Recovery with Aries

Locking

Goal: A protocol that to ensure that any schedule produced using the protocol is serializable.

Lock and **Unlock** take DB resources as arguments:
DB, a relation, tuple, etc.

TX locks X before an action on X is taken, and then unlocks after the action is taken.

Two-phase Locking (2PL): TX locks X before an action on X is taken. Never requests a lock after releasing one or more locks.

High-level comments on Paper

Paper has **incredible** amounts of detail: latches, conditional locking, lock durations, history of shadow paging, etc.

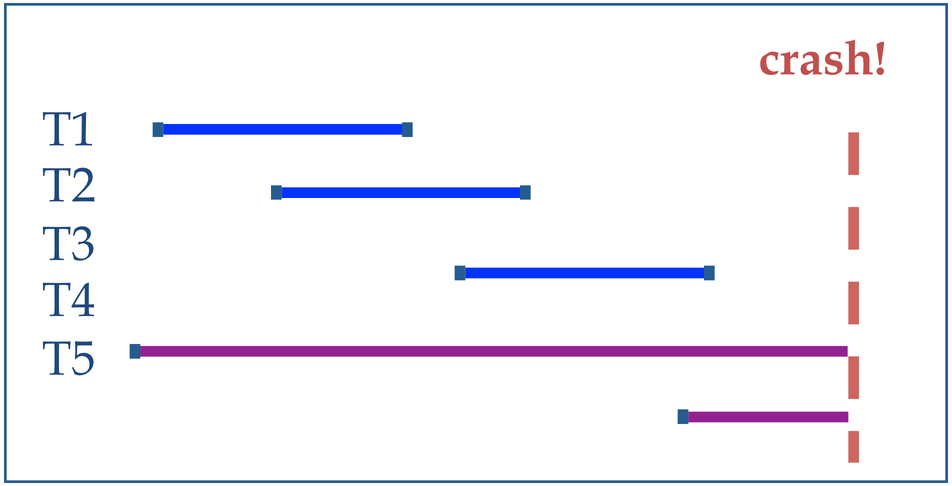
I'm focused on recovery in this lecture…

Motivation

- Atomicity:
  - Transactions may abort ("Rollback").
- Durability:
  - What if DBMS stops running? (Causes?)

❖ Desired Behavior after system restarts:
  - T1, T2 & T3 should be durable.
  - T4 & T5 should be aborted (effects not seen).

We may also want partial rollbacks.



High-level Goals

- **Always** be able to
  1. Back out effects of uncommitted TXs
  2. Recover results of committed TX
  3. Get consistent snapshot of the DB

Achieving the Goal

- Some Concurrency Control Mechanism (locking)
- DO-UNDO-REDO (more later)
- WAL

Review of Locking

Review: The ACID properties

- **A** tomicity: All actions in the Xact happen, or none happen.
- **C** onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one Xact is isolated from that of other Xacts.
- **D** urability: If a Xact commits, its effects persist.

- The **Recovery Manager** guarantees Atomicity & Durability.

Assumptions

- Concurrency control is in effect.
  - Strict 2PL, in particular.
- Updates are happening "in place".
  - i.e. data is overwritten on (deleted from) the disk.

Handling the Buffer Pool

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

	No Steal	Steal
Force	Trivial	
No Force		Desired

More on Steal and Force

<u>STEAL</u> *(why enforcing Atomicity is hard)*

- *To steal frame F:* Current page in F (say P) is written to disk; some Xact holds lock on P.
  - What if the Xact with the lock on P aborts?
  - Must remember the old value of P at steal time (to support UNDOing the write to page P).

<u>NO FORCE</u> *(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.

Write-Ahead Logging (WAL)

The Write-Ahead Logging Protocol:

1. **Must** force log record for an update *before* corresponding data page goes to disk.

2. Must write all log records for a Xact *before commit*.

#1 guarantees Atomicity.

#2 guarantees Durability.

Recovery

Three Critical Recovery Questions

1. What kind of failures do we protect against?

2. What kind of operations on the data do we support?

3. What are the characteristics of our available resources?

1. Types of Failures

- Action Failure: bad parameters
- Transaction Failure: Deadlock, abort, local errors
- System Failure: Hardware Crash, Panic
- Media Failure: Disk is corrupted and destroyed.

Ideally, all of them!

Today, some also worry at the data center level!

