Lecture 18 -- Isolation + Concurrent Operations April 13 2009

Sam Madden

Key Ideas:

Serial equivalence
Two-phase locking
Deadlock

So far, we've imagined only one transaction at a time -- in effect, a big lock around the whole system (database, file system, etc.).

What's wrong with that? Why does one transaction at a time utilize the system less than running everything serially? (Might expect the opposite to be true, since switching between transactions presumably has some overhead!)

- Hard to exploit multiple processors
- Might want to perform processing on behalf of one transaction while another waits for disk or CPU

Our goal in isolation is <u>serial equivalence</u> -- want final outcome of transactions to be same as running transactions one after another.

A = 50
B = 10

xferPercent(A,B,.2)   xferPercent(A,B,.2)

A = 40
B = 20

A = 32
B = 28

```
xfer(A,B,10)
RA  //  t = .1A
WA // A =  A - t
RB
WB // B = B + t
```

Example "schedule":

```
RA
WA  // a= 40
          RA
          WA // a= 32
RB
WB // B = 20
          RB
          WB // B = 28
```

Serializable? Yes. Why?
Outcome is the same.

```
RA
          RA
WA // a= 40
          WA  //  a = 40
RB
WB //B = 20
          RB
          WB // B = 30
```

Not serializable!

Is there some test for serializability?

What went wrong in this second example?

T1's read of A preceded T2's write of A, and T2's read of A preceded T1's write of A.

To formalize, define "conflict" -- two R/W operations o1 in T1 and o2 in T2 conflict if either o1 or o2 is a W and both are to the same object.

(aren't worried about two read operations.)

A schedule is serializable if all *conflicting* operations in a pair of transactions T1 and T2 are ordered the same way -- e.g., T1-T2 or T2-T1

Why? Suppose they weren't. Then one transaction might read something before another transaction updated it, and update it afterwards, as in example above. First transactions effects are lost!

Easy way to test for serializability: conflict graph

Make a graph with nodes as transactions
Draw arrows from T1 to T2 if op1 in T1 conflicts with op2 in T2 and op1 precedes op2

Example:

Now that we understand what it means for a schedule to be serializable, our goal is to come up with a locking protocol that ensures it.

We want to ensure that all conflict operations are ordered in the same way. Use locks to do that. Don't require the programmer to manually get locks.

Instead, transaction system acquires locks as it reads objects.

Locking protocol:
- before reading/writing an object, get lock on it
  - (if lock isn't available, block)

<u>Can I release right away? (No! example):</u>

```
T1          T2
Lock A
            Lock A
    RA      Block
    WA
Release A
            Lock A    //T2 "sneaks in" and makes its updates, violates serializability
            Lock B
                RA
                WA
                RB
                WB
            Release A
            Release B

Lock B
    RB
```

```
        WB
Release B
```

Exposed value of B before updating that T2 shouldn't have been able to see -- not serializable.

```
        Locking protocol:
        before reading/writing an object, get lock on it
                (if lock isn't available, block)

        release locks after transaction commit
```

```
Lock A                  Lock A (block)
        RA              I
        WA              I
Lock B                  I
        RB              I
        WB              I
commit                  I
release A, B            I
                        v
                        RA
                        WA
                        Lock B
                        RB
                        WB
```

(force T2 to happen completely after T1 -- clearly given up some concurrency; note that if T2 access other objects, e.g., C, first, that wouldn't be a problem.

will address this limitation in a minute.)

Issue: deadlocks

What if T2 tried to get lock on B first?

No transaction could make progress.

Can we just require transactions to always acquire locks in same order? No, because transactions may not know what locks they are going to acquire -- e.g., things that are updated may depend on the accounts that are read (suppose we deduct from all accts with value > x).

So what do we do about deadlocks? Simple: abort one of the transactions and force it to rerun. This is OK because apps must always be prepared for possibility that transaction system crashes and their action aborts.

How do we tell that transactions are deadlocked?

Two basic ways:

1) If a transaction waits more than t seconds for a lock, assume it is deadlocked and restart it.

(Simple, may think a long running transaction a deadlock.)
MySQL does this.

2) Build a "waits for graph" -- if T1 is waiting for lock held by T2, draw an edge from T1 to T2. If there is a cycle, then there is a deadlock.

Example:

Making locking protocol more efficient:

Optimization 1:

Once a transaction has acquired all of its locks, it can proceed with out any other transaction interfering with it.

Lock point:
It will be serialized before any other transaction that it conflicts with.

If a transaction is done reading/writing an object, it can release locks on that object *without affecting serialization order*. Doesn't have to wait until the end of the transaction since it isn't going to use the value again.

Example:

```
lock A
     RA                 lock A (block)
     WA
lock B  <--- lock point
release A
                             RA
                             WA
                        lock B (block)
     RB
     WB
release B
```

RB
WB

THis is called *two phase locking:*

Phase 1 -- acquire locks, up to lock point
Phase 2 -- release locks, after done with object and after lock point

Optimization 2:

Shared vs exclusive locks

Notice that two read operations of the same object don't conflict -- e.g., if I have two read only transactions, I can interleave their operations however I want and still get the same answer. Protocol didn't allow this, however.

Idea -- have two types of locks: shared (S) locks and exclusive (X) locks. Can have multiple transactions with an S lock, but only one with an X lock.

Transaction can upgrade from S to X if it is the only one with an S lock.

Two phase locking protocol w/ shared locks

Before reading an object, acquire an S lock on it
Before writing an object, acquire an X lock on it

Release locks after lock point

In practice, variants of two phase locking are used:

1) Never release write locks until after transaction commit? Why?

T2 reads A written by T1 before T1 commits -- now T1 aborts, T2 must also abort.

Strict two phase locking

2) Never release any locks until transaction commit? Why?

Can't tell when we are done with locks

Rigorous two phase locking