6.830 Lecture 15 **11/1/2017**

Recovery continued

Last time -- saw the basics of logging and recovery -- today we are going to discuss the ARIES protocol. First discuss two logging concepts: FORCE/STEAL

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

ARIES is the "gold standard" of log-based recovery in database systems, and one example of a way to implement !FORCE/STEAL

What are goals of ARIES:

- support for !FORCE/STEAL (UNDO and REDO)
- efficiency at runtime -- esp. lightweight checkpoints that don't stall system
- support for complex operations (e.g., operations that modify indexes and disk pages)
- support for escrow operations (e.g., increments / decrements)
- support for recovery even if system crashes while recovering

How does ARIES work?

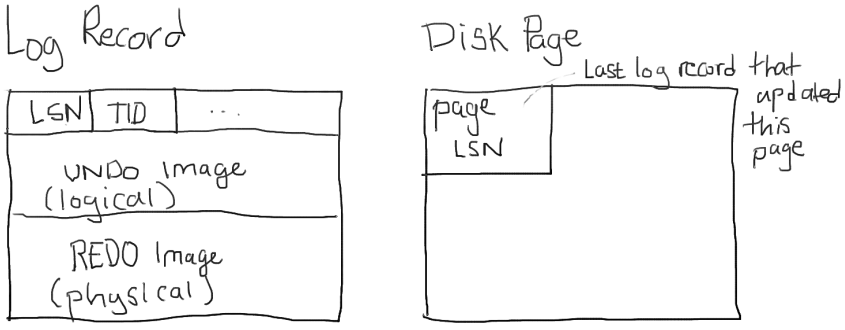
3 passes:

- analysis pass, to see what needs to be done (plays log forward)

- redo pass, to make sure the disk reflects any updates that are in the log but not on disk, including those that belong to xactions that will eventually be rolled back!

why do this? so we can be sure we are in "action consistent" state -- which will allow logical undo. (also plays log forward) -- "repeating history"

- undo pass, to remove the actions of any losing xactions
(plays log backward)
Update records have both UNDO and REDO information.
Show diagram.



- Every log record has an LSN associated with it.

- Every time a page is written, the latest LSN associated with that log record is included as the pageLSN.

UNDO data is logical, REDO data is physical.

Why?

Must do physical REDO, since we can't guarantee that the database is in a consistent state (so, e.g., logging "INSERT VALUE X INTO TABLE Y" might not be a good idea, because X might be reflected in an index but not the table, or vice versa, if system crashed halfway through operation.)

We can do logical UNDO, because after REDO we know that things are "action consistent". In fact, we MUST do logical UNDO. Why?

Because we only UNDO some actions, and physical logging of UNDOs of the form, e.g., "split page x of index y" might not be the right thing to do anymore in terms of index management or invariant maintenance.

Why don't we have to worry about this during redo?

Because we "repeat history" and replay everything, which means any physical modifications made to the database since last time will still be correct.

Pretty clever.

Normal Operation

Two data structures are maintained --

1) Transaction table -- list of active transactions, as well as "lastLSN" -- most recent log record written by that transaction.

- Log records for a given transaction are chained together using a "prevLSN" field that points to the previous log record from that transaction.

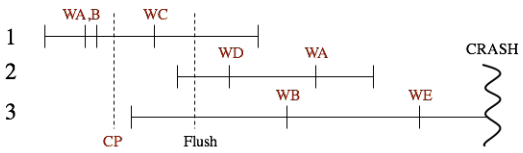
2) Dirty page table -- List of pages that have been modified and not yet written to disk, as well "recoveryLSN" of log record that first dirtied each page.

Dirty pages are periodically flushed to disk by a background process (flushes are not logged)

Checkpoints taken periodically that simply record the state of the DPT and TT. Will see how these help use later (limit the extent of log we have to replay.)

Recovery Operation

Example:



LSN	Type	Tid	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A
3	UP	1	2	B
4	CP			
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	UP	2	7	D
9	EOT	1	6	
10	UP	3	5	B
11	UP	2	8	A
12	EOT	2	11	
13	UP	3	10	E

xactionTable

lastLSN	TID
13	3

dirtyPgTable

pgNo	recLSN
D	8
B	10
A	11
E	13

Checkpoint

xactionTable	3 - 1
dirtyPgTable	A - 2, B - 3

Disk

Page	pageLSN
A	2
B	3
C	6
D	?
E	?

Checkpoints just consist of transaction table and dirty page table as of some point in time -- don't actually have to flush pages. More later.

Analysis Pass:

What's the goal of the analysis pass? What do we want at the end?

To reconstruct, as much as possible, the state of the transaction table and the dirty page table at the time the crash occurred.

Play log forward, adding new xactions to the transaction table and removing xactions from the xaction table when they commit, updating the lastLSN.

Where do we begin?

- beginning of log (ok, but will require us to scan a lot of log, maybe)
- last checkpoint! (how do we find it?)

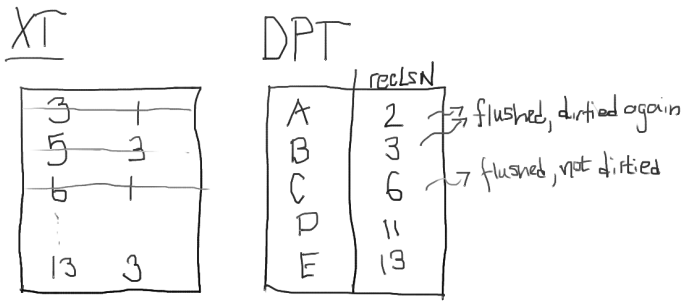
Also update the dirty page table as updates are encountered, adding pages that are modified to it.

