Comp115: Databases

Crash Recovery

Instructor: Manos Athanassoulis

Review: The ACID properties

Atomicity: All actions in the transaction happen, or none happen.

Consistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.

Isolation: Execution of one transaction is isolated from that of other transactions.

Durability: If a transaction commits, its effects persist.

Question: which ones does the Recovery Manager help with?

Atomicity & Durability (and also used for Consistency-related rollbacks)

Motivation

Atomicity:

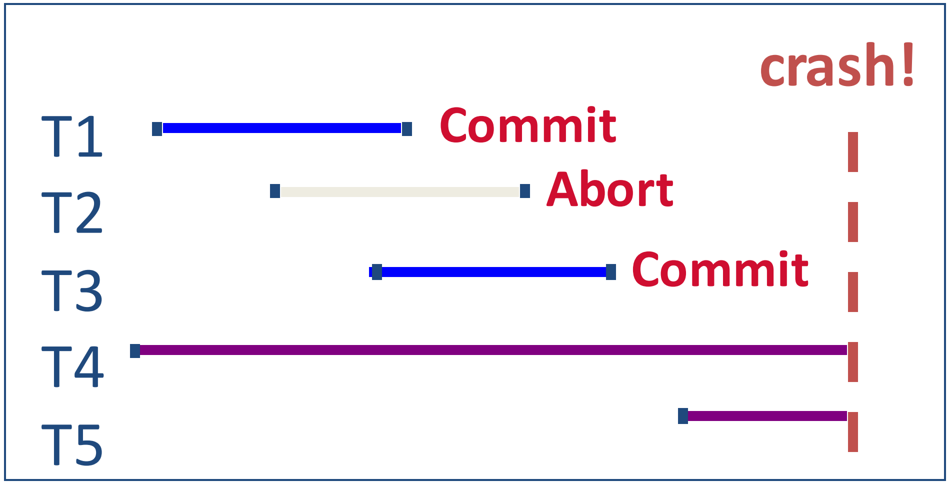
- Transactions may abort ("Rollback").

Durability:

- What if DBMS stops running? (Causes?)

Desired state after system restarts:

- T1 & T3 should be durable.
- T2, T4 & T5 should be aborted (effects should not be seen).



Assumptions

Concurrency control is in effect.

- Strict 2PL, in particular.

Updates are happening “in place”.

- i.e. data is overwritten on (deleted from) the actual page copies (not private copies).

Can you think of a simple scheme (requiring no logging) to guarantee Atomicity & Durability?

- What happens during normal execution (what is the minimum lock granularity)?
- What happens when a transaction commits?
- What happens when a transaction aborts?

Buffer Management Plays a Key Role

Force policy – make sure that every update is on disk before commit.

- Provides durability without REDO logging.
- But, can cause poor performance.

No Steal policy – don't allow buffer-pool frames with uncommited updates to overwrite committed data on disk.

- Useful for ensuring atomicity without UNDO logging.
- But can cause poor performance.

Of course, there are some nasty details for getting Force/NoSteal to work…

Preferred Policy: Steal/No-Force

More complicated but allows for highest performance

NO FORCE (complicates enforcing Durability)

- What if system crashes before a modified page written by a committed transaction makes it to disk?
- Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.

STEAL (complicates enforcing Atomicity)

- What if the transaction that performed updates aborts?
- What if system crashes before transaction is finished?
- Must remember the old value of P (to support UNDOing the write to page P).

Buffer Management summary

	No Steal	Steal
No Force		Fastest
Force	Slowest	

Performance Implications

	No Steal	Steal
No Force	No UNDO REDO	UNDO REDO
Force	No UNDO No REDO	UNDO No REDO

Logging/Recovery Implications

Basic Idea: Logging

Record REDO and UNDO information, for every update, in a *log*.

- Sequential writes to log (put it on a separate disk).
- Minimal info (diff) written to log, so multiple updates fit in a single log page.

