6.830 Lecture 15 -- Recovery **10/29/2014**

Have been talking about transactions

Transactions -- what do they do?

Awesomely powerful abstraction -- programmer can run arbitrary mixture of commands that read and modify data and, without worrying about locking, threads, etc get serial equivalence and high degree of parallelism.

Atomicity
Consistency
Isolation
Durability

Today going to talk about recovery, but first a brief interlude about granularity of locks…

So far, we've used an abstract model of "objects" being read, written, and locked, e.g.:

RX
WX

But not clear what "X" is here.

In practice, could be a tuple, page, table, or whole database.

What is the tradeoff here? Why not make it as small as possible?

A transaction that touches a lot of records will have to acquire a lot of locks!

So what is the problem with allowing some transactions to lock tables and others to lock tuples?

Shouldn't be allowed read access to a tuple if some other transaction has write access to the table.

So what is the solution?

Create a "locking hierarchy", e.g.:

Tables
|
Pages
|
Tuples

Introduce "Intention Locks" -- indicating that a transaction is going to read / write some part of a table.

Require that a transaction hold an intention lock on higher indicate that a transaction intends to read/write something at a lower level in the hierarchy.

E.g., to lock a tuple X in page P in Table T in X mode, I first need to hold Intention X (IX) locks on P and T.

Read (Record X) --> IX(Table X); IX(Page X); X(Record X)
Release in opposite order.

IX/IS locks prevent people locking just the upper levels of the hierarchy from conflicting with transactions locking lower levels of the hierarchy.

Lock compatibility table

		S	X	IX	IS
S	I	Y	N	N	Y
X	I	N	N	N	N
IX	I	N	N	Y	Y
IS	I	Y	N	Y	Y

Basically can't acquire an IX or IS lock if someone has X lock on upper levels of the hierarchy.

E.g., if T1 wants to update the entire table, it will acquire an X lock on the table. This will prevent readers and writers of individual tuples from being able to go forward because they cannot acquire IS/IX locks.

Recovery:

Recovery is about:

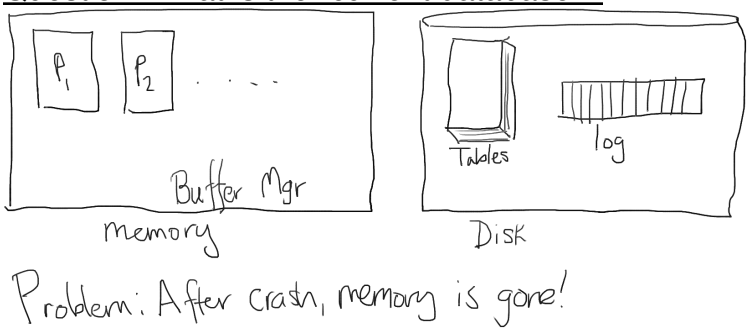
- ensuring atomicity by giving us a way to roll back aborted xactions
- ensuring durability -- e.g., committed xactions actually appear on stable storage after a crash

when would this be a problem?
(if we don't always flush all pages at commit time)

- ensuring that uncommitted xactions effects don't appear on stable storage after a crash

when would this be a problem?
(if we sometimes flush pages before commit time)

Question: What is the "current database"?



(Some combination of stuff in buffer manager, in log, and in database.)

Log basically always makes it possible to restore to a "transaction consistent" state.

Memory may include updates that aren't committed.

Disk may include updates that aren't committed, as well as garbage (partially written pages, or one of several pages that comprise an update.)

Recovery is about restoring disk to a "transaction consistent" state, which we typically do after a crash. This allows query execution to continue and be assured that the data that transactions read is committed.

Basic idea is to store two copies -- one that reflects the state before modification (so we can rollback to it if a transaction does commit, and one that reflects the change.)

Soln: **Log Based Recovery** (as opposed to e.g., shadow pages)

What rule do we have to follow when writing log records?
(Write ahead logging!)

Write log records before you write any update to disk.

Log records for a xaction must be on disk before you can commit.
(Only "force" log at commit time)

Why?

Otherwise, you might update a page as a part of an uncommitted xaction, crash (which should cause you to rollback that update), but not have any way to tell that you updated the page.

