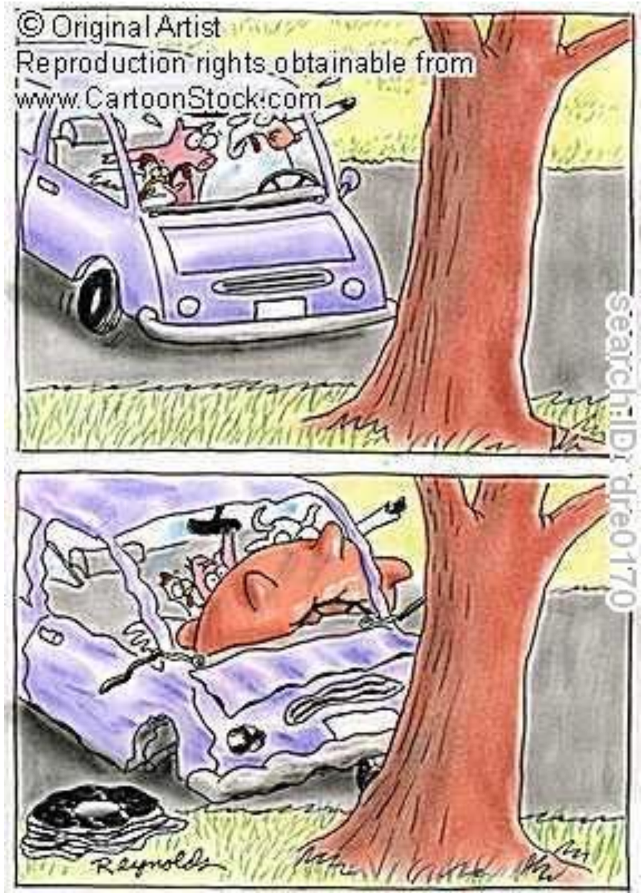
Database Crash Recovery



PS #3 graded
Move PS #4 due on 11/7

Review: The ACID properties

- **Atomicity:** All actions of a transaction happen, or none happen.
- **Consistency:** If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **Isolation:** Execution of one Xact is isolated from that of other Xacts.
- **Durability:** If a Xact commits, its effects persist.

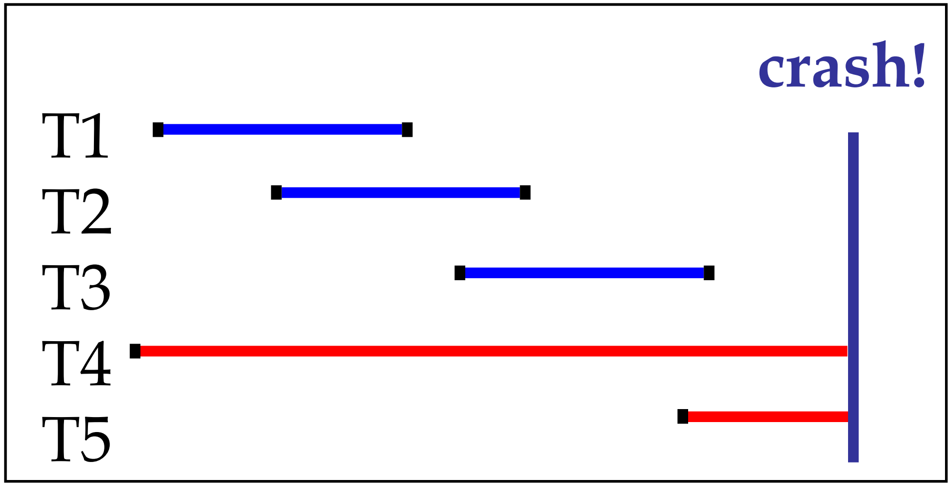
- The **Recovery Manager** guarantees Atomicity & Durability.

Motivation

- Atomicity:
  - Transactions may abort ("Rollback").
- Durability:
  - What if DBMS Crashes?
    ("Worse case", a few unfinished Xacts are lost)

Desired state after system restarts?

- T1, T2 & T3 should be durable.
- T4 & T5 should be aborted (no effect).



Assumptions

- Concurrency control is in effect.
  - In particular, locks are acquired on blocks before reading or writing and are released after commit.
- Updates are happening "in place".
  - i.e. data is overwritten on (or deleted from) non-volatile disk.
  - "In place" implies, we are not using a temporary/in memory database or cache, but one that is persistent.
- Can you think of a simple scheme to guarantee Atomicity & Durability?

Recalling the Buffer Pool

Which of the following types of pages might be found in the buffer pool?