Log: An ordered list of REDO/UNDO actions

- Log record contains:

  <XID, pageID, offset, length, old data, new data>

- and additional control info (which we'll see soon).

Write-Ahead Logging (WAL)

The Write-Ahead Logging Protocol:

1. Must force the log record for an update *before* the corresponding data page gets to disk.
2. Must force all log records for a Xact *before commit*. (e.g. transaction is not committed until all of its log records including its "commit" record are on the stable log.)

#1 (with UNDO info) helps guarantee Atomicity.

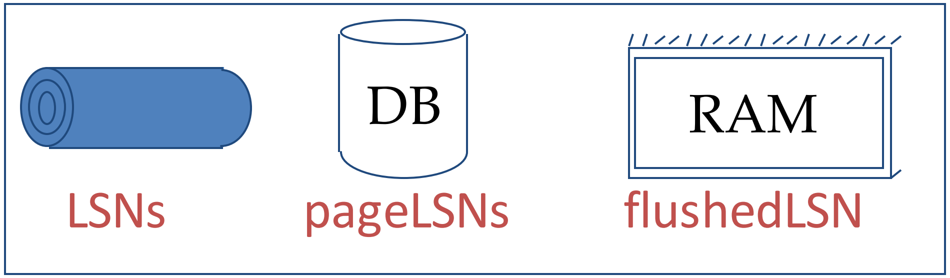
#2 (with REDO info) helps guarantee Durability.

This allows us to implement Steal/No-Force

Exactly how is logging (and recovery!) done?

- We'll look at the ARIES algorithm from IBM.

WAL & the Log



Each log record has an unique Log Sequence Number (LSN).

- LSNs are always increasing.

Each *data page* contains a pageLSN.

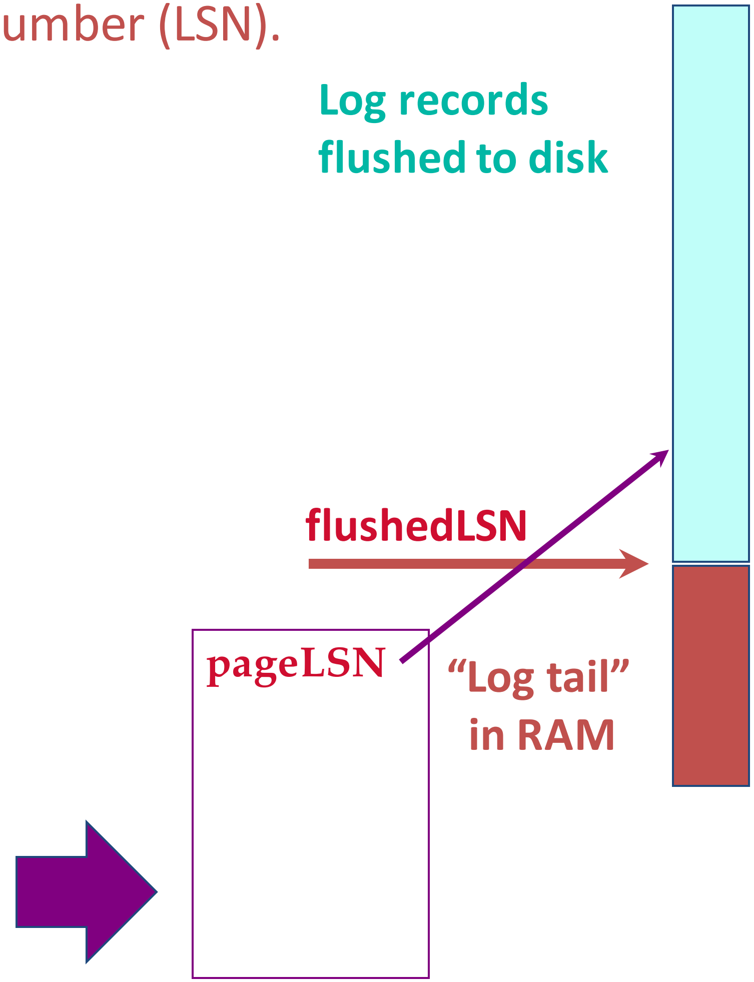
- The LSN of the most recent *log record* for an update to that page.

