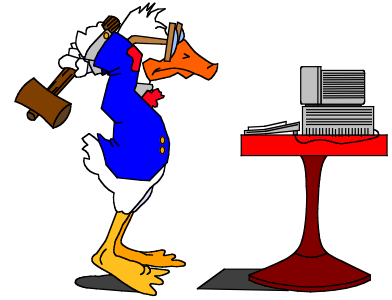


# Crash Recovery



**CSC 375 Fall 2019**  
**R&G - Chapter 18**

If you are going to be in the logging business, one of the things that you have to do is to learn about heavy equipment.

Robert VanNatta,  
*Logging History of  
Columbia County*

## Review: The ACID properties

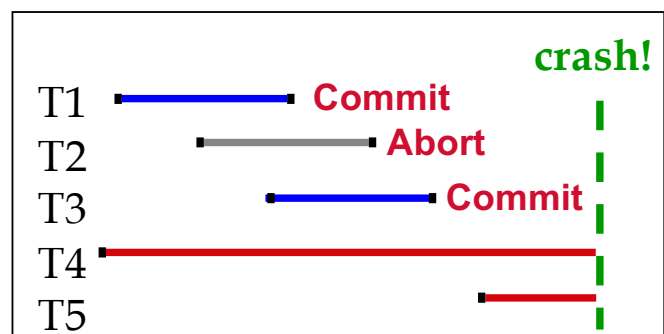
- **Atomicity:** All actions in the Xact happen, or none happen.
  - **Consistency:** If each Xact is consistent, and the DB starts consistent, it ends up consistent.
  - **Isolation:** Execution of one Xact is isolated from that of other Xacts.
  - **Durability:** If a Xact commits, its effects persist.
- 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 not seen).



## Big Ideas

- **Write Ahead Logging (WAL)**
  - and how it interacts with the buffer manager
- **ARIES Recovery algorithm**
  - “Repeats History” in order to simplify the logic of recovery.
  - Must handle arbitrary failures
    - Even during recovery!

# 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

### One possible approach – Force/No Steal:

- **Force** – make sure that every updated page is written to disk before commit.
  - Provides durability without REDO logging.
  - But, can cause poor performance.
- **No Steal** – don't allow buffer-pool frames with uncommitted updates to overwrite committed data on disk.
  - Useful for ensuring atomicity without UNDO logging.
  - But can cause poor performance.

## Preferred Policy: Steal/No-Force

- This combination is most complicated but allows for highest flexibility/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 Xact that performed updates aborts?
  - What if system crashes before Xact is finished?
  - Must remember the old value of P (to support UNDOing the write to page P).

## Buffer Management summary

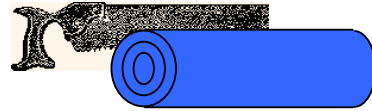
	No Steal	Steal
No Force		<b>Fastest</b>
Force	<b>Slowest</b>	

Performance  
Implications

	No Steal	Steal
No Force	<b>No UNDO REDO</b>	<b>UNDO REDO</b>
Force	<b>No UNDO No REDO</b>	<b>UNDO No REDO</b>

Logging/Recovery  
Implications

# Basic Idea: Logging

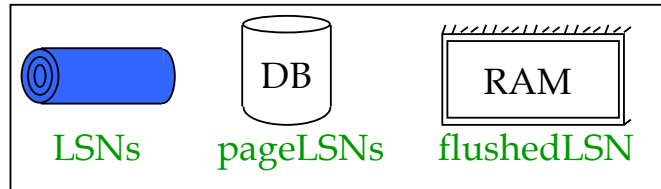


- **Record REDO and UNDO information, for every update, in a *log*.**
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.
- **Log: An ordered list of REDO/UNDO actions**
  - Log record contains:  
`<XID, pageID, offset, length, old data, new data>`
  - and additional control info (which we'll see soon).