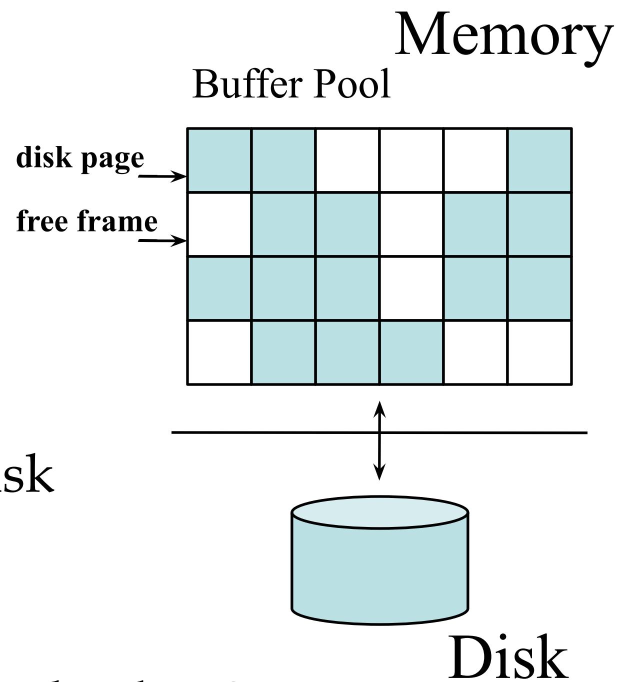
A) Interior steering nodes of a B^+-tree index

B) Intermediate sorted pages from a recent sort-merge-join

C) A bucket of <key, rid> pairs from a hash index

D) A “dirty” updated page from a relation that has yet to be commited to disk

E) All of the above



Of these, which must be tracked in by the log?

Handling the Buffer Pool

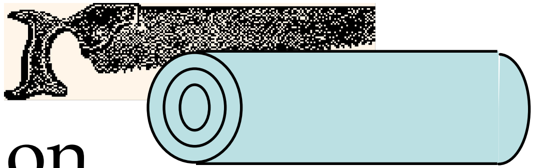
- Force every write to disk? Stall DBMS until completed
  - Poor response time.
  - But provides durability.
- Steal buffer-pool frames from uncommitted Xacts? (flush dirty frames, only when a new frame is needed)
  - If not, poor throughput (multiple writes to same page).
  - If so, how can we ensure atomicity?

	No Steal	Steal
Force	Trivial	
No Force		Desired

More on Steal and Force

- **STEAL** (why enforcing Atomicity is hard)
  - What if a page, P, dirtied by some unfinished Xact is written to disk to free up a buffer slot, and the Xact later aborts?
    - Must remember the old value of P at steal time (to ***UNDO*** the page write).
- **NO FORCE** (why enforcing Durability is hard)
  - What if system crashes before a page dirtied by a committed Xact is flushed to disk?
    - Write as little as possible, in a convenient place, at commit time, to support ***REDO***ing modifications.

Basic Idea: Logging



- Record sufficient information to REDO and UNDO every change in a *log*.
  - Write and Commit sequences saved to log (on a separate disk or replicated on multiple disks).
  - 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:

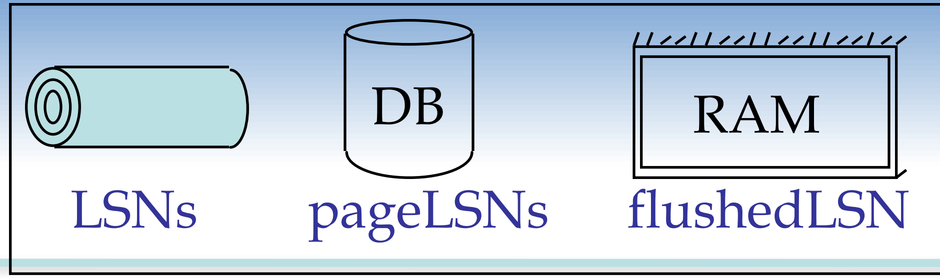
    <XID, pageID, offset, length, old data, new data>
  - and additional control info (which we'll see soon).

