6.830 Lecture 13 -- **Optimistic Concurrency Control** 4/7/2021

Recap 2PC

(Show rules)

Why is this necessary:

Suppose we didn't follow it -- e.g.,we just released locks when we were done.

(Show slides)

If T1 releases its lock on X after it is done with it, T2 can update both X and Y before T1 finishes. This is not serializable because T2 saw T1's write of X, and T1 saw T2's write of Y.

Note that we don't have to acquire or release all of our locks at the same time.

We call the end of the growing phase (after the last lock is acquired) the "lock point". Serial order between conflicting transactions is determined by lock points.

This is because once a transaction T has acquired all of its locks, it can run to completion, knowing that:

- Any transactions which haven't acquired their locks won't take any conflicting actions until after T releases locks, and
- Any transactions which already have their locks must have completed their actions (released their locks) before T is allowed to proceed.

Ok, so what is a problem with this schedule:

T1	T2
RX	
WX	
	RX
	WX
abort	
	?

Serializable, but T2 would have to rollback too because it read T1's dirty data. This is a "cascading abort" and is something you might like to avoid. To make 2PL **cascadeless**, we need to hold X locks til the end of the transaction. This is called strict two phase locking.

In practice, because we can't know when all locks have been acquired, database systems actually use rigorous two phase locking, where locks are held til end of transaction (show slide).

Note that rigorous 2PL doesn't permit many interesting schedules!

Another problem: deadlocks:

Consider the two transactions, (T1: RX WX RY WY, T2: RY WY RX WY)

And this schedule:

T1	T2
RX	
WX	
	RY
	WY
Wait T2	Wait T1

What's the problem? Waiting for each other to release locks! This is a deadlock!

What can we do about this?

- Require locks to be acquired in order (deadlock avoidance)
    - What's wrong with that? In general needed locks aren't know upfront.
- Detect deadlocks, by looking for cycles in this wait's for graph
- Then what? "shoot" one transaction -- e.g., force it to abort

What can we do instead of locking?

Optimistic concurrent control (today's topic.)

Optimistic concurrency control

What's the idea?

Locking is "pessimistic" in the sense that it acquires locks on things that do not actually conflict! Example:

T1	T2	
RA	RA	might "get lucky", actually end up doing RA1, WA1, RA2, WA2...
WA	WA	

Optimistic concurrency control tries to avoid this by only checking to see if two transactions conflicted at the very end of their execution. If they did, then we kill one of them.

In particular, they use the example of locking the root of a tree while waiting for an I/O to complete. Do you buy this?

Not really plausible -- turns out there are protocols that allow you to avoid acquiring locks while accessing a B-tree (Lehman and Yao -- in the Red Book.)

What's the tradeoff here?

- never have to wait for locks
- no deadlocks

But...

- transactions that conflict often have to be restarted
- transactions can "starve" -- e.g., be repeatedly restarted, never making progress

So when will OCC be a win?

Only when

P(failure) x restart_cost < AVG(Locking delay per query)

Assuming restart_cost is relatively fixed, the choice of when to use OCC vs PCC essentially boils down to the P(failure) -- or the probability that two transactions conflict.

Ok, so how does OCC work?

Transactions have three phases: Read, Validate, Write

What happens during the read phase?

- transactions execute, with updates affecting local copies of the data
- we build a list of data items that were read or written

Keep track of read and write sets for each transaction.
Show how read and write sets are built up.

```
twrite(n,i,v):
    if n not in write_set // never written, make copy
        m = copy(n);
        copies[n] = m;
        write_set = write_set union {n};
    write(copies[n], i, v)))
```

```
tread(n,i):
        read_set = read_set union {n};
        if n in write_set
                return read(copies[n],i);
        else
                return read(n,i);
```

Why do we make local copies of data that is written?
Don't want dirty results to be visible to other xactions
Want to be able to "undo"

What happens during the validate phase?
Make sure that we didn't conflict with any other transactions that have already committed but with whom our execution overlapped.

What happens during the write phase?
We "commit" -- make our writes visible to other transactions.

Ok -- so how does OCC validation work?
Assign an ordering to transactions -- e.g., T1, T2, T3, ...

When Tn commits, check if it conflicts wth T1 … Tn-1

What ordering should we pick?
Start time? (No, because a transaction Ti that finished its read phase has to wait until all Transaction Tj with j<1 finish their read phase -- so this is effectively a form of locking or blocking.)