System keeps track of flushedLSN.

- The max LSN flushed so far.

WAL: For a page i to be written must flush log at least to the point where:

$\text{pageLSN}_i \leq \text{flushedLSN}$



Log Records

LogRecord fields:

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

update records only: pageID, length, offset, before-image, after-image

prevLSN is the LSN of the previous log record written by *this* transaction (so records of an transaction form a linked list backwards in time)

Possible log record types:

Update, Commit, Abort

Checkpoint (for log maintenance)

Compensation Log Records (CLRs)

- for UNDO actions

End (end of commit or abort)

Other Log-Related State

In-memory table:

Transaction Table

- One entry per currently active transactions.
  - entry removed when the transaction commits or aborts
- Contains XID, status (running/committing/aborting), and lastLSN (most recent LSN written by transaction).

Also: Dirty Page Table (will cover later …)

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

LSN of most recent checkpoint

Xact Table

- lastLSN
- status

Dirty Page Table

- recLSN

flushedLSN

Normal Execution of a transaction

Series of reads & writes, followed by commit or abort.

- We will assume that disk write is atomic.
    - In practice, additional details to deal with non-atomic writes.

Strict 2PL.

STEAL, NO-FORCE buffer management, with Write-Ahead Logging.

Transaction Commit

Write commit record to log.

All log records up to transaction's commit record are flushed to disk.

- Guarantees that flushedLSN ≥ lastLSN.
- Note that log flushes are sequential, synchronous writes to disk.
- Many log records per log page.

Commit() returns.

Write end record to log.

Simple Transaction Abort

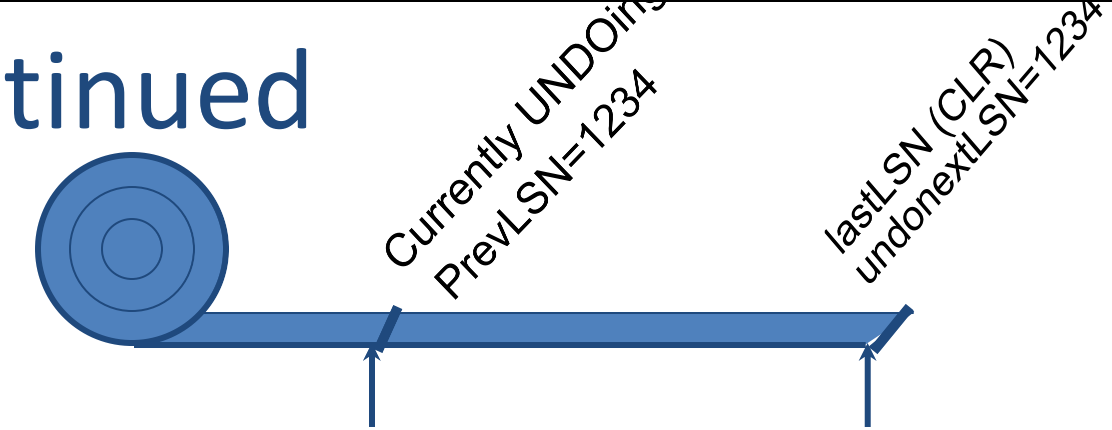
For now, consider an explicit abort of a Xact.

- No crash involved.

We want to “play back” the log in reverse order, UNDOing updates.

- Get lastLSN of Xact from Xact table.
- Can follow chain of log records backward via the prevLSN field.
- Write a “CLR” (compensation log record) for each undone operation.
- Write an *Abort* log record before starting to rollback operations.

