicity?

For our single lock implementation we employ a strict strategy of strong isolation between transactions - If a transaction wishes to read (and not write) from the database at some point a database-wide read (shared) lock is employed, and if a transaction wants to write to the database at some point a database-wide write (exclusive) lock is employed. This allows read only transactions to be executed intertwined while still resulting in results that would be as if they were executed sequentially. A write transaction demands exclusive access to the database, so one transaction cannot intertwine this transaction with others, thus isolating the modifying effects of these transactions. By this strict nature there is no possibility of having conflicts between transactions, thus ensuring that the implementation is conflict-serializable, meaning it has before-and-after atomicity.

For the two lock implementation we use a database lock for inserts and deletes and locks on individual items. By the same reasoning as above we have before-and-after atomicity for inserts and updates due to the database wide write lock excluding any other transaction from accessing the book store. For read-only and updating transactions the same arguments as above can be applied on specific items to show before-and-after atomicity: Transactions that only read from the book store can execute in tandem and the result will be indifferent from a sequential execution: When updates are executed, the write locks on individual items, meaning other transactions cannot access these items until the locks are lifted at the end of the current transaction. Much like the single lock version, this ensures that conflicts cannot happen, which means our method will produce a conflict-serializable schedule, ensuring it has before-and-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?

We test for correctness of our concurrent implementation through 4 different tests that are based in the strategy of accessing the book store through store managers and clients

running in different threads that are thus able to simultaneously access the server. We use two approaches for the tests:

1. We have different threads perform two conflicting accesses (one updates and the other updates/reads) and at the end assert that no dirty writes were made.
2. We have one thread that continuously accesses the database through updating, adding, or removing books, while another thread continuously reads from the database and ensures that no dirty reads happen meanwhile.

We have created the following tests:

1. Our first test uses the first approach with adding copies of a book and those copies again (updates to the book store).
2. Our second test uses the second approach with the same functionality.
3. Our third test uses the second approach but with inserts and deletes through adding and removing books.
4. Our fourth tests expands on the first by testing that multiple clients can purchase books simultaneously while the store manager continues to refill books.

(c) Did you have to consider different testing strategies?

We used the same testing strategy for both implementations contained in a single `BookStoreConcurrencyTest.java` file that contain all tests for both the classes. Having two different test strategies would break with modularity, since the classes need to provide the same API, schematics and functionality.

2 Is your locking protocol correct? Why? Argue for the correctness of your protocol by equivalence to a variant of 2PL

For the single lock implementation we employ a locking protocol where all locks needed are requested at the beginning of the transaction and all held locks are released at the end of the transaction. This protocol is equivalent to a Conservative Strict Two-Phase Locking protocol (CS2PL).

For the two-lock implementation all needed locks are still requested at the beginning of the transaction regardless of whether we exclusively lock the database or need the intention lock followed by locks on individual items. This again equates to the use of CS2PL. Regarding predicate reads, by employing the intention read lock across all items once one item is needed combined with keeping this lock for as long as the transaction is active, implying as long as the predicate read is relevant, we ensure that atomicity still holds for predicate reads.

3 Can your locking protocol lead to deadlocks?

For the single lock implementation we have implemented a locking protocol that equates to CS2PL. In terms of deadlocks, CS2PL ensures that deadlocks cannot happen due to transactions needing to request ALL locks before beginning executing, which prohibits the two transactions of requesting locks in a pattern that would cause deadlocks.

For the two-lock implementation we still request all locks needed at the beginning of a transaction. Thus neither locking layer can request locks in a way that could cause deadlocks.

4 Is/are there any scalability bottleneck/s with respect to the number of clients in the bookstore after your implementation?

Due to the conservative nature of our implementation transactions can hold locks for an unnecessary amount on time, and even have periods in which they hold a higher priority lock than they might need at the time. This is particularly costly for write operations that take hold of the entire database during the execution of a transactions. This limits scalability in general, and this limitation is significant in certain situations. Considering only the implementation without RPC components, as both clients and requests for the book store increase the strict and conservative locking approach can cause scalability issues for both reading and writing use - reading transactions stall completely while some writes are being performed, and if these are large they can completely stall the book store for a significant amount of time; and writing transactions are stalled completely as long as some reading transaction occurs.

This is alleviated by the two-lock approach, since now updates can be performed in parallel with reads as long as they do not access the same items. Still, the fact that item wide locks are held throughout the transaction limits scalability again - especially because these locks could potentially be held for a significantly shorter time span than they are used (only when a single item is accessed).

Considering that this service is a book store, it is likely that most operations will be reads (what books are available) and updates (purchase or replenish books), and as such the specific scalability issues pertaining to the book store would be much smaller in the case of the two-lock implementation.

It is also relevant to note that scalability is limited in such a way that our system cannot handle an infinite amount of concurrent requests, and thus there comes a point where the system becomes overloaded with requests. At that point, the server would start to thrash, processing a number of concurrent requests it does not have the resources to handle at capacity. Timeouts would also be a case in this situation, but given timeouts are caused by the request hanging for too long, which can be caused by an overloaded system, we can expect thrashing to be more of a concern than timeouts.

5 Discuss the overhead being paid in the locking protocol in the implementation vs. the degree of concurrency you expect your protocol to achieve.

The single lock implementations equivalence to CS2PL shows an immediate concurrency issues - the fact that we immediately lock at the initialization of a transaction limits the implementations concurrency as a transaction that performs a single write must acquire a write lock for the entirety of its duration, and even for read only transactions the locks are held for longer than necessary, limiting the available space for transactions that write. The locking overhead however only that a single lock needs to be acquired/released per transaction regardless of the operation performed on the book store.

For the two-lock implementation we are still inherently limited by the conservative nature of its 2PL equivalence. However, as we introduce the two levels of locks we increase the concurrency compared to the single lock implementation by allowing update operations to action on individual items using individual item locks instead of having to lock the entire database. Note that these individual locks are still held for

the duration of the transaction, leading to still limited concurrency in the case where certain items are frequently updated and read.

This higher concurrency for the two-lock implementation is contrasted by an increased overhead in terms of locking since read/update operations now not only read lock the database on database level (1 lock), but also read/write lock on each different item accessed during the transaction. For some transactions that accumulate across all items in the book store, this overhead can be significant.