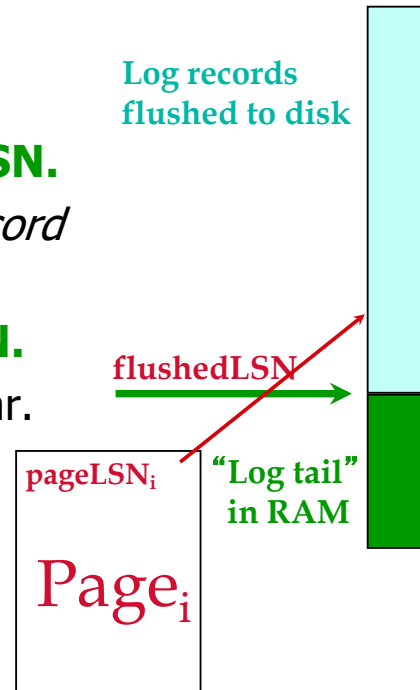
## Write-Ahead Logging (WAL)

- **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*.  
(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**
- **We'll look at the ARIES algorithms from IBM.**

# 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**.
  - max LSN flushed to stable log so far.
- **WAL (rule 1):** 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

for update records only

prevLSN is the LSN of the previous log record written by ***this transaction*** (i.e., the records of an Xact 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)

- **Two in-memory tables:**

- **Transaction Table**

One entry per currently active transaction.

- entry removed when Xact commits or aborts

Contains: **XID** (i.e., transactionId),  
**status** (running/committing/aborting),  
**lastLSN** (most recent LSN written by Xact)

- **Dirty Page Table**

One entry per dirty page currently in buffer pool.

Contains **recLSN** -- the LSN of the log record that ***first*** caused the page to be dirty.

## Normal Execution of an Xact

- **Assume:**
  - **Strict 2PL** concurrency control
  - STEAL, NO-FORCE buffer management, with **WAL**.
  - Disk writes are atomic (i.e., all-or-nothing)
- **Transaction is a series of reads & writes, followed by commit or abort.**
  - Update TransTable on transaction start/end
  - For each update operation:
    - create log record with LSN  $\ell = ++\text{MaxLSN}$  and  
 $\text{prevLSN} = \text{TransTable}[\text{XID}].\text{lastLSN}$ ;
    - update  $\text{TransTable}[\text{XID}].\text{lastLSN} = \ell$
    - if modified page NOT in DirtyPageTable,  
then add it with  $\text{recLSN} = \ell$
  - When buffer manager replaces a dirty page, remove its entry from the DPT

## Transaction Commit

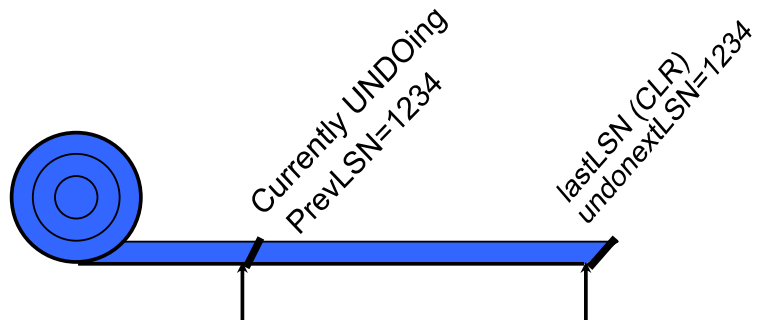
- Write **commit** record into log.
- Flush all log records up to Xact's **commit record** to log disk.
  - WAL Rule #2: Ensure **flushedLSN**  $\geq$  **lastLSN**.
    - Force log out up to lastLSN if necessary
  - Note that log flushes are sequential, synchronous writes to disk and many log records per log page.
    - so, cheaper than forcing out the updated data and index pages.
- **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.
  - Write an *Abort* log record before starting to rollback operations.
  - Get **lastLSN** of Xact from Transaction table.
  - Can follow chain of log records backward via the **prevLSN** field.
  - For each update encountered:
    - Write a “**CLR**” (compensation log record) for each undone operation.
    - Undo the operation (using before image from log record).