2. Operations and Programming Model

- Fine-grained Read and Writes
- Increment/Decrement: why?
- Explicit Rollback/Partial Rollback/Nested Rollback

ARIES Supports this, but we will return to it next time.

3. Resources: Storage Types

- *Volatile Storage* (buffers in main memory)
  - Lost when a crash occurs
- *Non-Volatile Storage*: survives a crash (more reliable than volatile storage)
- *Stable Storage:* “never” fails.
- *Non-Volatile Offline-Storage: Highly reliable stuff (geographically diverse backup, tapes)*

We use different storage types to store data that will guard against a set of failures.

Aries Main Ideas

Recovering From a Crash: Aries

- Main Idea: *Repeat history using the log*. 3 Phases.
  1. **Analysis**: Find the earliest transactions that were active at the time of the crash
  2. **Redo**: Put the DB back into the state at the time of the crash by redoing operations in the log.
  3. **Undo:** Abort those TXs still in flight!

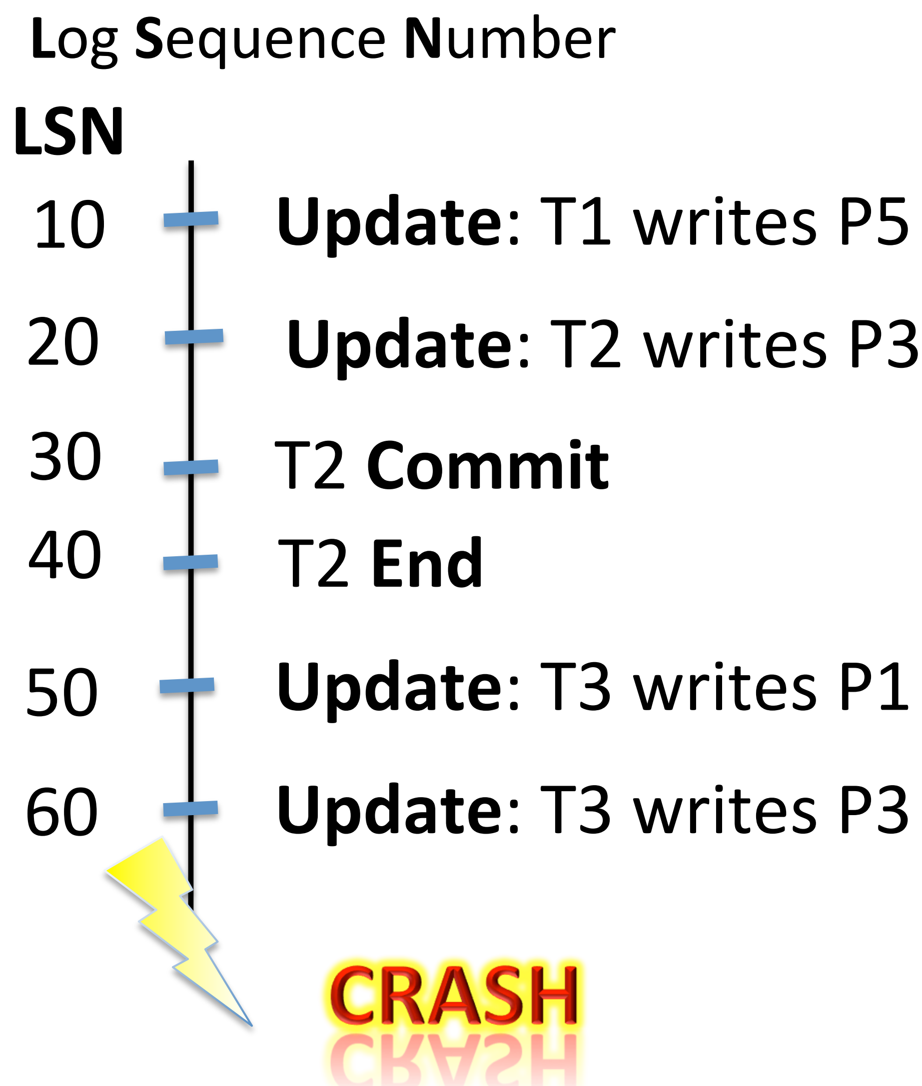
Some more details!

Example Execution History

Analysis: Identify dirty pages in the buffer pool at time of crash and active TXs

Redo: Redo all the writes (even if they didn't go to disk!)

Undo: Which transactions need to be aborted?



