# Crash Recovery

Chapter 18

Sayyed Nezhadi

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.

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).
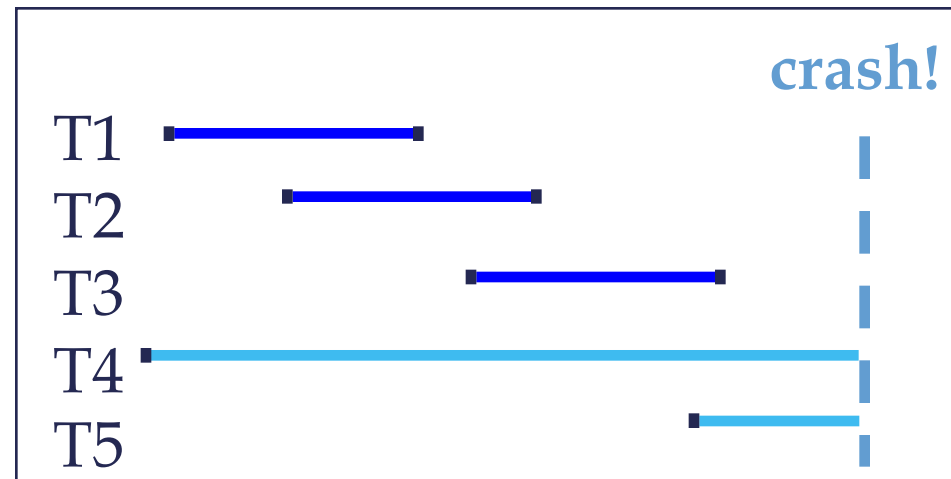
Assumptions

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

- A simple scheme to guarantee Atomicity & Durability?

Handling the Buffer Pool

- Force every write to disk on commit?
  - 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

- **STEAL** (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).

- **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.

Basic Idea: Logging
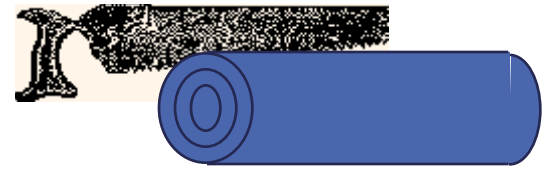
- Record REDO and UNDO information, for every update, in a *log*.
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.

- Log: An ordered list of REDO/UNDO actions
  - Log record contains:

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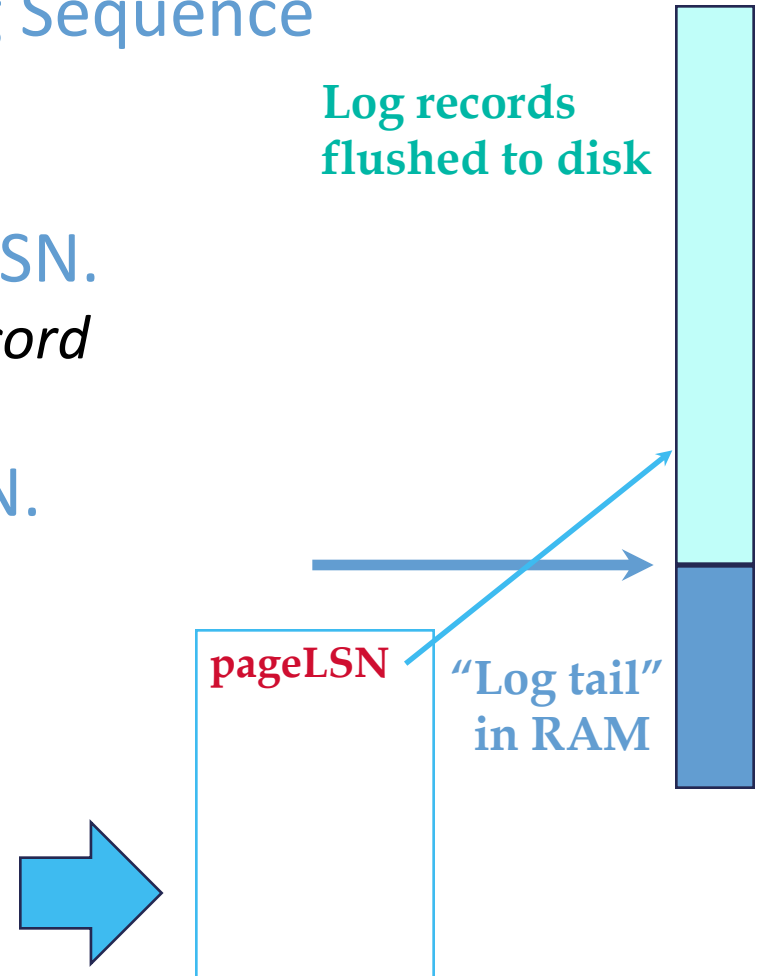 - and additional control info (which we'll see soon).

Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  - Must force the log record for an update *before* the corresponding data page gets to disk.
  - Must write all log records for a Xact *before commit*.
- #1 guarantees Atomicity.
- #2 guarantees Durability.