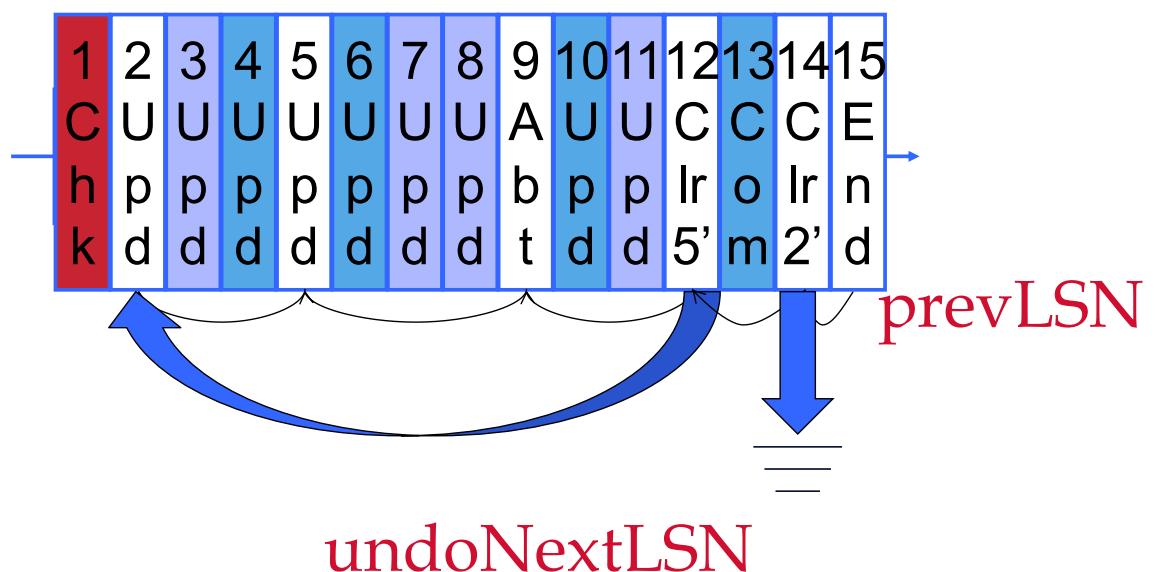


## Abort, cont.



- **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 are *never* Undone (but they might be Redone when repeating history: guarantees Atomicity!)
- **At end of UNDO, write an "end" log record.**

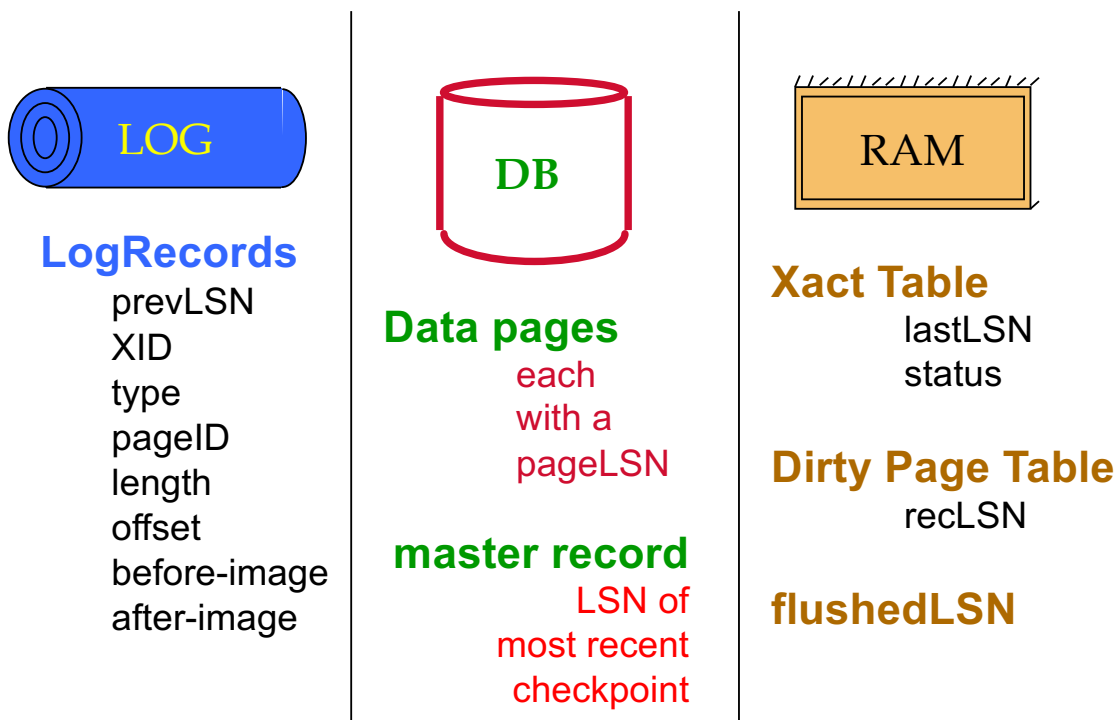
## Abort Example (no crash)



