6.830 Lecture 13 -- Transactions October 22, 2014

Quiz 1 Back? Standard deviation / mean?

Transaction Processing:

Today: Transactions -- focus on concurrency control today

Transactions
One of the 'big ideas in computer science'

Third big idea we've seen in this class -- the others being relational algebra/data models and query processing/optimization

Four properties
- A tomicity - many actions look like one - 'all or nothing'
- C onsistency - database at time t has certain invariants --
  want to guarantee those invariants are still true at time t+1
- I solation - concurrently running actions can't see each others partial results
- D urability - completed actions remain in effect even after a crash --

"recoverable" meaning system returns to consistent state after crash

Essential properties of transactions -- atomic and recoverable actions.

Example:
t = A
t = t + 1
A = t
Looks correct!
But maybe not if other updates to A are interleaved!
Suppose our increment executes right before another increment's write
our increment will be lost

Why isolation helps:
lets me write and reason about serial code, as if no concurrency
it is possible but very difficult to write correct code w/o isolation

Example use:
```
BEGIN TRANSACTION
SELECT XXX
...
UPDATE XXX   ---------> other concurrent xactions can't see these until commit
UPDATE YYY              (but xaction sees its own updates)
...                <----------- crashes here (effects aren't visible)
COMMIT (or ABORT)
```

History:

According to Gray, earliest transaction processing systems on computers were IBM CICS (customer information and control systems) -- developed for remote access to collections of e.g., customer information, released in 68.

SABRE (for airline reservations) for American in 1962, built by IBM, now called TPF, still used for very high end transaction processing systems.

IMS had transactions (network database) had transactions too -- it was built in the early 70's.

Goal:
atomicity: Make a sequence of actions look like one action that happens as of one instant in time, or doesn't happen at all.

Of course, a sequence of actions takes time, and so to provide this illusion, we need some way to guarantee that multiple actions don't interfere with each other (are isolated).

<u>What's the simplest way to isolate actions?:</u>
One trivial way to do this is to force only one action to run at a time -- e.g., to serialize execution.

<u>What's wrong with that?</u>
Means only one thing is running at once -- may underutilize resources -- e.g., extra processors, disks, or even spare CPU capacity (e.g., when waiting for a disk IO to return.)

Instead, we will allow multiple actions to run concurrently. To deal with this, we need <u>concurrency control</u> -- e.g., someway to prevent these actions from seeing each other's updates.

We will talk about several mechanisms for enforcing concurrency control in a minute.

<u>Why do we talk about concurrency control together with recovery?</u>

Because the type of concurrency control we use will have a dramatic effect on how recovery works. <u>Suppose, for example, that we knew that all actions executed serially? How would that affect recovery?</u>

It'd be a lot easier, because there'd be only one outstanding action at a time.
First, we need some definitions.

<u>How do we know whether a particular interleaving of executions is "correct"?</u> Need a definition. Mostly we focus on one definition in 6.830: "<u>serializability</u>": this means we interleave the execution of transactions, but to have the end result be as though those concurrent actions had run in some serial order.

Lots of other consistency models are possible -- will touch on briefly next time

Example Schedule:

```
        T1          T2                       A and B are objects on disk
        RA          RA   ; t1 = A            could be tuples, pages, etc.
        WA          WA   ; A = t1 * 2
        RB          RB   ;  t2 = B
        WB          WB   ;  B = t1 + t2
```

(Usually, we only show read and write operations for database objects. We assume that there can be arbitrary other instructions between those reads and writes.)

<u>What is the significance of this?</u>

Means that all writes after a read can depend on the value of that read.

<u>What is mapping from SQL to reads and writes?</u>

Consider the query: UPDATE bal SET bal = bal + 1000 where bal > 1M

Needs to read all balances, update those with balance > 1M (what is actually read depends on access method used!)

Serializabe interleaving

```
    RA=1
    WA=2
        RA=2        equivalent to T1 running
        WA=4        before T2
    RB=10
    WB=11
        RB=11
        WB=13
```

Not serializabe interleaving

RA=1	
	RA=1
	WA=2 ---> this update is lost
WA=2	
RB=10	
WB=11	
	RB=11
	WB=12 ---> but this one is visible!

How do we test whether a schedule is serializable?

Two widely known methods for determining that two schedules are serially equivalent: view serializability and conflict serializability.

View serializability says that a particular ordering of instructions in a schedule S is view equivalent to a serial ordering S' iff:

- Every value read in S is the same value that was read by the same read in S'.
- The final write of every object is done by the same transaction T in S and S'

Less formally, all transactions in S "view" the same stuff they viewed in S', and the final state after the transactions run is the same.

Each transaction "views" the same values. Show that this works on transactions above.

S		*S'*	
T1	T2	T1	T2
R A=A1		R A= A1	
W A->A2		W A->A2	
	R A = A2	R B = B1	
	W A_F	W B->B2	
R B=B1			R A = A2
W B->B2			WA_F
	R B=B2		R B = B2
	W B_F		WB_F

===> view equivalent

Second kind of serializability is conflict serializability.

