**6.033 Lecture 17 -- Logging** **April 8, 2009**

Sam Madden

Last time: introduced atomicity

two key ideas:
- all or nothing
- isolation

saw transactions, which are a powerful way to ensure atomicity

```
begin
    xfer(A,B,10)
    debit(B,10)
commit (or abort)
```

started talking about

**all-or-nothing without isolation**
(isolation next time)

saw idea of shadow copies, where you write changes into a secondary copy, and then atomically install that copy

(show slide)

this gets at the key idea of atomicity -- the golden rule

never overwrite the only copy

most atomicity schemes work kind of like this -- you make changes to some separate copy of the data, and then have a way to switch over to that new copy when needed.

if you crash mid-way through making changes, old copy is still there. typically when you crash, because transaction was incomplete, you just revert to old copy.

shadow copy approach has some limitations; what are they?

- only works for one file at a time (might be able to fix using, e.g., shadow directories)

- requires us to make a whole copy of the file

shadow copies are used in many places -- e.g., text editors , (emacs)

today we are going to learn about a general method for all-or-nothing atomicity that addresses these limitations -- **logging**.

basic idea is that after every change, you record the before and after value of the object you changed.

let's work with bank account balances;  suppose we have the following table in memory

account id  balance
A  100
B  50

suppose I do
    begin
    debit (A,10)
    debit (B, 20)
    end

    begin
    deposit (A,50)

what kinds of things do I keep in the log:
    transaction begin / commit / abort
        transaction id
    updates
        tid
        variable updated
        before (undo) / after (redo) state

log:

    begin 1
    update 1 A before: 100;  after: 90
    update  1 A before: 50; after: 30
    commit 1

    begin 2
    update 2 A before: 90; after 140
    crash!

this log now records everything I need to determine the value of an object

```
read(var, log):
        cset = {}
        for r in log(len-1) ... log(0)
                if r is commit
                        cset = cset U r.TID
                if r is update
                        if (r.TID in cset and r.var = var)
                                return r.after
```

but this is really slow, since i have to scan the log after every read

what is the commit point?  (when commit record goes to log)

(writes, however, are fast, since i just append to the log, probably sequentially)

how to make this faster?  keep two copies -- "cell storage" -- e.g., the actual table contents, in addition to log on disk.  reads can just read current value from cell storage, and writes can go to both places.

```
read(var)
        return read(cell-storage,var)

write(TID,var, newval)
        append(log,TID, var, cell-storage(var), newval)
        write(cell-storage,var,newval)
```

let's see what happens:

state:

| | |
|---|---|
| A | 100 --> 90 --> 140 |
| B | 50 --> 30 -> 60 -> 30 -> 40 |

| | |
|---|---|
| begin | log: |
| debit (A,10) | begin T1 |
| debit (B, 20) | update(A, T1, 100, 90) |
| commit | update(B, T1, 50, 30) |
| | commit T1 |
| | |
| begin | begin T2 |

| | |
|---|---|
| deposit (B,30) | update(B,T2, 30, 60) |
| abort | abort T2 |
| | |
| begin | begin T3 |
| debit (B,20) | update(T3,B, 30, 40) |
| commit | commit T3 |
| | |
| begin | begin T4 |
| deposit (A,50) | update(T4,A,90,140) |
| crash | abort T4 |

After crash, cell storage has the wrong value for A in it. What should we do?

Need to add a recover procedure, and need to do a backwards pass on the log to undo the effects of uncommitted transactions. (undo the "losers")

Anything else we need to do?

Also need to do a forward pass on the log, to redo the effects of cell storage writes in case we crashed before writing cell storage but after doing append. (redo the "winners")

```
recover(log):
        cset = {}
        for each record r in log[len(log)-1 ... 0]  //UNDO
                if r is commit, cset = cset U r.TID
                if r is update and r.TID not in cset:
                        write(cell-storage, r.var, r.before)
        for each record r in log[0...len(log)-1]  //REDO
                if r is update and r.TID in cset:
                        write(cell-storage, r.var, r.after)
```

Why backwards pass for UNDO? Have to do a scan to determine cset. If we do this scan backwards, we can combine with UNDO.

Why forwards pass to REDO? Want the cell-storage to show the results of the most recent committed transaction.

other variants possible -- e.g., redo then undo, forward pass for undo (with previous scan to determine c set), etc.

what if I crash during recovery?

OK -- recovery is idempotent

example:  after crash:  A=140;  B=40

cset = {1, 3}

UNDO

A -> 90

B -> 30

REDO

A -> 90

B -> 30

B -> 40

T1 and T3 committed, and in that order.

Optimization 1: "Don't REDO updates already in cell storage"

Possible to optimize this somewhat by not REDO updates already reflected in cell storage.  Simplest way to do this is to record on each cell a log record ID of the most recent log record reflected in the cell.  Diagram:

UPDATE(tid,logid,var,before,after)

(in our example, all we needed to actually do was UNDO A->90)

Q:  Why did I write the log record before updating the cell storage?  What would have happened if I had done these in opposite order?

cell storage would have value but might not be in log

recovery wouldn't properly undo effects of updates

This is called "write ahead logging" -- always write the log before you update the data

Optimization 2:

Defer cell storage updates

Can keep a cache of cell storage in memory, and flush whenever we want.  Doesn't

really matter, as long as log record is written first, since logging will ensure that we REDO any cell storage updates for committed transactions.

```
read(var):
        if var in cache
                return cache[var]
        else
                val = read(cell-storage, var)
                add (var,val) to cache
                return val

write(var,val)
        append(log,TID, var, cell-storage(var), newval)
        add (var,val) to cache
```

Optimization 3:

Truncate the log.

If log grows forever, recovery takes a long time, and disk space is wasted.

Q: What prefix of the log can be discarded?

A: Any part about completed transactions whose changes are definitely reflected in cell storage (because we will never need to reapply those changes.)

Idea:

checkpoint:

```
write checkpoint record
flush cell storage
truncate log prior to checkpoint
```

Write checkpoint.
Write any outstanding updates to cell storage to disk. This state written to disk definitely reflects all updates to log records prior to the checkpoint. Truncate the log prior to the checkpoint.

Most databases implement truncation by segmenting the log into a number of files that are chained together. Truncation involves deleting files that are no longer used.

Diagram:

Logging --

- good read performance -- cell storage, plus in memory cache
- decent write performance --
  - have to write log, but can write sequentially;
  - cell storage written lazily
- recovery is fast -- only read non-truncated part of log

Limitations --

- external actions -- "dispense money", "fire missile"
  - cannot make them atomic with disk updates
- writing everything twice -- slower?

Next time --

- isolation of multiple concurrent transactions