Abort, continued



- To perform UNDO, must have a lock on data!
  - No problem (we’re doing Strict 2PL)!
- 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.

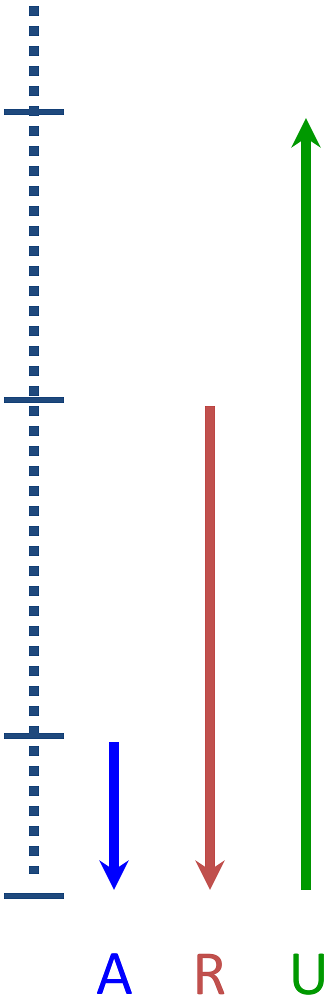
Checkpointing

Conceptually, keep log around for all time. Obviously this has performance/implementation problems…

Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover in the event of a system crash. Write to log:

- begin_checkpoint record: Indicates when chkpt began.
- end_checkpoint record: Contains current *transaction table* and *dirty page table*. This is a 'fuzzy checkpoint':
  - Other Xacts continue to run; so these tables accurate only as of the time of the begin_checkpoint record.
  - No attempt to force dirty pages to disk; effectiveness of checkpoint limited by oldest unwritten change to a dirty page.
- Store LSN of most recent checkpoint record in a safe place (*master* record).

Crash Recovery: Big Picture



Oldest log rec. of Xact active at crash

Smallest recLSN in dirty page table after Analysis

Last chkpt

CRASH

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

Recovery: The Analysis Phase

Re-establish knowledge of state at checkpoint.

- via transaction table and dirty page table stored in the checkpoint

Scan log forward from checkpoint.

- End record: Remove Xact from Xact table.
- All Other records: Add Xact to Xact table, set lastLSN=LSN, change Xact status on commit.
- also, for Update records: If page P not in Dirty Page Table, Add P to DPT, set its recLSN=LSN.

At end of Analysis…

- transaction table says which xacts were active at time of crash.
- DPT says which dirty pages *might not* have made it to disk

Phase 2: 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 DPT.

Q: why start here?

For each update log record or CLR with a given 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. (this last case requires I/O)

To REDO an action:

- Reapply logged action.
- Set pageLSN to LSN. No additional logging, no forcing!

Phase 3: The UNDO Phase

ToUndo={lastLSNs of all Xacts in the Xact Table}

Repeat:

- Choose (and remove) largest LSN among ToUndo.
- If this LSN is a CLR and undonextLSN==NULL

  Write an End record for this transation.
- 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