How does this work?

Define conflicting actions. A conflict means for two operations, if you perform them in a different order you get a different result.

What are those:

T1: READ(A)	T2: READ(A)	- no
T1: WRITE(A)	T2: READ(A)	- yes
T1: READ(A)	T2: WRITE(A)	- yes
T1: WRITE(A)	T2: WRITE(A)	- yes (later reads affected)

Two operations conflict if they operate on the same data item and at least one is a write.

We say a schedule is "conflict serializable" if it is possible to swap non-conflicting operations to derive a serial schedule. This means that for all pairs of conflicting operations {O1 in T1, O2 in T2} either

- O1 always precedes O2, or
- O2 always precedes O1.

Show example.

```
    1 RA   5 RA          Conflicts: 1-6, 2-5, 2-6, 3-8, 4-7, 4-8
    2 WA   6 WA
    3 RB   7 RB
    4 WB   8 WB
Serial sched
    1 RA
    2 WA
           5 RA          Can swap 3 and 6, 3 and 5, 4 and 6, 4 and 5,
           6 WA          to get a serial schedule
    3 RB
    4 WB
           7 RB
           8 WA

No serial sched:
    1 RA
           5 RA          1 before 6      Conflicting operations ordered
           6 WA          5,6 before 2         differently, not serializable
    2 WA                                 No way to swap
    ...
```

How can we determine if a schedule is "conflict serializable" ? Attempt swapping, or:

Build a "precedence graph" , with an edge from Ti->Tj if:

Ti reads/writes some A before Tj writes A, or
Ti writes some A before Tj reads A

Example:

```
T1      T2      T3                RA b WA       RB b WB
RA                           T1 ---------- > T2 ---------> T3
WA                                         <-----------
        RB                                    RB b WB
        RA                                 cycle ---> conflict!
        WA
                RB
                WB
        WB
```

If this graph is cycle free, we have a conflict serializable schedule

Testing for view serializability is NP complete (in that you have to consider all possible serial orderings). Most widely used concurrency control schemes provide conflict serializability -- not because of NP completeness but because we have an way to enforce it as transactions run. Note that this is a restriction -- some view serializable schedules are not conflict serializable. Example:

```
T1          T2          T3
RA
            WA
WA
                        WA
```

Schedule violates conflict serializability, but is equivalent to T2 T1 T3 and is view serializable. Only happens when there are "blind writes" as in T2 / T3

Ok -- now that we know how to understand if a particular schedule is serializable, we'll talk about how we actually implement concurrency control. First, let's look more at how concurrency control relates to recovery. What's wrong with the following schedule:

```
            RA
            WA
                        RA
                        commit
            RB
            abort
```

This schedule is not "recoverable"; if T1 were to abort, we might undo its effects, and then T2 would have committed based on uncommitted data. Need to force commits to happen such that T1 commits before T2 if T2 depends on some value written by T1.

Ok, how about this one:

RA		
WB		
WA		
	RA	
	WA	
		RA
abort		

This is "cascading rollback". A "cascadeless" schedule is one in which, if T2 reads something written by T1,T1 has committed before T2 performs that read. Cascadeless schedules are recoverable.

We want our concurrency control scheme to provide serial equivalence and be cascadeless (implying recoverable).

Ok -- so what is the concurrency control scheme presented in the readings? <u>Locking</u>.

Basic idea: a transaction acquires a shared (read) lock before reading an item, and acquires an exclusive (write) lock before writing an item.

What if someone else has a lock?
Only allowed to acquire if lock compatibility table says so:

		T1	
		S	X
T2	S	y	n
	X	n	n

This codifies conflict serializability -- two reads don't conflict, but writes and reads do. If we can't acquire a lock, we wait until the other transaction that has it releases it.

<u>Two phase locking</u> is a special locking discipline. <u>What does it say?</u>
Two phases:

Growing -- acquire locks
Shrinking -- release locks.

Can't acquire any more locks once we've released our first lock. Why?
(Some other transaction could sneak in, update what we've just unlocked. Then it can update something we have yet to lock / update, such that serializability is violated. If we don't release our lock, the other transaction will have to wait for us to acquire all of our locks.)

Example:

T1	T2	T1 locks	T2 locks
RA		SA	
WA		XA (upgrade)	
WB		XB	
		----- lock point	
	RA	Rel A	SA
	WA		XA
RB			
WB			
	RB	Rel B	SB
	WB		XB
			---- lock point (release afterwards)

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.

(In other words, all conflicting actions will happen before or after us.)

Is 2PL cascadeless?

No. Why? Example:

(T1 aborts after Rel A in above example)

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! We'll read a performance study of one, called "optimistic concurrency control" next time.

We saw:
- Transactions , which provide us with a way to provide atomic, recoverable actions.
- How to define a serial equivalent schedule
    - View serialiability
    - Conflict serializability
- Two-phase locking protocol that provides conflict serializability
    - Issues:
        - cascadelessness
        - deadlocks
        - "pahntom" problem