After analysis, what can we say about dirty page table and transaction table?

- Transaction table is stuff that needs to be undone.
- Dirty pages is a conservative list of pages that need to be REDOne

why is it conservative?
(don't actually know what is on disk -- some pages may already have updates applied.)

Example:



<u>REDO pass:</u>

To figure out where to start REDO from.

REDO Lsn - start of recovery == min(recoveryLSN of all pages in dirtyPageTable)
(e.g., 2)

Scan forwards. Why forwards?

Example:

Don't need to write 2,3,6, since those updates have already been applied to pages

How do I know this? Play forward, and for each update, <u>test to see if I should apply</u>.

Under what circumstances do I not need to apply an update?

- if the page is not in the dirtyPageTable
  (will happen if we flushed the page prior to CP, and didn't dirty again)

ex *suppose we had flushed B prior to CP but not dirtied it again*

- if the LSN of the log record is less than the recoveryLSN of the page in the dirty page table.
(page flushed prior to CP and re-dirtited prior to CP)
ex *suppose we had flushed B and dirtied again prior to CP*

- if the LSN of the log record is less than or equal to the pageLSN of the page
(page updated by a later action and flushed to disk -- as in 2,6 in our example)

Note that the first two conditions can be tested without even reading the page from disk. These are important optimizations because reading back from disk requires seeks, which will slow down recovery.

Updates don't necessarily write back to disk (recovered pages can be in the buffer pool.)

UNDO pass:

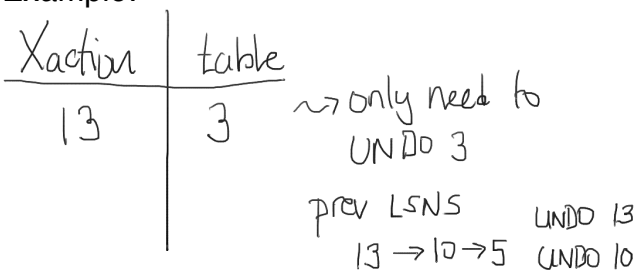
From start of log, play backwards, UNDOing actions from the loser transactions (in the transaction table.) Why backwards?

Can figure out which is the next log record to undo by taking the max of the lastLSN field, and updating the lastLSN field with the prevLSN field of every record as I read it in.

Are there any circumstances under which I do not need to apply an UNDO?
(No, because we repeated history so we always UNDO all actions of losers.)

At end of this, I have a "transaction consistent" database -- only winner transactions are there, and transaction table is empty.

Example:

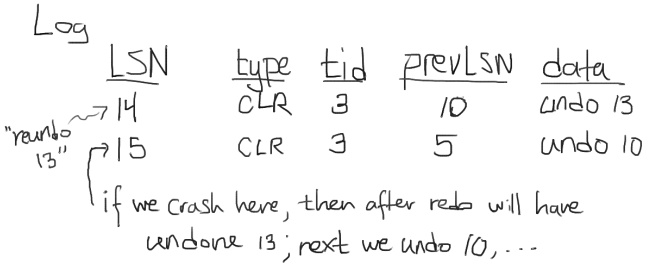


Last wrinkle -- as we're undoing, we write CLRs before every UNDO. CLRs point to the next action that needs to be UNDOne (the undoNextLSN) for a given page, and also log the effect of the undo action. During REDO, we replay this action, and then during the UNDO phase, we follow the undoNextLSN whenever we reach a CLR.

Why are CLRS necessary?

Because UNDO Is logical, and we don't check if records have already been UNDONE. Could get into trouble if re-undid some logical operation.

Example :



Are there alternatives to log based recovery?

e.g., command logging, or recovery from a network backup

Additional Questions:

How do we support transaction Abort?
- Just use UNDO Logic!

Recap:
NO FORCE, STEAL logging
Use write ahead logging protocol
Must FORCE log on COMMIT
Periodically take (lightweight) checkpoints
Asynchronously flush disk pages (without logging)

ARIES features:
Complicated by operational logging / escrow locks
Complicated by desire to not exclusive locks on index pages

Physiological logging
CLRs

Question: How does logging impact performance?
- Writing to separate log disk, sequentially, so hope is that the throughput is good
- Only have to force write log at COMMIT time.

Doesn't that limit us to (say) 1000 transactions /sec if it takes 1 ms to flush the log? (Group commit)

Of course, we are giving up some performance.

When can log be truncated?

(When all dirty pages reflected in it are on disk)

Note that this requires that dirty pages be written back regularly.

Other types of failures (Disasters)

So far -- just focused on what if the database crashes

App fails:
    Not an issue in this class

Comm. Failures:
    We will come back to these when we deal with multi-processor issues

Machine room fails (fire, flood, earthquake, 9/11, …)

Most of these failures dealt with by logging (journaling) – next time.

Disasters are a bigger problem.

1970's solution to disasters:

    Write a log (journal) of changes
    Spool the log to tape
    Hire iron mountain to put the tapes under the mountain
    Buy IBM hardware – they were heroic in getting you back up in small numbers of days

1980's solution
    Hire Comdisco to put the tapes at their machine room in Atlanta
    Send your system programmers to Atlanta, restore the log tapes, divert comm to Atlanta
    Back up in small numbers of hours
    (average CIO conducts a disaster drill more than once a year)

2000's solution

Run a dedicated “hot standby” in Atlanta
Fail over in small numbers of minutes

Driven by plummeting cost of hardware and the increasing cost of downtime (thousands of dollars per minute)

In all cases, you tend to lose a few transactions. Too costly to lower the probability to zero. Write disgruntled users a check!!!