COURSE 5

Database Recovery:

Buffer Management

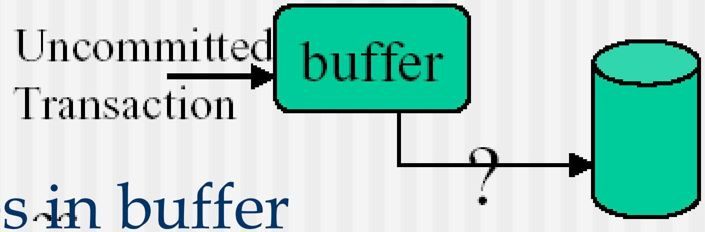
ARIES

A Few Words on Buffer Management

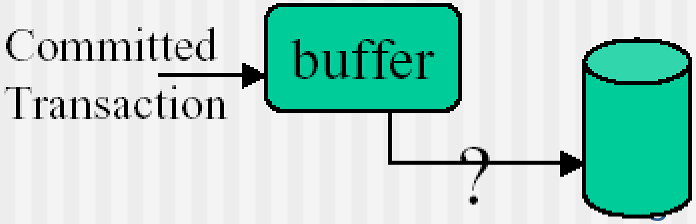
- Data access in terms of disk blocks
  - Each block must be brought into buffer pool if not already there
- May require replacing existing pages if pool is full
  - Remember replacement strategies?
- Use *dirty bit* to determine if page contents have been modified
  - Write back if dirty bit is set, o.w. discard
- Use *pin/unpin bit* to determine if page is candidate for replacement
- Two main strategies for flushing modified buffers back to disk
  - *In-place updating:* data is written back to same original disk location
  - *Shadowing:* before and after image of data on disk (BFIM/AFIM)

Recovery Manager ⇔ Buffer Manager

- Can a Buffer Manager decide to write some of the buffer pages being accessed by a transaction into stable storage or does it wait for Recovery Manager to instruct it?
  - **Steal / no-steal** decision
  - No-steal means RM pins pages in buffer



- Does the Recovery Manager force the Buffer Manager to write certain buffer pages in stable database at the end of a transaction's execution?
  - **Force / no-force** decision



Handling the Buffer Pool

- Force every write to disk?
  - Poor response time.
  - But provides durability.
- Steal buffer-pool frames from uncommited transactions?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?

	No Steal	Steal
Force	Trivial	
No Force		Desired

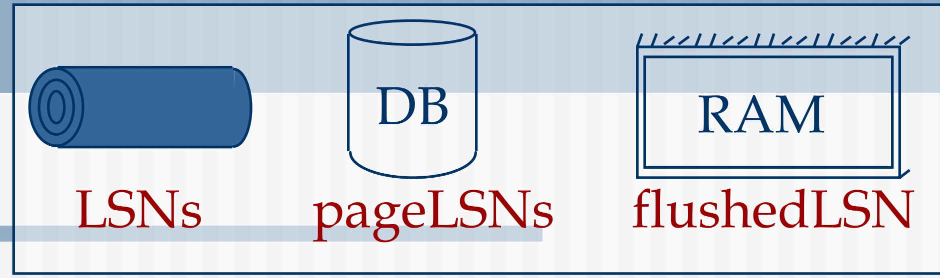
Possible Execution Strategies

- **Steal / No-force**
  - BM may have written some of the updated pages into disk. RM writes a commit
- Steal / force
  - BM may have written some of the updated pages into disk. RM issues a *flush* and writes a commit
- No-steal / no-force
  - None of the updated pages have been written. RM writes a commit and sends unpin to BM for all pinned pages.
- No-steal / force
  - None of the updated pages have been written. RM issues a *flush* and writes a commit