Recovering From a Crash

- Main idea in Aries: "Repeating History"
  - *Analysis*: Scan the log forward (from the most recent *checkpoint*) to identify all Xacts 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 Xacts 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!)

Outline for this Section

- The Main Characters: logs, DPTs, Xact tables, Checkpoints

- How does Abort work? Commit?

- The big, awesome, messy recovery

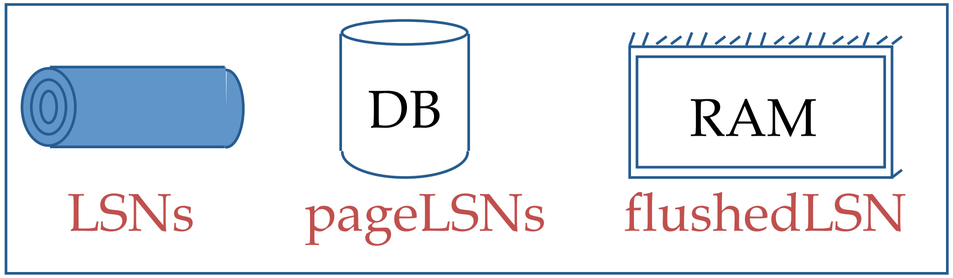
NB: We will start with physical UNDO/REDO.

Basic Idea: Logging

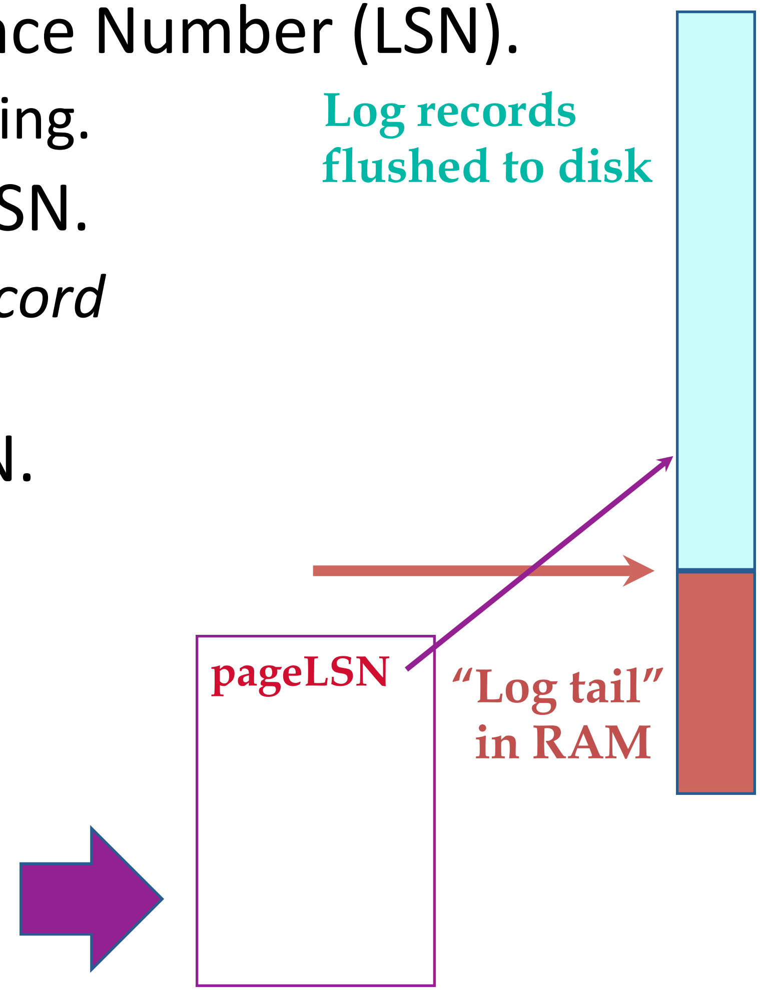
- Record REDO and UNDO for every update, in a *log*.
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.
- <u>Log</u>: An ordered list of REDO/UNDO actions
  - Log record contains:

    <XID, pageID, offset, length, old data, new data>
  - and additional control info (which we'll see soon).

WAL & the Log



- Each log record has a Log Sequence Number (LSN).
  - LSNs is unique and always increasing.
- Each <u>*data page*</u> 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.
- <u>WAL</u>: *Before* a page is written,
  - pageLSN ≤ flushedLSN



Log Records

LogRecord fields:

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

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

Possible log record types:

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

Other Log-Related State

- **Transaction Table:**
  - One entry per active Xact.
  - 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.

