**6.033 Lecture 17 — Isolation** 4/9/2014

Last time:

Saw how to implement atomicity in the presence of crashes, but only one transaction running at a time

[show slides]

This time, going to see how we preserve atomicity when we run multiple transactions. I.e., how to *isolate* running transactions from each other.

Remember our goal: want state of the database after concurrent transactions run to be equivalent to running the transactions in some serial order, i.e., to be *serializable*

<u>So why not just run transactions serially, i.e., one at a time?</u>

Performance! Can't use multiple CPUs to execute multiple transactions at the same time that don't touch the same data, and can't make progress on some other transactions while one transactions is waiting for I/O or user input.

<u>Can't we just manually acquire locks on all records before we read / write them?</u>

Maybe, but that's pretty tricky — have to know exactly what objects a transaction will access, and insert acquire and release statements at the right point to ensure isolation. Would be better if system did this automatically.

Especially complex if using SQL, e.g., "UPDATE emp WHERE sal > 100k"

Some definitions:

*read-set* = objects read by transactions
*write-set* = objects written by transactions

For two transactions T1 and T2, if their neither their read sets nor their write sets intersection, can clearly be run in parallel.

If read sets intersect, but write sets don't, also OK to run in parallel

If write set of one intersections write or read set of other, then trickier.

Consider banking example:

```
xfer(A, B, amt):
  begin
  A = A - amt
  B = B + amt
  commit

int(rate):
  begin
  for each account x:
    x = x*(1+rate)
  commit
```

[show slide]

What happens if we have A=100, B=50, and concurrent xfer(A,B,10) & int(0.1)?
Serial executions:

```
.. -> [xfer] -> A=90,  B=60 -> [int]  -> A=99,  B=66
.. -> [int]  -> A=110, B=55 -> [xfer] -> A=100, B=65
```

According to serializability def'n, either outcome is OK if xfer() & int() are executed concurrently.

What could happen if we run these operations concurrently?

```
xfer:        int:
  RA [100]
  WA [90]
               RA [90]
               WA [99]
  RB [50]
  WB [60]
               RB [60]
               WB [66]
```

This is called a "schedule" for these two transactions.

The two transactions didn't run serially, but is this schedule serializable?   Yes: outcome is the same as if they ran sequentially.

What about:

```
  xfer:       int:
   RA [100] //before WA in int
               RA [100]  //before WA in xfer
   WA [90] //before WA in int
               WA [110]  //after WA in xfer
   RB [50]
   WB [60]
               RB [60]
               WB [66]
```

Not serializable: could not have possibly gotten (A=110,B=66) serially.

[show slide]

Suppose we have two transactions T1 & T2. For a read of object o in T1, define *conflicts* to be all writes of o in T2. For a write of object o in T1, define *conflicts* to be all reads or writes of o in T2.

Formally, a schedule is serializable if, for two transactions T1 & T2, if:

for all conflicts in T1, every conflicting read or write is ordered before the operation it conflicts with in T2,

OR

for all conflicts in T2, every conflicting read or write is ordered before the operation it conflicts with in T1

In example, neither xfer nor int sees each other's write to A, which couldn't have happened in a serializable schedule

<u>So how can we ensure we get serializability?</u>

Introduce a locking protocol that our transaction system follows as it runs to ensure a serializable execution

Plan: Associate a lock with every variable; grab lock before accessing var.
[ slide: locking protocol ]

Q: When should we release?

Should not release right after read/write operation.

Can we release after txn is done with some variable (e.g., A or B)?

```
  xfer:        int:
   RA [100]              \ T1 holds A.lock
   WA [90]               /
               RA [90]      \ T2 holds A.lock
               WA [99]      /
               RB [50]      \ T2 holds B.lock
               WB [55]      /
   RB [55]               \ T1 holds B.lock
   WB [65]               /
```

No good: (99,65) is not serial-equivalent.

Violates conflict serializability rule since xfer does WA before int reads it, but int does WB before xfer reads it.

If we wait until after commit to release, we will be good.

This ensures our conflict serializability goal, since for the first conflicting operation in T1 (RA):

- either T2 already has lock (in which case T2 will complete all of its conflict operations before T1 does any of them),

or

- T1 will get the lock, in which case T2 will have to wait for T1 to finish all of it's conflicting operations.