- Exactly how is logging (and recovery!) done?
  - We'll study the ARIES algorithms.

WAL & the Log

LSNs — pageLSNs — flushedLSN
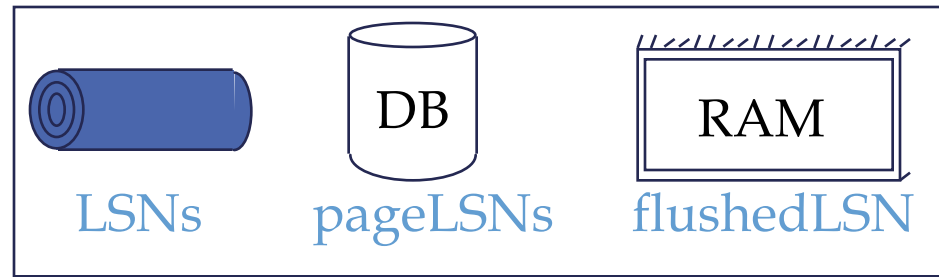
- Each log record has a unique Log Sequence Number (LSN).
  - LSNs always increasing.

- 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,
  - $pageLSN \le flushedLSN$

Log records flushed to disk

pageLSN

"Log tail" in RAM

Log Records

LogRecord fields:

prevLSN
XID
type

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.
- 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 an Xact

- 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.

Checkpointing

- 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. (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

LOG

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

DB

Data pages
  each
  with a
  pageLSN

master record

RAM

Xact Table
 XID
  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.

Currently UNDOing
PrevLSN=1234

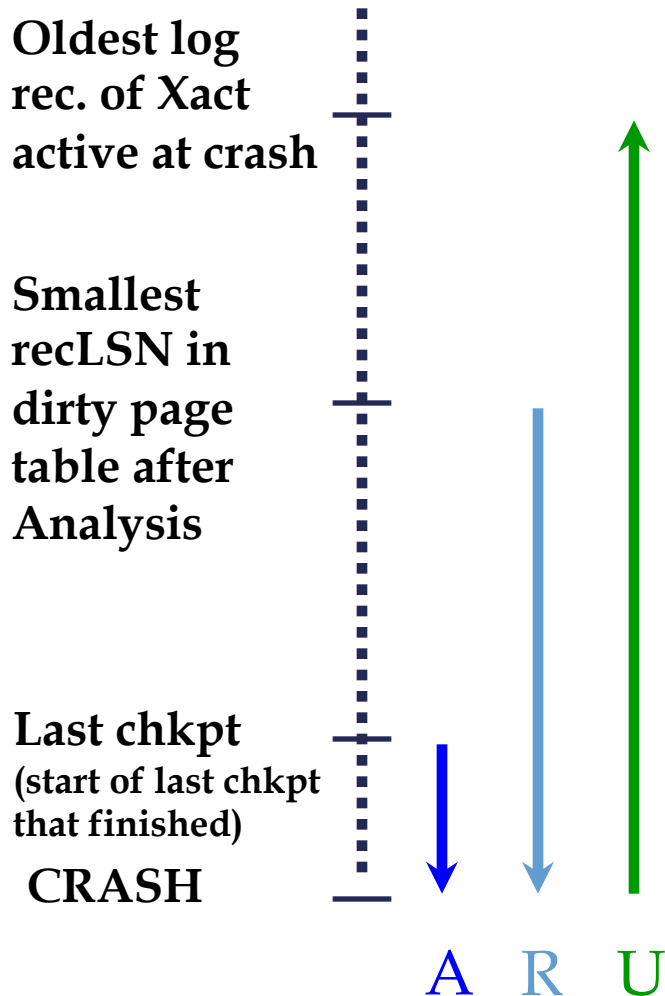lastLSN (CLR)
undonextLSN=1234
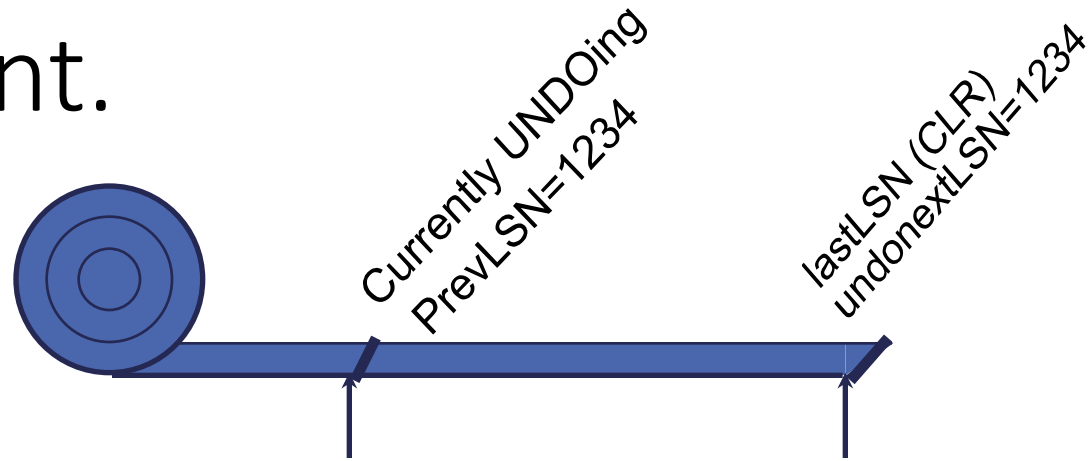
- To perform UNDO, must have a lock on data!
  - No problem!
- Before restoring old value of a page, write a CLR:
  - You continue logging while you UNDO!!
  - CLR has one extra field: undonextLSN
    - Points to the next LSN to undo (i.e. the prevLSN of the record we're currently undoing).
  - CLRs *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 Xact'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.