Write-Ahead Logging (WAL)

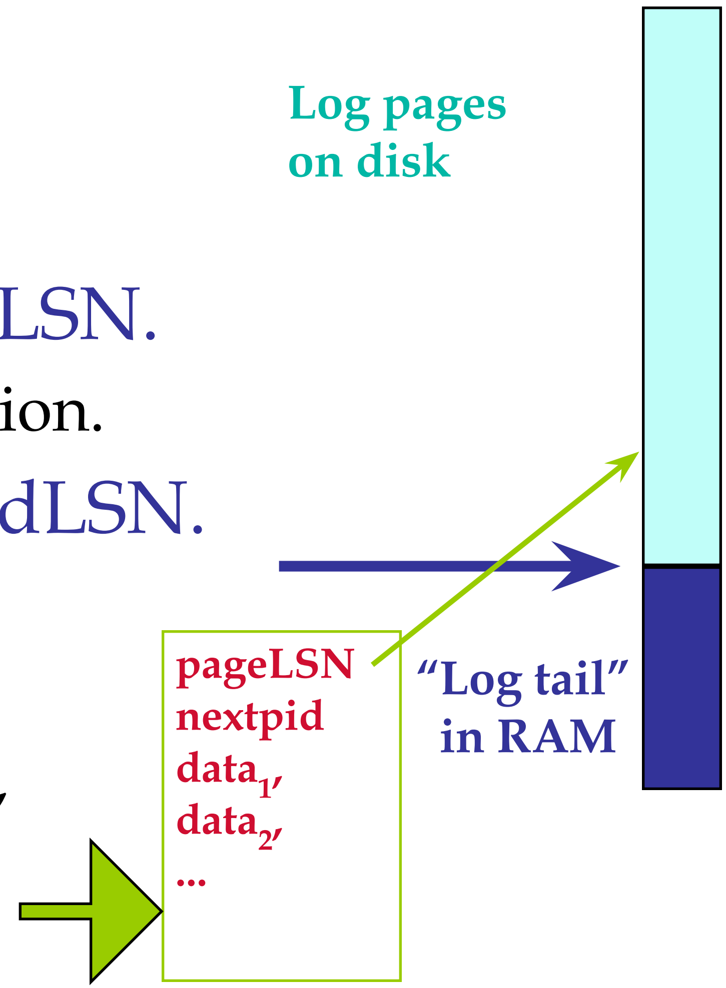
Key Idea of WAL: *Before writing any page to disk, every update log that describes any previous change to this page must be forced to stable storage.*

- The Write-Ahead Logging Protocol:
  1. Modifications of database objects must ***first*** be recorded in the log, and the log updated, ***before*** any change to the actual object
  2. Must write all log records of a Xact *before it commits*.
- #1 guarantees Atomicity.
- #2 guarantees Durability.
- Exactly how is logging (and recovery!) done?
  - We'll study the ARIES algorithm.

WAL & the Log



- Each log record has a unique Log Sequence Number (LSN).
  - LSNs are always increasing.
- Each *data page* contains a pageLSN.
  - LSN of its most recent modification.
- System keeps track of a flushedLSN.
  - Max LSN flushed from the page buffer so far.
- WAL: *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/commited/aborted), and lastLSN due to Xact
- **Dirty Page Table:**
  - One entry per dirty page in buffer pool
  - Contains recLSN -- the LSN of the log record which ***first*** dirtied the page

Log and Table Entries

pageID	recLSN
500	
600	
505	

Dirty Page Table

prevLSN	XID	type	pageID	length	offset	before	after
	T1000	update	500	1	1800	B	Z
	T2000	update	600	3	42	DEF	GHI
	T2000	update	500	2	1799	AZ	MN
	T1000	update	505	1	128	Q	R

Log's "Tail"

transID	status	lastLSN
T1000	running	
T2000	running	

Transaction Table

Normal Execution of an Xact

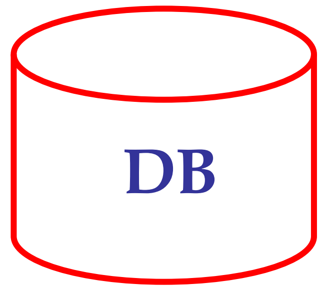
- Series of reads & writes, terminated by commit or abort.
  - We will assume that write is atomic on disk.
    - In practice, additional details to deal with non-atomic writes.
- Strict 2PL.
- STEAL, NO-FORCE buffer management, with Write-Ahead Logging.

Checkpointing

- Periodically, the DBMS creates a checkpoint, to minimize recovery time in the event of a system crash. What is written to log and disk:
  - begin_checkpoint record: Indicates when chkpt began.
  - end_checkpoint record: Contains current active *Xact table* and *dirty page table*. This is a "fuzzy checkpoint":
    - Xacts continue to run; so these tables are 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).