Demo: waiting transactions

```
postgres -D /usr/local/var/postgres
test=# select * from accounts;
 username | balance
----------+---------
 mike     |     100
 sam      |     200
(2 rows)
```

one terminal (blue):

```
      begin; update accounts set balance=balance+5 where
username='mike';
```

other terminal (red):

```
      begin; update accounts set balance=balance-10 where
username='mike';
```

transaction in red terminal hangs:
  why?  because it conflicts with T1
  would result in non-serializable behavior if it didnt; must block
abort blue transaction
  2nd one commits
  cool!

Hands-on: several isolation levels in Postgres
  Snapshot isolation (which is slightly different from serializability as defined above)
  Read committed isolation  (which is weaker than snapshot and serializability)
  I'll pretend that Postgres is using locking, even though it isn't for reads
  (it use an optimistic multiversion scheme; see textbook 9.4.3).

<u>What's a problem we can have here?</u>

**Deadlock**!

Terminal 1:

```
begin; update accounts set balance=balance-10 where
username='sam';
update accounts set balance=balance-10 where username='mike';
```

Terminal 2:

```
begin; update accounts set balance=balance+5 where
username='mike';
update accounts set balance=balance+5 where username='sam';
```

What is the problem here?

Terminal 1 is waiting for terminal 2, and terminal 2 is waiting for terminal 1.

What do we think Postgres will do?

(Kill one)

How did it do that?

```
ERROR:  deadlock detected
DETAIL:  Process 2101 waits for ShareLock on transaction 1008;
blocked by process 2099.
Process 2099 waits for ShareLock on transaction 1007; blocked by
process 2101.
HINT:  See server log for query details.
```

You can see what it did — it detected that T1 was waiting for T2, and T2 was waiting for T1.

“Waits for graph” — graph of which other transaction a transaction is waiting to have complete

```
     “sam”
T1 — — > T2
    <— —
      “mike”
```

When there is a cycle in this graph, it indicates a deadlock

To resolve, kill one transaction

Ok - -couple of things that are inefficient about what we have so far.

<u>Optimization 1) First, transactions always wait for each other, even when they are just reading the same data.</u>

Conflicts are only when at least one operation is a writer.

Multiple readers should be able to operate on the same data concurrently.

Introduce a new idea — reader/writer locks

[ slide: locking with rw locks ]

A reader can acquire lock at the same time as other readers (but no writers).

A writer can acquire lock only if no other readers or writers.

<u>Optimization 2) Release locks early</u>

Suppose we knew all the objects we were going to access and when we were done with them — then we could lock them up front, and release locks as we finish w/ objects.

This will be fine because all conflicting operations in other transactions will still run after all conflicting operations in our transactions.

Our earlier example:

```
   xfer:       int:
    RA [100]             \ T1 locks A and B — lock point
    WA [90]              /
     ...                 > T1 releases A.lock — it’s done w/ A
               RA [90]      \ T2 holds A.lock
               WA [99]      /
               lock B: wait
    RB [50]              \ T1 holds B.lock
    WB [60]              / T1 releases B.lock
               RB [60]      \ T2 holds B.lock, lock point, waited for T1
               WB [66]      /
```

This is called "**two-phase locking**" (2PL).
  Phase 1: acquire read and write locks, until transaction reaches lock point.
  Phase 2: release locks, after lock point & done with object.

If we don’t know locks up front, we can still release early if we know we are done, but programmer would have to tell us this.

<u>What is problem with releasing write locks early?</u>

What if xfer() aborts?  Now int will have read uncommitted data, and will have to abort — this is called *cascading aborts*.

In practice most systems implement what is known as **strict 2PL**, i.e., they hold write locks til end of transaction.

<u>Optimization 3:  relaxed consistency</u>

Don’t always need to enforce serializability

Ex, read committed

[show slide]

W/ serializable, T1 will wait for T2

W/ read committed, T2 will release read lock after select, which will allow T1 to run;  T2 will see
  T1’s update (but do we care)?

**Recap**: saw how reason about concurrent action using conflicts, and how to ensure serializability

2PL protocol allows transaction system to implement locking for us

Have now developed a complete protocol for providing atomicity in the presence of concurrent operations and crashes. Next time will start talking about how this works in a multi-node system.

<u>What to do about external actions, e.g., dispensed cash, or fire a missile, or shipped a package?</u> Can't undo those things (usually). And aren't idempotent, so don't want to redo them!

Require special care. One approach: wait for software that controls action to commit, then take the action afterwards, but have some special way to detect if action already happened (e.g., sensor of amount of cash, or whether missile bay is empty / full.)