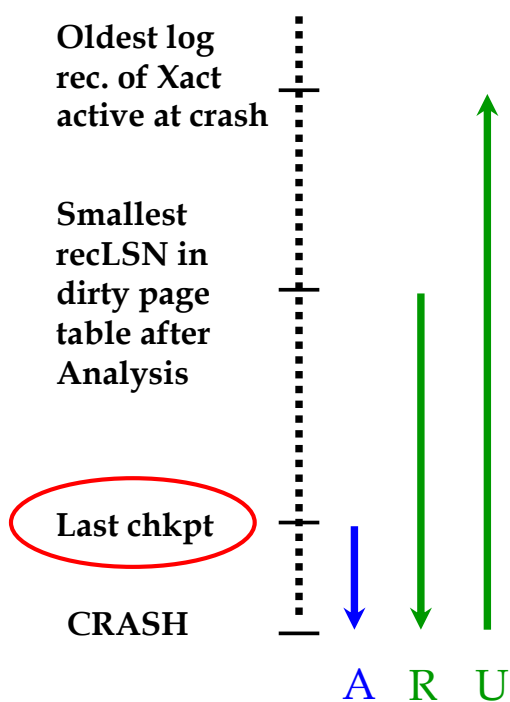
# Checkpointing

- **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 *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.
  - Store LSN of most recent chkpt record in a safe place (**master** record).

## The Big Picture: What's Stored Where



# Crash Recovery: Big Picture



- ❖ Start from a **checkpoint** (found via **master** record).
- ❖ Three phases. Need to:
  - 1. Analysis** - update structures:
    - **Trans Table**: which Xacts were active at time of crash.
    - **Dirty Page Table**: which pages *might* have been dirty in the buffer pool at time of crash.
  - 2. REDO** *all* actions. (repeat history)
  - 3. UNDO** effects of failed Xacts.

## 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 Xacts!), redo CLR's.
- 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)  $\geq$  **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

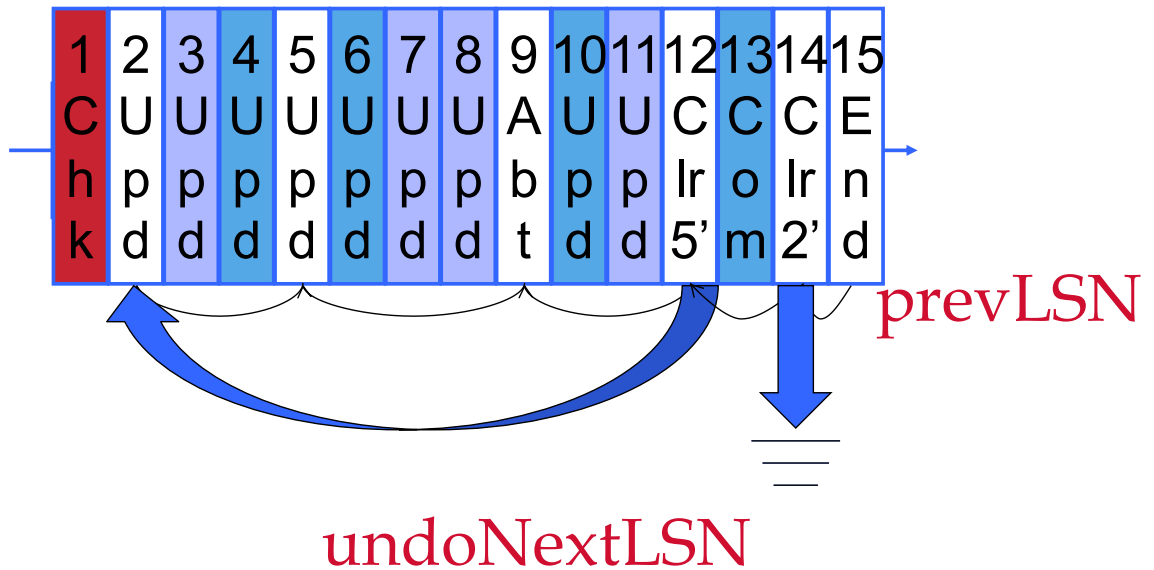
**ToUndo**={lastLSNs of all Xacts in the Trans Table}

Repeat:

