6.830 Lecture 12
3/18/2013

Office hours tomorrow 4--5

Recall last time:

Two phase locking is a special locking discipline. What does it say?
Two phases:

Growing -- acquire locks
Shrinking -- release locks.

Can't acquire any more locks once we've released our first lock. Note that we don't have to acquire or release all of our locks at the same time.

If we release some locks before end of growing phase, someone else could "sneak in" and violate serializability, e.g.:

```
        RB  //lock B
        WB
//release B
RA //lock A
WA
//release A
        RA   //lock A
         WA
RB
WB
```

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.

(In other words, all conflicting actions from another transaction will happen before or after us. -- the definition of conflict serializable!)

Is 2PL cascadeless?

No. Why?

Because a transaction may release its locks before it commits, and another transaction may read its uncommitted state.

How do we guarantee that it is cascadeless? Use "Strict two phase locking".

What's that?

Don't release write locks until transactions commits. Sometimes called "long duration locks". (In example, anything that T2 reads that T1 wrote must happen after T1 commits)

Can also talk about rigorous two phase locking, in which case we hold long read locks as well. Then, we can order transactions by their commit points.

```
T1              T2
RB
                RB
RA
RD
WA                      ;T1 has acquired all locks, can release RB
                RC
                WB
                COMMIT
RD
COMMIT  ; equivalent serial order is T1 T2, even though commit is T2 T1
```

Most database systems use strict two phase locking, which is conflict serializable and cascadeless.

Simple protocol:

Before every read, acquire a shared lock
Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
Release all locks after the transaction commits

What other problem can we have with locking? Deadlocks!

```
T1:             T2:
RA              RB
WB              WA
```

T1	T2	Lock Mgr	
S-Lock(A)		Grant(A)	
RA			
	RB		
	S-Lock(B)	Grant(B)	
X-Lock(B)		Wait(T2)	
	X-Lock(A)	Wait(T1)	--> Deadlock!

What can we do about this?

- 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

What is the "phantom problem"?

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.
Is this a good idea? No

Why not?
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.

Are there other protocols besides locking that we can use to get a serial schedule?
Yes! Optimistic concurrency control.

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	
RA1	RA2	might "get lucky", actually end up doing RA1, WA1, RA2, WA2...
WA1	WA2	

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 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);
```

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?

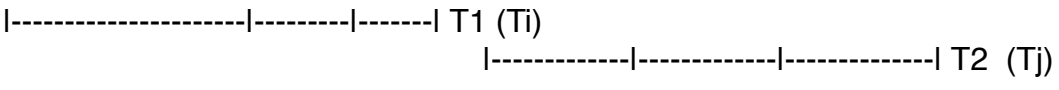
Start time? (No, because a transaction Ti that finished its read phase has to wait until all Transaction Tj with j<1 finish their read phase -- so this is effectively a form of locking or blocking.)

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.

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)

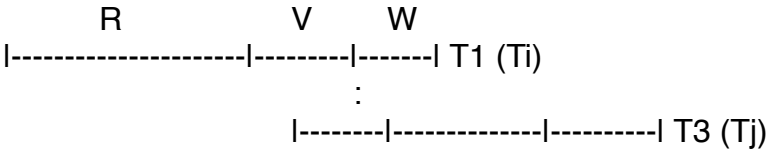
Show timeline:



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

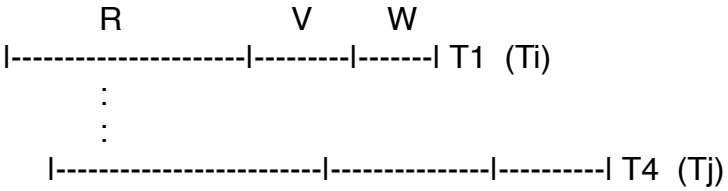
(Ti didn't write anything Tj read, and if Tj wrote anything Ti read/wrote, Ti will be complete and on disk before Tj's writes are.)

Only situation we have to worry about his that Tj is reading something while Ti writes, so it may seem some but not all of its 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 transactions that didn't conflict to be restarted by these rules? Yes, if locking granularity is off, or:

T1	T2	T3
RA		
	WA	
WA		
		WA

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 the 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.) hence, for a read-mostly workload,

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.