Crash Recovery

Chapter 18

Review: The ACID properties

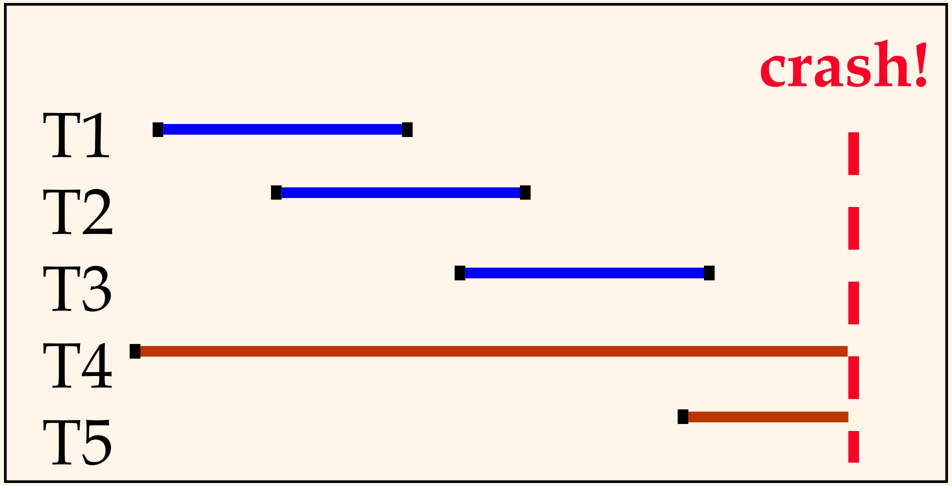
- **A** tomicity: All actions in the Xact happen, or none happen.
- **C** onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one Xact is isolated from that of other Xacts.
- **D** urability: If a Xact commits, its effects persist.

- 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).



Handling the Buffer Pool

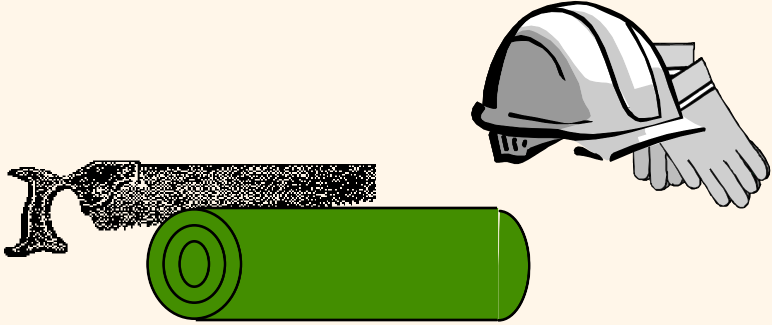
- Force every write to disk?
  - Poor response time.
  - But provides durability.
- Steal buffer-pool frames from uncommited Xacts?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?

	No Steal	Steal
Force	Trivial	
No Force		Desired

More on Steal and Force

- **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>

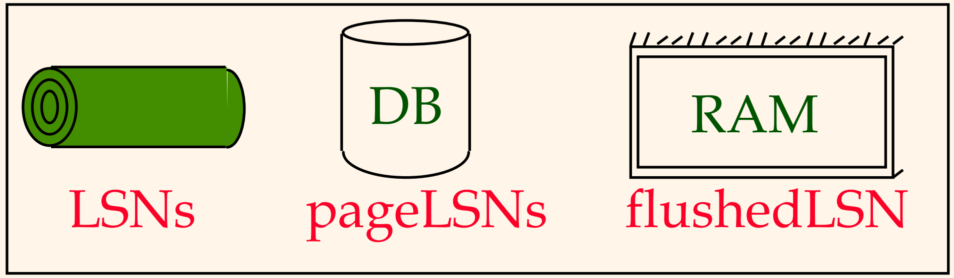
  - and additional control info (which we'll see soon).

Write-Ahead Logging (WAL)