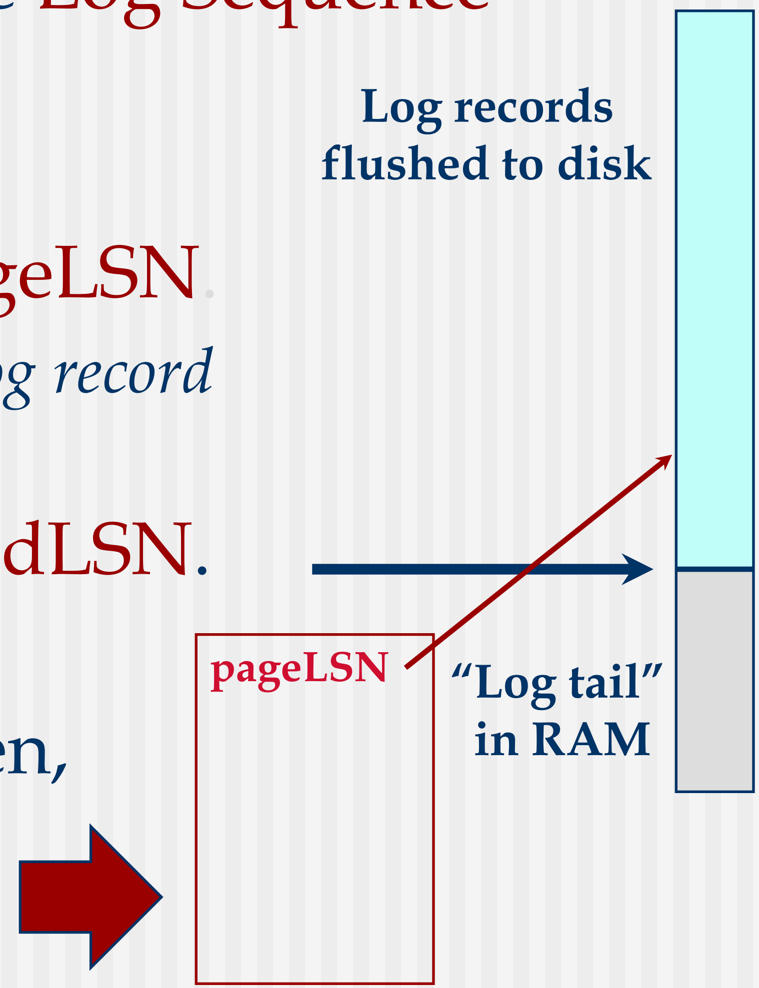
More on Steal and Force

- **STEAL** (why enforcing *Atomicity* is hard)
  - *To steal frame F:* Current page in F (say P) is written to disk; some transaction holds lock on P.
    - What if the transaction with the lock on P aborts?
    - Must remember the old value of P at steal time (to support UNDOing the write to page P).
- **NO FORCE** (why enforcing *Durability* is hard)
  - What if system crashes before a modified page is written to disk?
  - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.

WAL & the Log



- Each log record has a unique Log Sequence Number (LSN).
  - LSNs always increasing.
- Each *data page* contains a pageLSN.
  - The LSN of the most recent *log record* for an update to that page.
- System keeps track of flushedLSN.
  - The max LSN flushed so far.
- WAL: *Before* a page is written,
  - pageLSN ≤ flushedLSN



Log Records

LogRecord fields:

LSN
prevLSN
TransID
type
pageID
length
offset
before-image
after-image

update records only

Possible log record types:

- **Update**
- **Commit**
- **Abort**
- **Checkpoint**
- **End** (signifies end of commit or abort)
- Compensation Log Records (CLRs)
  – for UNDO actions

Compensation Log Record (CLR)

- Written just before change recorded in one update log record is undone
- Contains a field called undoNextLSN
  - LSN of next log record to be undone for the transaction that wrote the update record
  - Set to prevLSN of the update log record
- Indicate which actions have already been undone
- Prevent undoing same action twice

Other Log-Related State

- Transaction Table:
  - One entry per active transaction.
  - Contains XID, status (running / committed / aborted), and lastLSN.
- Dirty Page Table:
  - One entry per dirty page in buffer pool.
  - Contains recLSN -- the LSN of the log record which ***first*** caused the page to be dirty.

Normal Execution of a Transaction

- Series of reads & writes, followed by commit or abort
  - We will assume that write is atomic on disk.
    - In practice, additional details to deal with non-atomic writes.
- Strict 2PL.
- STEAL, NO-FORCE buffer management, with Write-Ahead Logging.