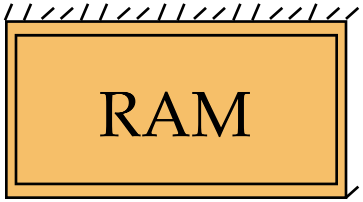
The Big Picture: What's Stored Where

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



Data pages
- each with a pageLSN

master record



Xact Table
- lastLSN
- status

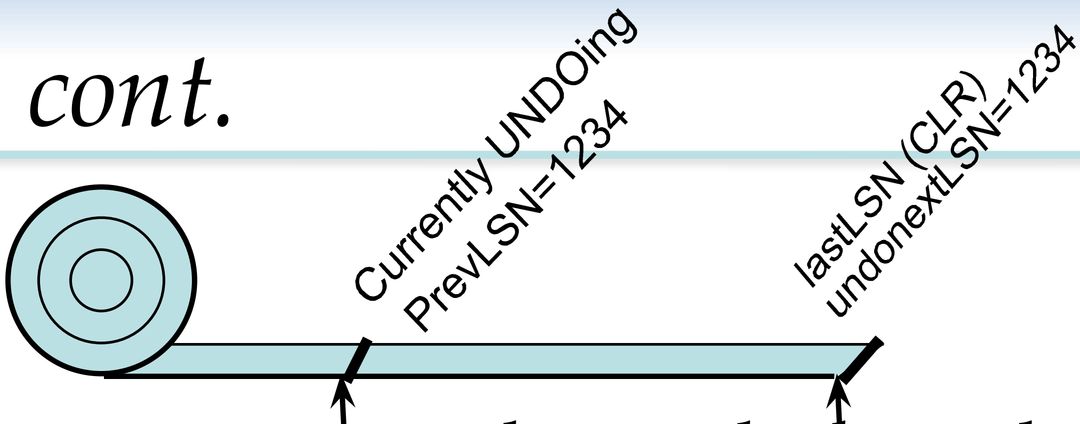
Dirty Page Table
- recLSN

flushedLSN

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.
  - Before starting UNDO, write an *Abort* log record.
    - For recovering from crash during UNDO!

Abort, cont.

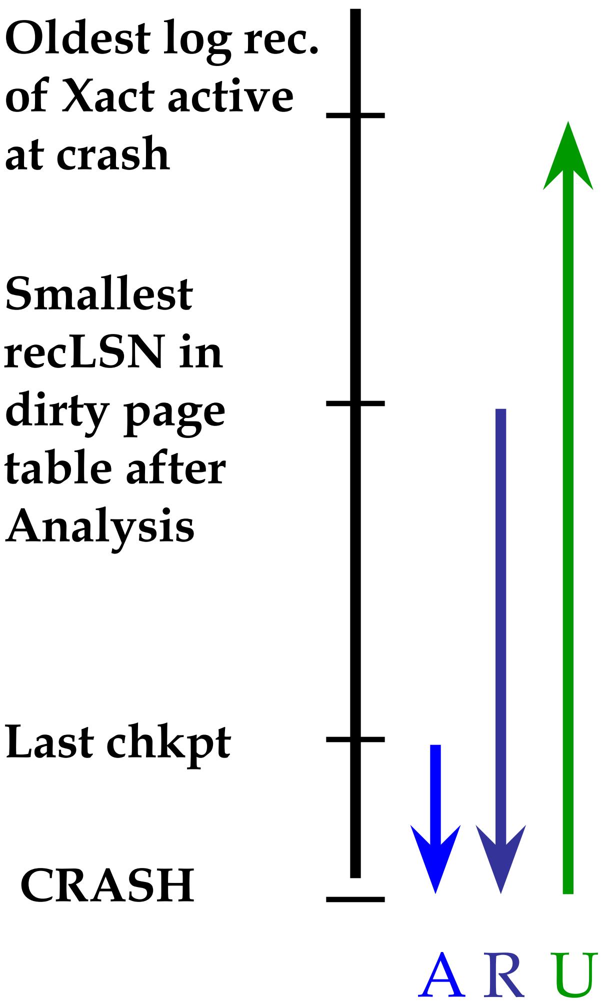


- To perform UNDO, must have a lock on data!
- Before restoring old value of a page, write a Compensation Log Record (CLR):
  - Continue logging while you UNDO!!
  - CLR has one extra field: undonextLSN
    - Points to the next LSN to undo (prevLSN of log entry)
  - CLRs are *never* Undone (but they 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 Xact's lastLSN are flushed on a commit.
  - 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.

Crash Recovery: Big Picture



- Start from a checkpoint (found via master record).
- ARIES 3 phases. Need to:
  - Analysis: Figure out which Xacts committed since last checkpoint, and which did not finish.
  - REDO ***all*** logged actions.

    Repeats "writing" history to recreate buffer pool
  - UNDO effects of unfinished "loser" Xacts.

