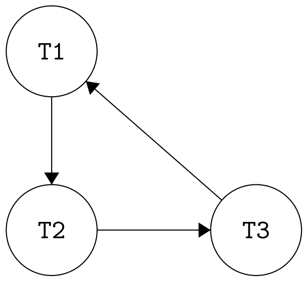
Assignment 2

Carl-Johannes Johnsen, Daniel L. Pedersen

1st December 2015

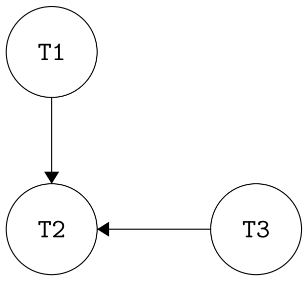
1 Serializability and Locking

Schedule 1



This schedule is not conflict-serializable, due to the precedence cycle in the graph. This schedule could not be created using strict 2PL, since `T1` holds a lock on `X`, when `T2` writes to `X`.

Schedule 2



This schedule is conflict-serializable, since there are no cycles in the precedence graph. This schedule could be created using strict 2PL by injecting locks (`S` is a shared lock, `X` is an exclusive lock (`S` is a shared lock, `X` is an exclusive lock)):

```
T1: S(X) R(X)                       X(Y) W(Y) C
T2:                       S(Z) R(Z)            X(X) W(X) X(Y) W(Y) C
T3:            X(Z) W(Z) C
```

2 Optimistic Concurrency Control

Scenario 1

Roll back since the validation fails for T2, with `Test 1` and `Test 3` failing, because of scheduling, and `Test 2` fails because of: $WS(T2) \cap RS(T3) = \{4\}$

Scenario 2

Roll back since the validation fails for T1, with `Test 1` and `Test 3` failing, because of scheduling, and `Test 2` fails because of: $WS(T1) \cap RS(T3) = \{3\}$

Scenario 3

T3 will be allowed to commit, with it passing `Test 2` for both T1 and T2: $WS(T1) \cap RS(T3) = \emptyset$ and $WS(T2) \cap RS(T3) = \emptyset$

3 Questions on the Implementation

1. Short description of the imlementation and tests

We have implemented conservative strict 2 phase locking. We have made a lock for the book map and a map of locks, one for each book. Every lock is a `ReadWriteLock`. In every method we start by locking the book map lock, the write lock is aquired in the methods that either add or remove books from the map, and read lock otherwise (e.g. accessing book data). A lock is aquired for each book within the sanity check loop. All locks are released before either return, or before an exception is thrown.

We created two tests outside of the two specified ones. The first of these two test that `addBooks()` and `removeAllBooks()` are atomic, by having one thread continously adding books and removing them after. A second thread then runs tests that either all books are present or none. The second test, ensures the atomicity of `updateEditorPicks`, by continously switching between two states of editor picks, one being half of the books being editor picks, and the other having the second half as editor picks. A second thread then checks that the right number of books are editor picks.

2. Is the locking protocol correct?

We argue that it is correct. We are using conservative strict 2 phase locking, i.e. we do not release locks, before the operation is complete. We guarentee predicate read/write, by aquiring a read lock on all the affected books, before returning the books, and by aquiring an exclusive lock on the book map, before adding or removing entries.

3. Can the locking mechanism deadlock?

Yes, since we don't ensure that locks are aquired in a consistent order, for all operations. E.g. a transaction could aquire a write lock on one book, try to aquire another write lock on another book, which is locked by another transaction that is trying to lock the first book. It is possible to sort the book sets to ensure that no dead locks occur.

4. Are there any scalability bottlenecks?

Yes, since the write locks are exclusive, only one client can access that resource at a time, when one is aquired.

5. Discuss the overhead vs degree of concurrency

There is an introduced overhead, due to locks being aquired in the sanity check loop or in a seperate loop, and due to a loop for releasing all the locks. However, concurrency is greatly increased for reads, and also some for writes, while maintaining atomicity for writes.