RAM

Xact Table

lastLSN

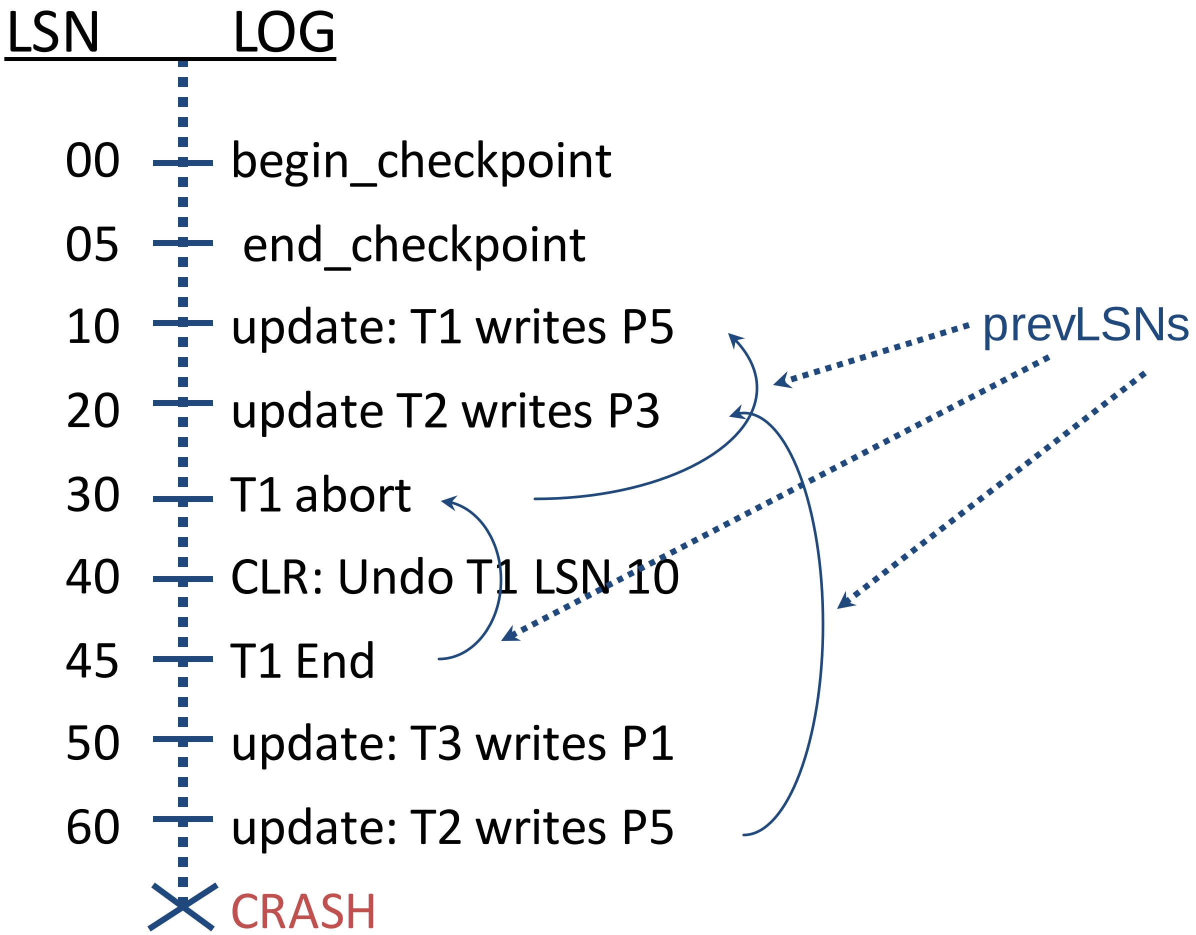
status

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

prevLSNs

Example: Crash During Restart!