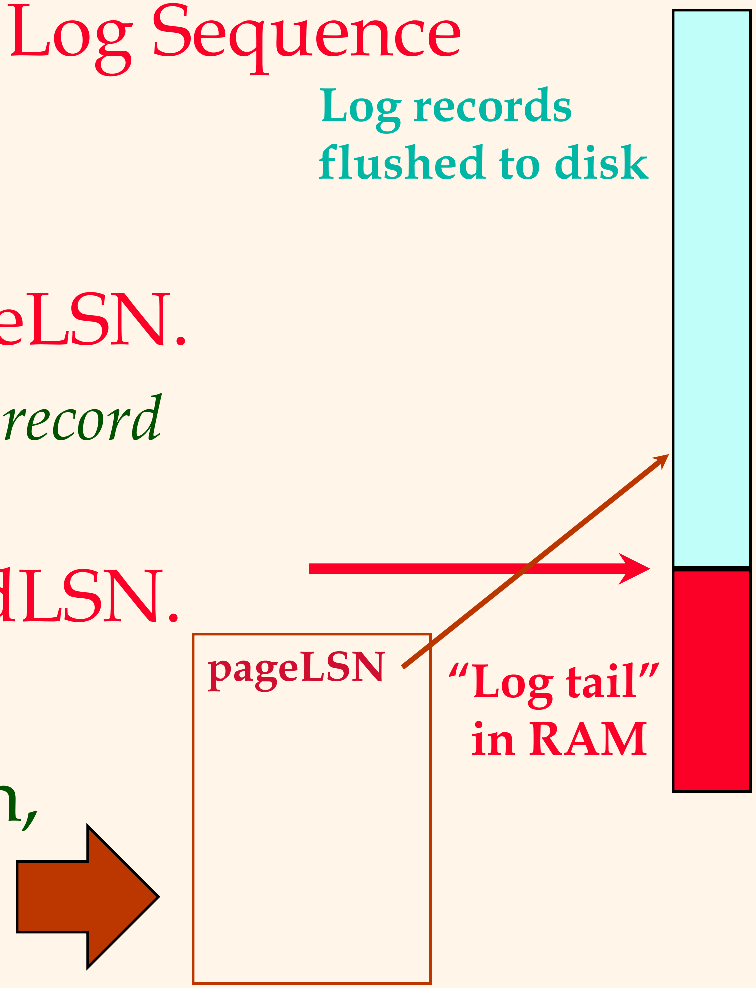
- The Write-Ahead Logging Protocol:
  ① Must force the log record for an update *before* the corresponding data page gets to disk.
  ② Must write all log records for a Xact *before commit*.
- #1 guarantees Atomicity.
- #2 guarantees Durability.

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

WAL & the Log



- 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 ≤ flushedLSN



Log Records

Possible log record types:

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

LogRecord fields:

prevLSN
XID
type

update records only:

pageID
length
offset
before-image
after-image

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.

Other Log-Related State

- Transaction Table:
  - One entry per active Xact.
  - Contains XID, status (running/commited/aborted), and lastLSN (The LSN of the most recent log record).
- 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.

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

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

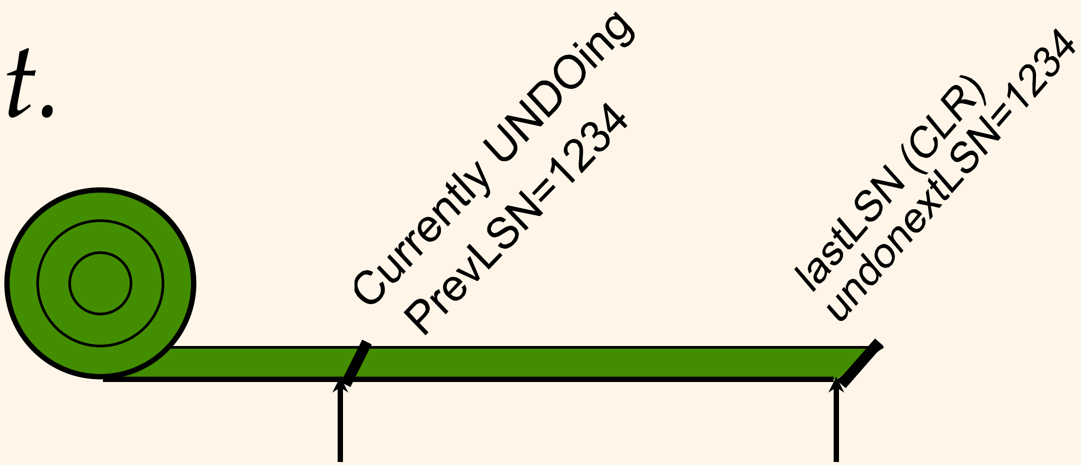
recLSN

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.
  - Before starting UNDO, write an *Abort* log record.
    - For recovering from crash during UNDO!
  - Get lastLSN of Xact from Xact table.
  - Can follow chain of log records backward via the prevLSN field.

Abort, cont.