The Big Picture: What's Stored Where

LogRecords

- prevLSN
- XID
- type
- pageID
- length
- offset
- before-image
- after-image

Data pages

- each with a pageLSN

Master record

Transaction Table

- lastLSN
- status

Dirty Page Table

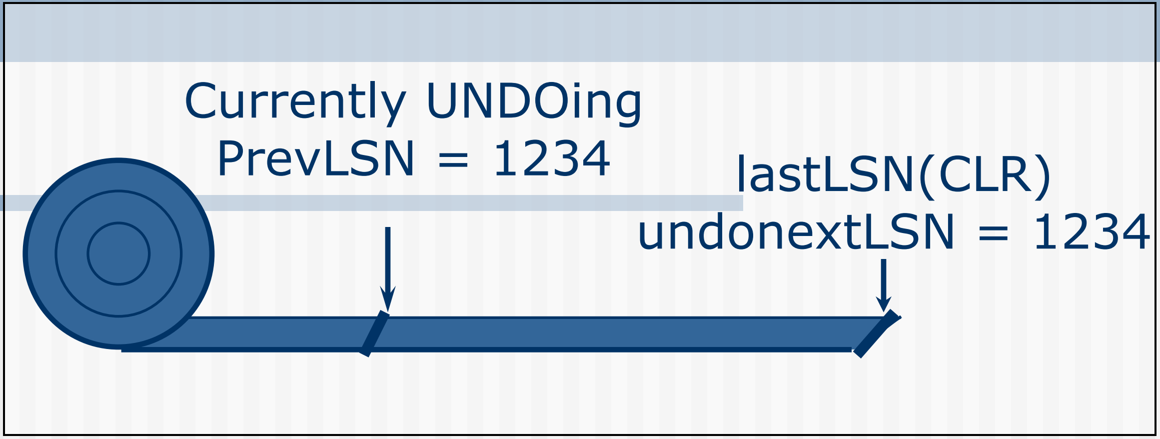
- recLSN

flushedLSN

Simple Transaction Abort

- For now, consider an explicit abort of a transaction (*no crash involved*).
- We want to "play back" the log in reverse order, *UNDO*ing updates.
  - Get lastLSN of transaction from trans. table.
  - Can follow chain of log records backward via the prevLSN field.
  - Before starting UNDO, write an *Abort* log record
    - For recovering from crash during UNDO!

Abort (cont)



- To perform UNDO, must have a lock on data!
- Before restoring old value of a page, write a CLR:
  - You continue logging while you UNDO!!
  - CLR has one extra field: undonextLSN
    - Points to the next LSN to undo (i.e. the prevLSN of the record we're currently undoing).
  - CLRs *never* Undone (but they might be Redone when repeating history: guarantees Atomicity!)
- At end of UNDO, write an end log record.

Transaction Commit

- Write commit record to log.
- All log records up to transaction's lastLSN are flushed.
  - Guarantees that flushedLSN $\geq$ lastLSN.
  - Note that log flushes are sequential, synchronous writes to disk.
  - Many log records per log page.
- Commit() returns.
- Write end record to log.

Phases of ARIES

= Advanced Recovery and Integrated Extraction System

- *Analysis*: Scan the log forward (from the most recent *checkpoint*) to identify all transactions that were active, and all dirty pages in the buffer pool at the time of the crash
- *Redo*: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
- *Undo*: The writes of all transactions that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)

Crash Recovery: Big Picture

- Start from a checkpoint (found via master record).
- Three phases. Need to:
  - Figure out which transactions committed since checkpoint, which failed (Analysis).
  - REDO ***all*** actions (repeat history)
  - UNDO effects of failed transactions.

Recovery: The Analysis Phase

- Reconstruct state at checkpoint.
  - Set of active transactions and dirty pages
- Scan log forward from checkpoint.
  - End record: Remove transaction from transaction table.
  - Other records: Add transaction to transaction table, set lastLSN=LSN, change transaction status on commit.
  - Update record: If P not in Dirty Page Table,
    - Add P to D.P.T., set its recLSN=LSN.

Recovery: The REDO Phase

- We *repeat History* to reconstruct state at crash:
  - Reapply *all* updates (even of aborted transactions!), redo CLRs.
- Scan forward from log rec containing smallest recLSN in D.P.T. For each CLR or update log rec LSN, REDO the action unless:
  - Affected page is not in the Dirty Page Table, or
  - Affected page is in D.P.T., but has recLSN > LSN, or
  - pageLSN (in DB) ≥ LSN.
- To REDO an action:
  - Reapply logged action.
  - Set pageLSN to LSN. No additional logging!

Recovery: The UNDO Phase

ToUndo={ l | l a lastLSN of a "loser" transaction}

Repeat:

Choose largest LSN among ToUndo.

If this LSN is a CLR and undonextLSN==NULL

Write an End record for this transaction.

If this LSN is a CLR, and undonextLSN != NULL

Add undonextLSN to ToUndo

Else this LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo.

Until ToUndo is empty.

Example of Recovery

Trans Table
- lastLSN
- status

Dirty Page Table
- recLSN

flushedLSN

ToUndo

LSN	LOG
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART

prevLSNs

Example of Recovery

RAM

Trans Table

lastLSN

status

Dirty Page Table

recLSN

flushedLSN

ToUndo

LSN	LOG
00,05	begin_checkpoint, end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40,45	CLR: Undo T1 LSN 10, T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART
70	CLR: Undo T2 LSN 60
80,85	CLR: Undo T3 LSN 50, T3 end
	CRASH, RESTART
90	CLR: Undo T2 LSN 20, T2 end

undonextLSN

Additional Crash Issues

- System crash may occur during database recovery:
  - Apply redo and undo to a record either once only, or
  - Make redo and undo as idempotent operations
    - idempotent: Acting as if used only once, even if used multiple times
- How do you limit the amount of work in REDO?
  - Flush asynchronously in the background.
  - Watch "hot spots"!
- How do you limit the amount of work in UNDO?
  - Avoid long-running transactions.