Crash Recovery: Big Picture

Oldest log rec. of Xact active at crash

Smallest recLSN in dirty page table after Analysis

Last chkpt
(start of last chkpt that finished)

CRASH

A R U

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

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.
  - 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 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) $\geq$ 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" 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.
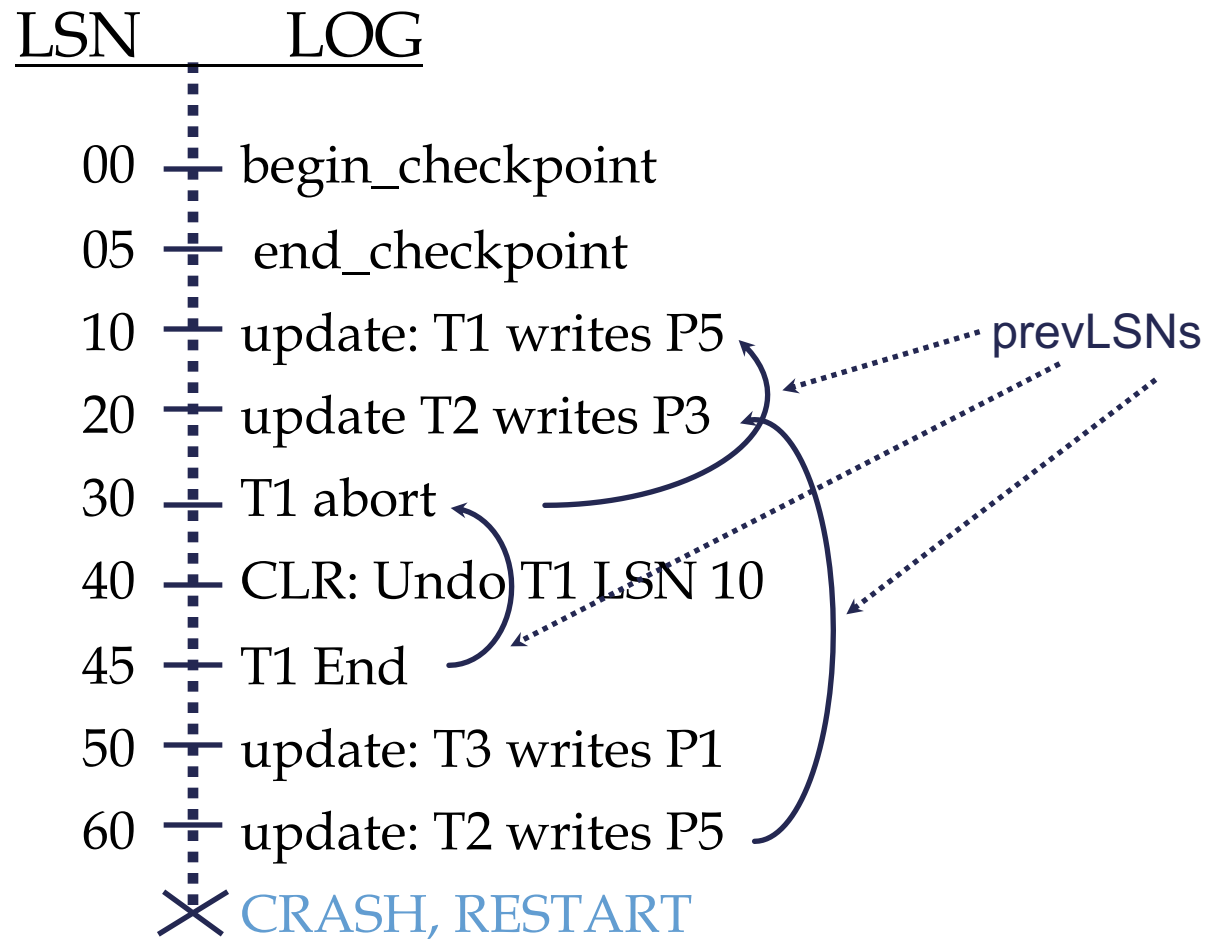
Example of Recovery

RAM

Xact Table
    lastLSN
    status
Dirty Page Table
    recLSN
flushedLSN

ToUndo

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

prevLSNs

CRASH, RESTART

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!