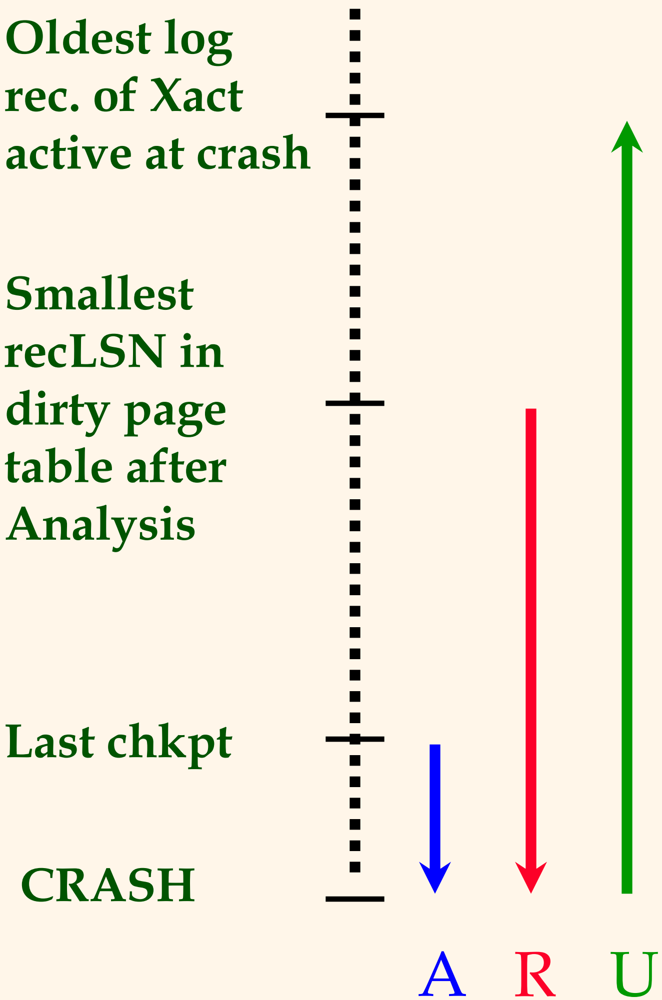


- To perform UNDO, must have a lock on data!
- Before restoring old value of a page, write a CLR:
  - You continue logging while you UNDO!!
  - CLR has one extra field: undonextLSN
    - Points to the next LSN to undo (i.e. the prevLSN of the record we're currently undoing).
  - CLRs *never* Undone (but 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



- Start from a checkpoint (found via master record).
- Three phases. Need to:
  - ANALYSIS. Figure out which Xacts committed/failed since checkpoint.
  - REDO actions of dirty pages.
  - UNDO effects of failed Xacts.

Recovery: The Analysis Phase

- The *analysis* phase performs three tasks:

  1. It determines the point in the log at which to start the redo pass

  2. It determines pages in the buffer pool that were dirty at the time of crash

  3. It identifies transactions that were active at the time of crash and must be undone

Recovery: The Analysis Phase

- *Analysis* begins by examining the most recent begin_checkpoint log record

  - Initialize the dirty page table and transaction table to the copies of those structures in the next end_checkpoint record

  - If additional log records are between begin_checkpoint and end_checkpoint records, the tables must be adjusted accordingly.

Recovery: The Analysis Phase

- Scan log forward from checkpoint.
  - End record for T: Remove T from the transaction table where it is not active anymore.
  - Other records for T: Add T to the transaction table if it is not there. The entry for T is modified to be:
    - The LastLSN field is set to current LSN
    - The status is set to C if this log record is commit, o.w., the status is set to U
  - Update record for Page P: Add P to the dirty table if it is not already there with RecLSN = LSN

Recovery: The REDO Phase

- During the REDO phase, we reapply the updates of all transactions to reconstruct state at crash (*repeat History)*
  - Reapply *all* updates (even of aborted Xacts!), redo CLRs.
- The REDO phase starts with the log record that has the smallest recLSN of all pages in the dirty page table.
  - This log record identifies the oldest update that may not have been written to disk

Recovery: The REDO Phase

- Scan forward from smallest recLSN in dirty page table until the end of the log.

- For each rodoable log record (Update or CLR), check weather the logged action must be redone.

- To REDO an action:
  - Reapply logged action.
  - Set pageLSN to LSN. No additional logging!

Recovery: The REDO Phase

- All redoable actions must be redone unless one of the following conditions holds:
  1. Affected page is not in the Dirty Page Table. This means that this page has alreay made it to disk
  2. Affected page is in the Dirty Page Table, but has recLSN > LSN. This means that this change is not the one that is responsible for making this page dirty, i.e., this change has already made it to disk
  3. Affected page is in the Dirty Page Table, but has pageLSN that is greater than or equal LSN. This means that either this update or a later update to the page was written to disk.

Recovery: The UNDO Phase

- Unlike the other two phases, the UNDO phase scans backward from the end of the log
- The goal of this phase is to undo the actions of all transactions active at the time of the crash.
- Undo identifies a set of *loser transactions* by scanning the transaction table constructed by the analysis phase and selecting those transactions with status U

Recovery: The UNDO Phase

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

Repeat:

- Choose largest LSN among ToUndo.
- If this LSN is an update. Undo the update, write a CLR, add prevLSN to 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

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, RESTART

prevLSNs

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

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!