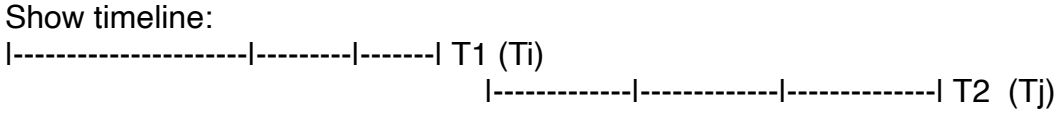
Can we use write time? No -- we have to have an ordering to figure out which transactions to compare against at validation time.

Ok, so what do we do?
Instead, assign ordering based on time at which the read phase finishes. Use sequence numbers instead of times to guarantee distinct values.

During validate phase, what do we want to guarantee for a committing transaction Tj?
That 1) Tj read all of the (conflicting) writes of every earlier transaction, and 2) Tj completed all of it writes after the writes of every earlier transaction.

When Tj complete its read phase, require that for all Ti < Tj, one of the following conditions is true:
1) Ti completes its write phase before Tj starts its read phase (don't overlap at all)

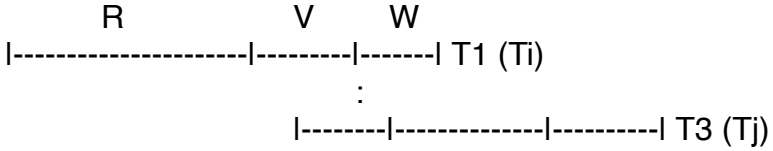
Show timeline:



2) W(Ti) does not intersect R(Tj), and Ti completes its write phase before Tj starts its write phase.

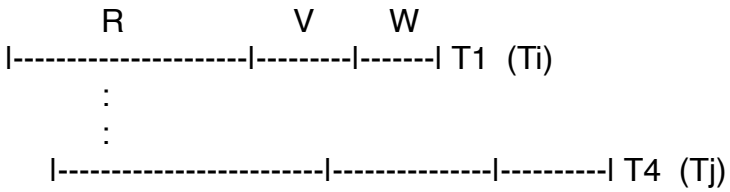
(Ti didn't write anything Tj read, and if Tj wrote anything Ti read/wrote, Ti will be complete and on disk before Tj's writes are.)

Only situation we have to worry about is Tj is reading something while Ti writes, so it sees some but not all of Tis updates.



3) W(Ti) does not intersect R(Tj) or W(Tj), and Ti completes its read phase before Tj completes its read phase.

(Ti didn't write anything Tj read or wrote -- so the only concern is that Tj wrote something that Ti read. But there can't be a conflict here if Ti completes its read phase before Tj starts writing.)



Note that Ti will definitely complete its read phase before Tj by our assignment of transaction ids.

(Otherwise, abort Tj)

Is it possible for serializable schedules to be restarted by these rules? Yes, if locking granularity is off, or in the presence of blind writes, e.g.:

T1	T2	T3
RA		
	WA	
WA		
		WA

(Write sets overlap but no conflict)

(Order is T1 T2 T3) -- OCC provides conflict serializability, not view serializability

(Note that 2PL also won't permit this schedule)

Serial validation -- show notes, describe.

Note that this really only deals with the first two conditions. Third condition can never occur because if Ti completes its read phase before Tj, it will also complete it's write phase before Tj.

What's bad about this?

- Critical section is large,
- Only one transaction can write at a time.

How do we address the first problem?

Check all the transactions that we can before we enter the critical section. (note that we can repeat) -- show code.

How do we address the second problem? Introduce "parallel validation"

Now we check transactions that entered the validate/write phase concurrently.
Is this always a win? Probably generates more conflicts, since third condition now applies. Two transactions that both wrote the same value and entered parallel validation can both abort, but if we'd used sequential validation, only one of them would have aborted.

ok, so how do we deal with read only transactions?

no need for critical section -- can only conflict with transactions that wrote before we began validation. no need to assign transaction ids, since no later transaction cares what we read (review rules.)

have to compare our read set to write set and abort if there is an intersection, but doesn't need to happen in critical section.

in common case of a read-mostly workload, we will often have starttn = finishtn, and no validation is needed at all!

how do we deal with starvation?

They propose that we allow starving transactions to retain the lock on the critical section. This means they will DEFINITELY finish first the next time around. This is a hack.

OCC has had a resurgence of late for its applicability to main memory databases.

Key reason is that in OCC, there is no shared per-object state. Read and write sets are purely local to the transaction. Locking requires use of a shared lock table (which is a point of contention, since every transaction has to read this lock table), or association of locks with individual objects. Both of these induce contention. For read-only queries especially there is never a need for another transaction to look at their read-set. Huge win.

Multicore means that:

- locking delays are (relatively) high
- restart cost is low

Only when

P(failure) x restart_cost < AVG(Locking delay per query)

So OCC becomes more attractive