This is the first record which may have to be redone

Normal Execution of an Xact

- Series of reads & writes, followed by commit or abort.
  - We will assume that write is atomic on disk.
- Strict 2PL.

Are disk-writes atomic in practice?

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

Xact Table

lastLSN
status

Dirty Page Table

recLSN

flushedLSN

Checkpointing

- Periodically, the DBMS creates a <u>checkpoint</u> to minimize the time taken to recover
- Log for Checkpoint
  - begin_checkpoint record: Indicates when chkpt began.
  - end_checkpoint record: Contains current *Xact 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. (So it's a good idea to periodically flush dirty pages to disk!)
  - Store LSN of chkpt record in a safe place (*master* record).

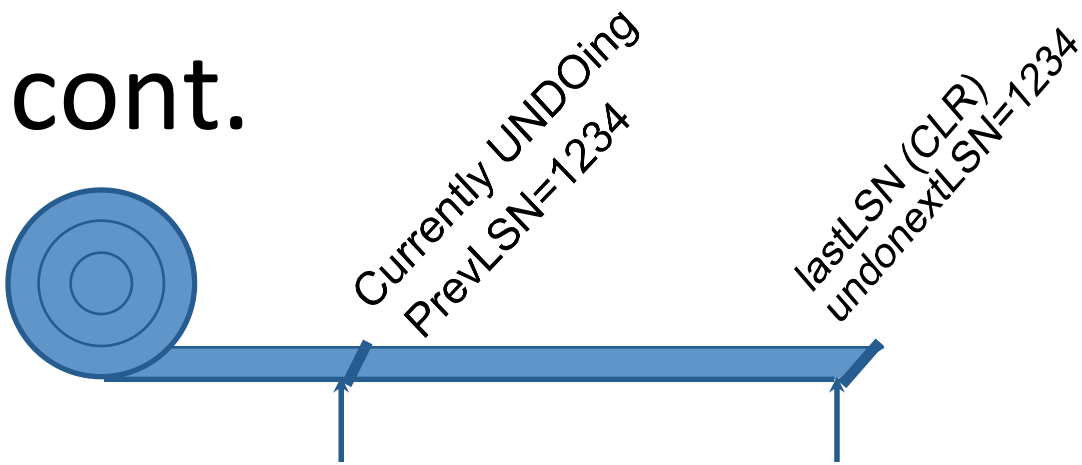
Is this enough to make sure recovery is fast?
(Think: Hot Pages -> lots of log records…)

End of Characters,
Beginning of Abort/Commit

Simple Transaction Abort

- For now, consider an explicit abort of a Xact.
  - No crash involved.
- Idea: "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.
  - 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!
  - No problem! Why?
- 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
    - Possibly REDOne (for atomicity)
- At end of UNDO, write an "end" log record.