Xact Table

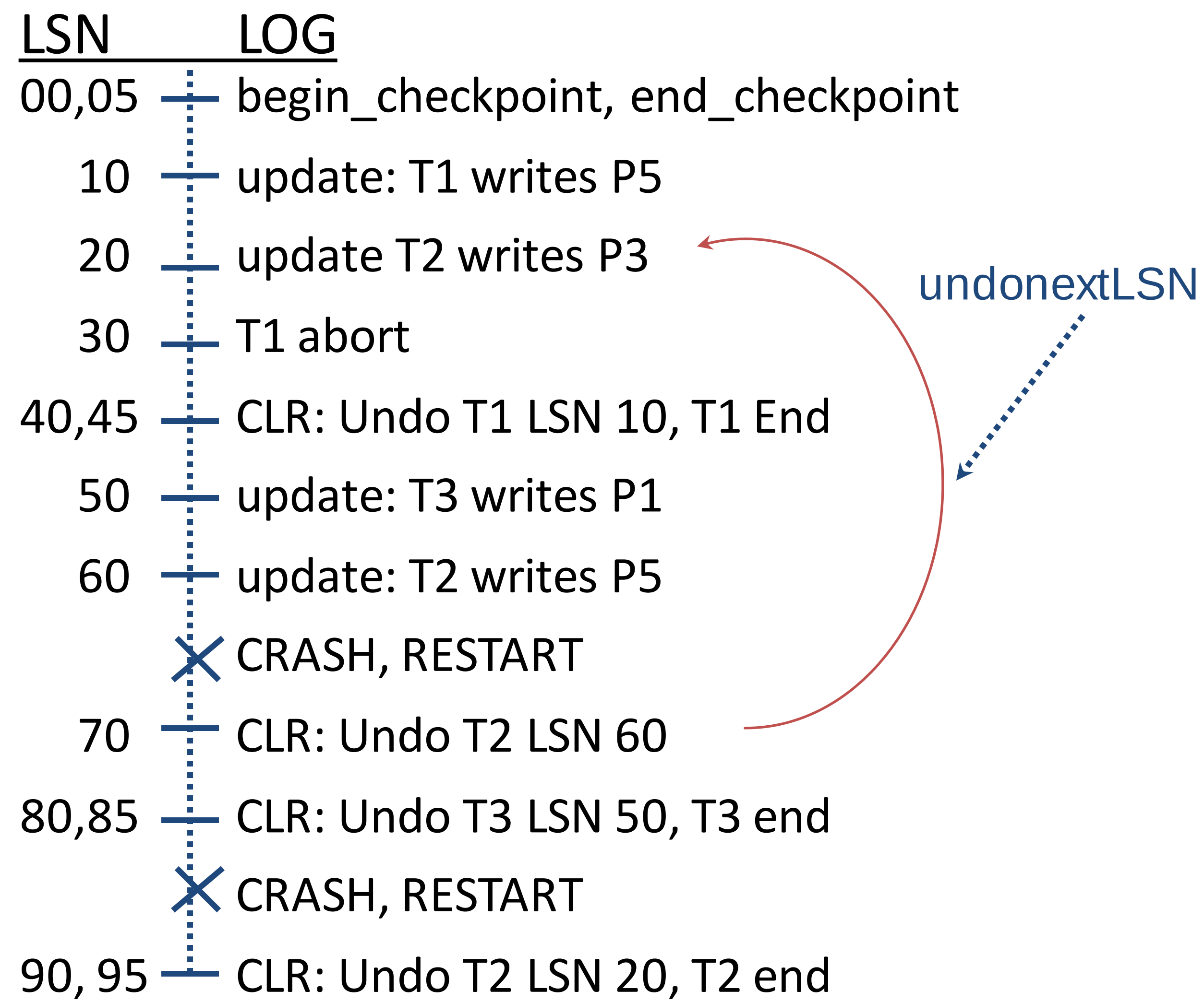
- lastLSN
- status

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, 95	CLR: Undo T2 LSN 20, T2 end

Additional Crash Issues

What happens if system crashes during Analysis? During REDO?

How do you limit the amount of work in REDO?

- Flush asynchronously in the background.

How do you limit the amount of work in UNDO?

- Avoid long-running transactions.

Summary of Logging/Recovery

Recovery Manager guarantees Atomicity & Durability.

Use WAL to allow STEAL/NO-FORCE without sacrificing correctness.

LSNs identify log records; linked into backwards chains per transaction (via prevLSN).

pageLSN allows comparison of data page and log records.

Summary, continued

Checkpointing: A quick way to limit the amount of log to scan on recovery.

Recovery works in 3 phases:

- Analysis: Forward from checkpoint.
- Redo: Forward from oldest recLSN.
- Undo: Backward from end to first LSN of oldest Xact alive at crash.

Upon Undo, write CLRs.

Redo “repeats history”: Simplifies the logic!