6.814 - Lecture 13 Notes

Long Nguyen

October 30, 2017

Last time, we covered serial-equivalent schedules coming out of a sequence of concurrent operations.

1 Lock-Based Concurrency

Two-Phase Locking

Before every read, acquire an S lock. Before every write, acquire an X lock. Only release locks after all locks needed have been acquired.

Cascading Aborts

A cascading abort is an abort that propagates from one transaction to another due to concurrency issues. Suppose T_1 is before T_2 and T_1 reads and writes X before T_2, but aborts after T_2. Then, T_2 saw something it shouldn't have and must also abort, because it relied on values from T_1's pre-abort actions.

How do we solve this? Well, we need **strict two-phase locking,** where we hold *all write locks (X locks) until the end of the transaction.* There's also **rigorous two-phase locking,** which also holds S locks as well until the end of the transaction as well. Being rigorous ensures recoverability without cascading aborts and ensures that the commit order follows some serial order.

Deadlocks

Two currently locked objects wait on each other forever. How do we solve this? Traditionally, we can enforce some ordering on the locks (think semaphores).

More formally, **deadlock avoidance** is the principle above. However, it's impractical and doesn't work (in general) for database systems because we don't know what we're going to do, and thus, what we need to lock, in advance.

Instead, we focus on **deadlock detection**. Abort something when we detect a deadlock so that other transactions can continue. Two possible ways to detect deadlocks:

- If the precedence graph (remember, directed edges) has a cycle, we know there's some dependence on each other, and therefore, there must be a deadlock.
- Use a timer to enforce some timeout. It's easy to implement, but it's difficult to tune it properly to general situations.

2 Optimistic Concurrency Control

This is an alternative to locking. In contrast, locking is pessimistic in the sense that because we acquire all locks and don't release until the end, we're assuming a worst-case scenario. What if you were given the best case and you still used locks? There's an overhead!

Our goal is to "get lucky" and achieve some serializable schedule, which we can check after our execution.

Read Phase

Local writes are done here. Keep track of values read and written. Build read and write sets, which will be used during validation time.

Validate Phase

We check for serializability here. Use read and write sets to check for valid scheduling. Order the transactions $T_1, \ldots, T_n$ and compare T_i to $T_1, \ldots, T_{i-1}$—comparing some transaction to those that came before it.

What kind of ordering should we do? *Assign the ordering at the end of the read phase.*

If we read some value of x that wasn't the most recent value written to x in previous transactions, then something wrong happened. Likewise, if earlier transactions saw things that we wrote, there's a contradiction.

Validation Rules

R and W are the read and write sets from before. For some T_j, all T_i where $i < j$ must obey at least one rule below.

1. T_i completes write phase before T_j starts read phase. Essentially, T_j started after T_i.
2. $W(T_i)$ does not intersect $R(T_j)$—if they do intersect, then T_j wrote something T_i wrote, or T_i read something T_j wrote. Also, T_i completes write phase before T_j starts write phase.
3. $W(T_i)$ doesn't intersect $R(T_j)$ or $W(T_j)$; T_i completes read phase before T_j completes read phase.
4.

Write Phase

Local writes are finalized here.

Read-Only Transactions

There are clearly no conflicts if only read between transactions. We don't need to keep track of write sets or anything complicated. There shouldn't be any conflicts, and the rules will clearly follow.