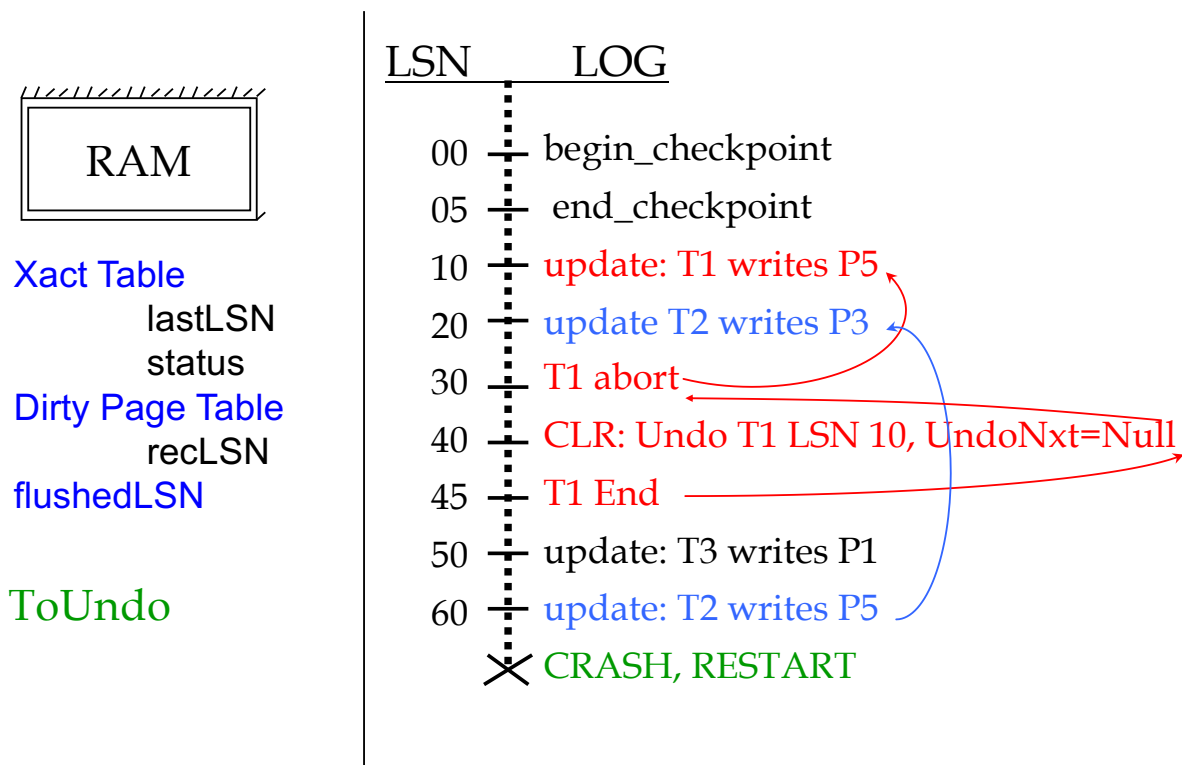
- Choose (and remove) largest LSN among ToUndo.
- If this LSN is a **CLR** and **undonextLSN==NULL**
  - Write an **End** record for this Xact.
- If this LSN is a **CLR**, and **undonextLSN != NULL**
  - Add **undonextLSN** to **ToUndo**
- Else this LSN is an **update**. Undo the update, write a CLR, add **prevLSN** to **ToUndo**.

Until **ToUndo** is empty.