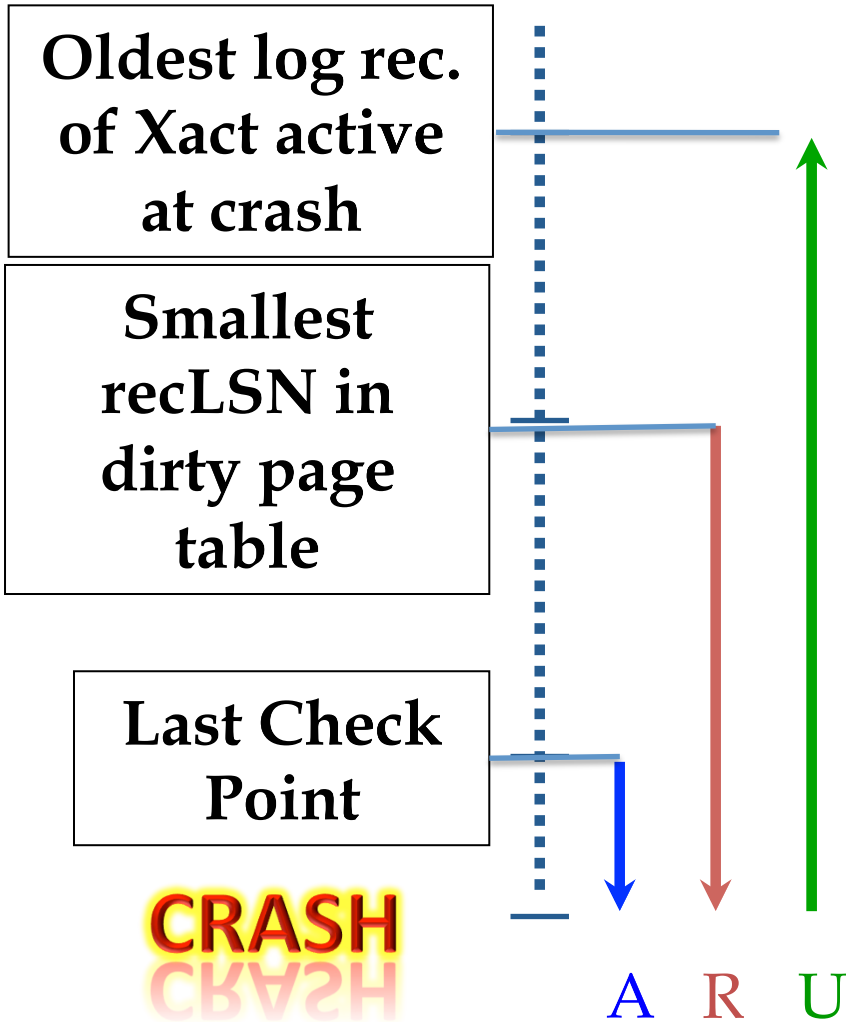
Transaction Commit

- Write commit record to log and force write the log tail
- All log records up to Xact's lastLSN are flushed.
  - 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.

Transaction is committed *once commit record is on stable storage*

Crash Recovery: Aries.

Crash Recovery: Big Picture



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

Recovery: Analysis

Goals:

1. Determine the point in the log from which to start REDO

2. Determine a superset of the pages that are dirty at the time of the crash

Avoids unnecessary IO.

3. Identifies transactions that were "in flight" (losers). Why?

Recovery: The Analysis Phase

- Reconstruct state at checkpoint.
  - via end_checkpoint record.
- Scan log forward from checkpoint.
  - End record: Remove Xact from Xact table.
  - Other records: Add Xact to Xact table, set lastLSN=LSN, change Xact status on commit.
  - REDOable record: If P not in Dirty Page Table,
    - Then, add P to D.P.T., set its recLSN=LSN.

DPT is a superset of all dirty pages.
Where does the slop come from?

Recovery: The REDO Phase

- We *repeat history* to reconstruct state at crash:
  - Reapply *all* updates (even of aborted Xacts!), redo CLRs.
- To REDO an action:
  - Reapply logged action.
  - Set pageLSN to LSN. No additional logging!

If you remember nothing else about Aries:
Remember repeating history.

Recovery: The REDO Phase

- firstLSN = min recLSN in DPT.
- Scan Forward from here.
- For each CLR or update log rec LSN, REDO the action unless:
  1. Affected page is not in the Dirty Page Table, or
  2. Affected page is in D.P.T., but has recLSN > LSN, or
  3. pageLSN (on disk) ≥ LSN.

Which checks require IO?

recLSN in DPT *"First LSN that dirtied this page"*

Recovery: The UNDO Phase

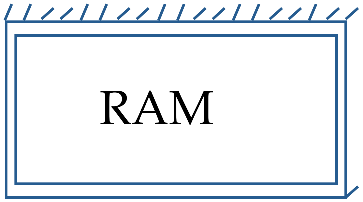
ToUndo={ *l* | *l* a lastLSN of a "loser" Xact}

Repeat:

- Choose largest LSN among ToUndo.
- If this LSN is a CLR and undonextLSN==NULL
  - Write an End record for this Xact.
- 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



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, RESTART

prevLSNs

18.6.3 in Cow book Walks through this example

Example: Crash During Restart!

RAM

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	CLR: Undo T2 LSN 20, T2 end

undonextLSN

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.
  - Watch "hot spots"!
- How do you limit the amount of work in UNDO?
  - Avoid long-running Xacts.

Summary of Logging/Recovery

- Recovery Manager guarantees Atomicity & Durability.
- Use WAL to allow STEAL/NO-FORCE w/o sacrificing correctness.
- LSNs identify log records; linked into backwards chains per transaction (via prevLSN).
- pageLSN allows comparison of data page and log records.

Summary, Cont.

- 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!