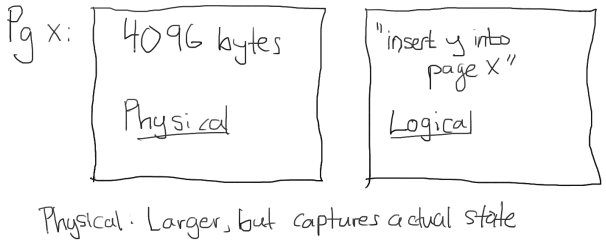
Note that log only reflects WRITES -- READS do not need to be logged.

Idea with logging is to write what you planned to do before you do it, and to leave enough info in
the log such that you can figure out whether you did it or not.

Effectively this means that we again have two copies of data -- the log records plus the current on-disk state, which together are sufficient to get the before or after state.

What kinds of records appear in a log?

SOT - LSN, transaction id (LSN is monotonically increasing sequence number)
EOT - LSN, transaction id / commit or abort
UNDO - LSN, before image or logical update that allows us to remove the effects of an action
example:



REDO - LSN, after image or logical update that allows us to remove the effects of an action
CHECKPOINT -- LSN, current state ofallow us to limit how much we have to UNDO/ REDO
CLR -- allows us to restart recovery

Buffer Manager -- what does it have to do with recovery?

Need it to guarantee that each object is only touched by one outstanding xaction at a time. Otherwise, it may be hard to ensure that we can recover (since undoing effects of one xaction affect results written by another, etc.)

2PL protocol guarantees that only one transaction updates something at a time.

If dirty pages are never written to disk, then we never need to undo any actions at recovery time.

Why do we sometimes then want to write out dirty pages?
Because if we don't those pages are locked in memory. This is STEAL vs !STEAL.

If modified pages are always written to disk before the commit record, then we will never need to
REDO any work.

This is FORCE vs !FORCE.

Why is FORCE not always a good idea?

It's expensive (lots of writes at EOT), and if a page is modified by
many transactions, may be wasteful.

	FORCE	!FORCE
STEAL	UNDO	UNDO/REDO
!STEAL	UNDO?	REDO

FORCE by itself implies some UNDO (since you eventually write some dirty data before commit time.)

If we don't do FORCE the only non-async I/O is logging, which is purely sequential!

Still -- building a FORCE/!STEAL DB is much easier than a !FORCE/STEAL DB.

SimpleDB is FORCE/!STEAL, plus will not crash during FORCE, so does not need logging or recovery! We will relax this assumption in Lab 5.

Almost all commercial databases do !FORCE/STEAL for performance reasons.

So the main idea of recovery in a !FORCE/STEAL database is to:

- undo losing transactions
- redo winning transactions

Determine winners and losers by scanning the log for SOT with EOT records.
Determine what to UNDO from loser records in the log. Losers are those with SOT and not EOT.

Determine what to REDO by checking most recent update applied to winner transactions identified in the log scan (need to store most recent update on pages.)

Example:

Suppose we have 3 transactions, using !FORCE, STEAL

T1 writes A, commits

T2 writes B, aborts
T3 write C, system crashes

T1 ----------W(A)---------- C
T2----------W(B)---------- A
T3--------W(C)---------- crash!

Log:

S(T1) S(T2) W(A) S(T3) W(C) W(B) C(T1) A(T2)

After crash:
- memory is empty
- log is as above
- database pages ("cell store") is in indeterminate state

What do we do?

(Lots of options ....) -- have to REDO A, UNDO B, UNDO C, in some order, but could: UNDO, then REDO
or REDO, then UNDO (making sure we don't REDO undone stuff or UNDO redone stuff)

ARIES Protocol

Aries Gossip.

Considered THE standard in logging protocols. Not clear that its much different than what all commercial databases do, but they were the first to write it down.

Some discontent when it was published as others thought that this stuff was common sense, that it had been codified elsewhere, etc. -- but it has survived as the protocol of note.

What's interesting about it?

Specifies all of the details,
Assumes !FORCE/STEAL
Shows how its possible to make recovery recoverable,
Shows how to use logical UNDO logging,
Shows how to handle nested transactions (which we won't talk about),
Shows how to make fuzzy checkpoints work for real.

Also incredibly painful to read about.
Go through the details next time.