# Abort Example (after crash)



# Example of Recovery – (up to crash)



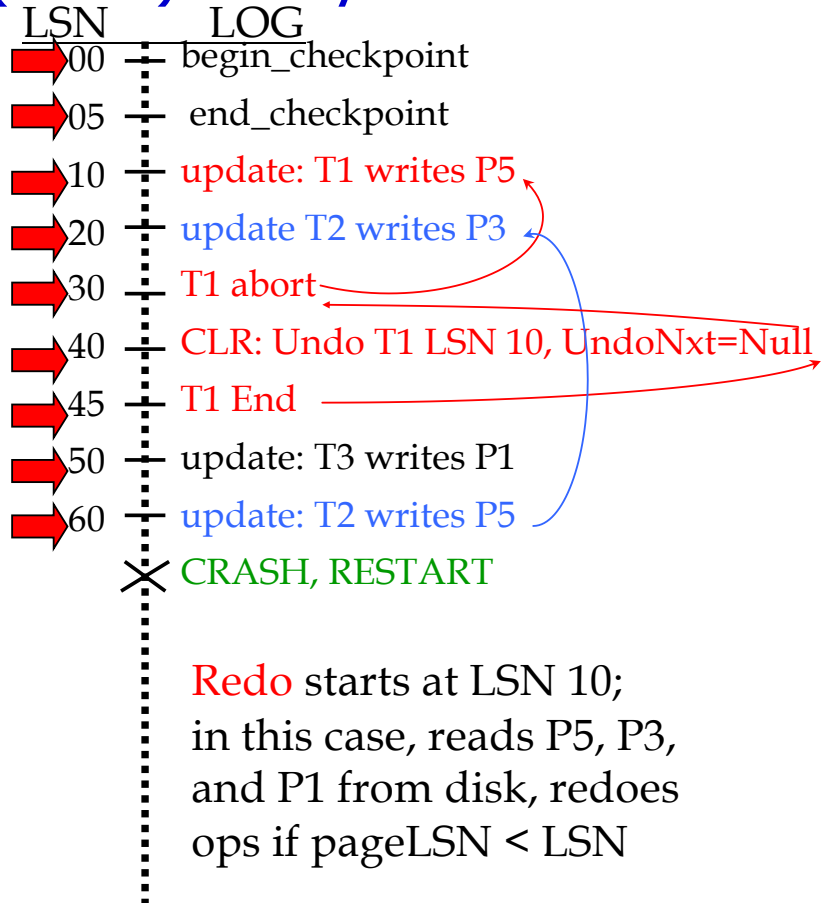
# Example (cont.): Analysis & Redo

Xact Table

Trans	lastLSN	Stat
T2	60	r
T3	50	r

Dirty Page Table

PageId	recLSN
P5	10
P3	20
P1	50



# Ex (cont.): Undo & Crash During Restart!

After Analysis/Redo:

ToUndo: 50 & 60

ToUndo:

50 & 20

ToUndo:

20

After Analysis/Redo:

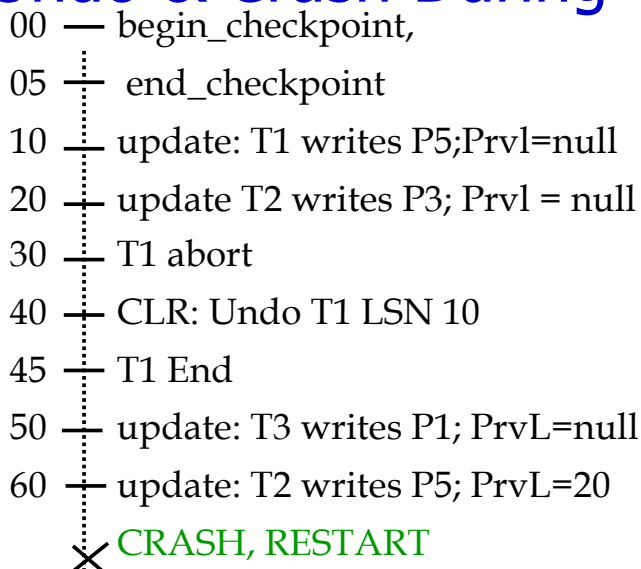
ToUndo: 70

ToUndo:

20

ToUndo:

Finished!



## Additional Crash Issues

- **What happens if system crashes during Analysis? During REDO?**
- **How to reduce the amount of work in Analysis?**
  - Take frequent checkpoints.
- **How do you limit the amount of work in REDO?**
  - Frequent checkpoints plus
  - Flush data pages to disk asynchronously in the background (during normal operation and recovery).
    - Buffer manager can do this to unpinned, dirty pages.
- **How do you limit the amount of work in UNDO?**
  - Avoid long-running Xacts.

## Summary of Logging/Recovery

- **Transactions support the ACID properties.**
- **Recovery Manager guarantees Atomicity & Durability.**
- Use **Write Ahead Logging (WAL)** to allow STEAL/NO-FORCE buffer manager 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, Cont.

- **Checkpointing:** A quick way to limit the amount of log to scan on recovery.
- **Aries recovery works in 3 phases:**
  - **Analysis:** Forward from checkpoint. Rebuild transaction and dirty page tables.
  - **Redo:** Forward from oldest recLSN, repeating history for **all** transactions.
  - **Undo:** Backward from end to first LSN of oldest Xact alive at crash. Rollback all transactions not completed as of the time of the crash.
- **Redo “repeats history”:** Simplifies the logic!
- **Upon Undo, write CLR. Nesting structure of CLR avoids having to “undo undo operations”.**