Recovery: The Analysis Phase

- Reconstruct state at checkpoint.
  - via the end_checkpoint record.
- Scan log forward from checkpoint.
  - Look for End records: Remove Xact from Xact table because it safely completed.
  - Other records: Add Xact to Xact table, set lastLSN=LSN, change Xact 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 Xacts!), redo CLRs.
- Scan forward from log record of the smallest recLSN in the dirty page table. 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 changes (restore to before state).
  - Set pageLSN to LSN. No additional logging!

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

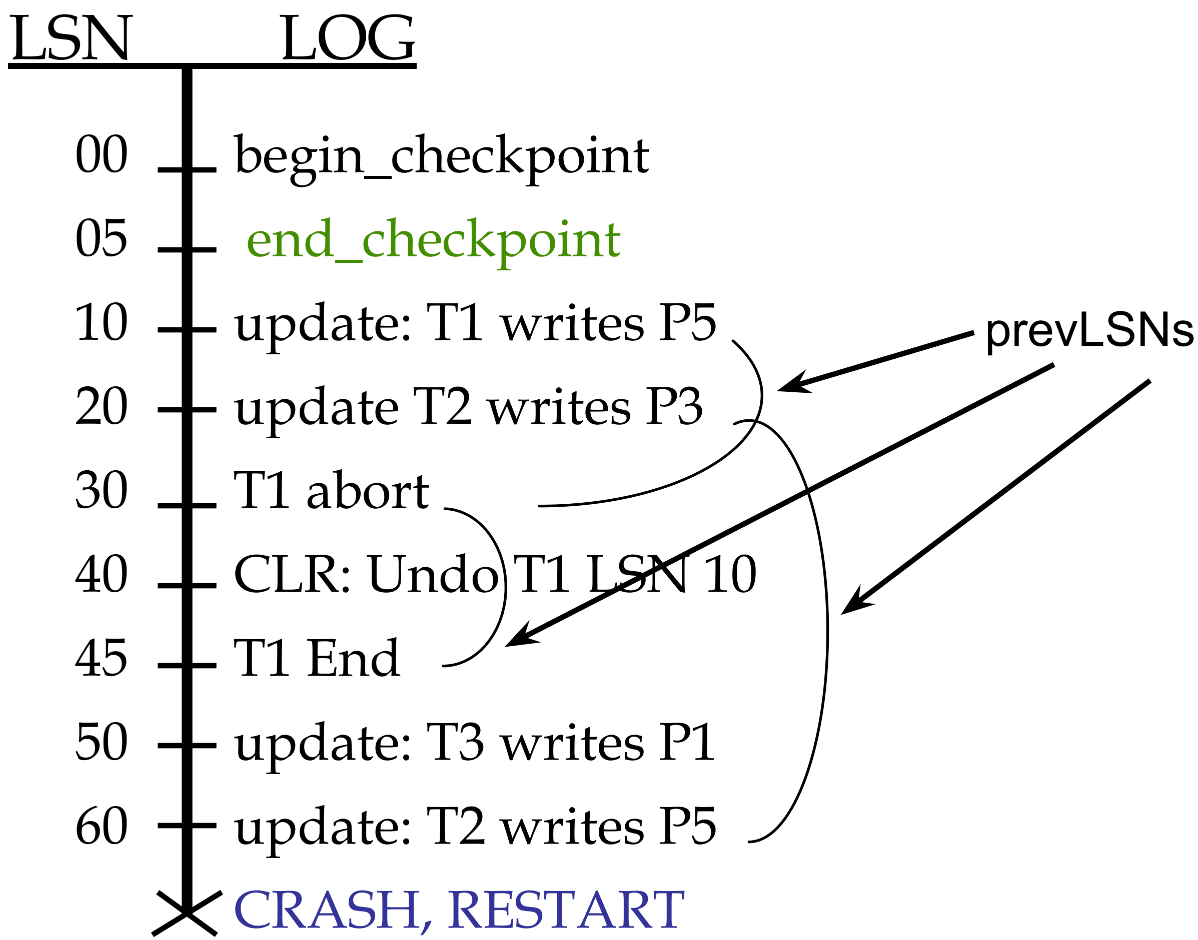
RAM

Xact Table
- lastLSN
- status

Dirty Page Table
- recLSN

flushedLSN

ToUndo



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 to limit the amount of work in REDO?
  - Flush dirty pages asynchronously in the background.
  - Watch out for "hot spots"!
- How to limit the amount of work in UNDO?
  - Avoid long-running Xacts.

Summary of Logging/Recovery

- Recovery Manager guarantees Atomicity & Durability.
- Uses 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!