6.830 Lecture 14 -- **Optimistic Concurrency Control** 10/27/2014

Recap 2PC

(Show rules)

Why is this necessary:

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

Consider the example:

```
T1                  T2
--------------------------------
                  (T2 acquire Y)
                  WY
                  (T2 release Y)
(T1 acquire Y)
RY
(T1 acquire X)
RX
WX
(T1 release X)
                  (T2 acquire X)
                  RX
                  WX
```

Here T2 releases its lock on Y after it is done updating Y but before it updates X.

This is not serially equivalent because T1 sees T2's write to Y, and T2 sees T1's write to X, which would not have been possible in any serial schedule.

You can see that by holding its lock on Y, T2 prevents T1 from doing any of its updates before T2 has finished its updates.

If T2 is allowed to apply its writes to X before T2 does its writes, then we would violate conflict serializability.

But there's a problem here -- suppose T1 and done its write of X first, e.g.:

```
T1                 T2
-------------------------------
                  (T2 acquire Y)
                  WY

(T1 acquire X)
RX
WX
(T2 attempt to acquire Y)
RY
                  (T2 attempt to acquire X)
                  RX
                  WX
```

What happens? Deadlock -- waiting for each other

<u>What can we do about this?</u>

- Require locks to be acquired in order (deadlock avoidance)
    - What's wrong with that? Reduces concurrency.
- Detect deadlocks, by looking for cycles in this wait's for graph
- Then what? "shoot" one transaction -- e.g., force it to abort

<u>What is the "phantom problem"?</u>

If we acquire locks on individual data items, then when one T1 scans a range and later T2 inserts something in the middle of it, and T2 commits before T1, this is not a serial equivalent schedule. Arises because locked objects aren't the same as logical objects queried in the system.

So what can we do? Predicate locks -- e.g., lock a logical range of values.
<u>Is this a good idea?</u> No

<u>Why not?</u>
Execution cost -- testing if two arbitrary predicates is expensive -- don't really want to do this as a part of the inner loop of the lock manager!

Pessimistic -- can end up not allowing things to lock (e..g, when there are consistency req. that prevent 2 ranges from conflicting; e.g., T1 locks "mothers" and T2 locks "fathers")

Hard to determine what range to lock -- someone has to decide which ranges need locking.

In practice, arbitrary predicate locks derived from incoming SQL statements aren't used. We'll return to this later when we talk about lock granularity.

One practical solution is to use locks on keys in a B+Tree, and to lock the next key after every range scan. This "next key locking" prevents new data from being inserted in a range when a B+Tree is available.

<u>What can we do instead of locking?</u>

**Optimistic concurrent control** (today's topic.)

**<u>Optimistic concurrency control</u>**
<u>What's the idea?</u>
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. <u>Do you buy this?</u>

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

<u>So when will OCC be a win?</u>

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.

<u>Ok, so how does OCC work?</u>

Transactions have three phases:  Read, Validate, Write

<u>What happens during the read phase?</u>

- transactions execute, with updates affecting local copies of the data
- we built 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);
```

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

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

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

<u>Ok -- so how does OCC validation work?</u>
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.

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:
```
I----------------------I---------I-------I T1 (Ti)
                                             I-------------I-------------I--------------I T2  (Tj)
```

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 his that Tj is reading something while Ti writes, so it may seem some but not all of its updates.

```
         R                 V        W
I----------------------I---------I-------I T1 (Ti)
                                 :
                           I--------I--------------I----------I T3 (Tj)
```

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.)

```
         R                 V        W
I----------------------I---------I-------I T1  (Ti)
         :
         :
    I-------------------------I---------------I----------I T4  (Tj)
```

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

(Otherwise, abort Tj)

Is it possible for transactions that didn't conflict 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 |

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

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

Serial validation -- show slide, 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.  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!

<u>how do we deal with starvation?</u>

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.

<u>OCC has had a resurgence of late for its applicability to main memory databases.</u>

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