Recovery: Basic Concepts
Recovery: ARIES, normal operation

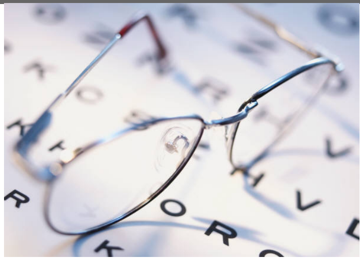
ACS, Marcos Vaz Salles

Do-it-yourself-recap: The many faces of atomicity

- **Atomicity** is strong modularity mechanism!
  - Hides that one high-level action is actually made of many sub-actions

- **Before-or-after** atomicity
  - == Isolation
  - Cannot have effects that would only arise by interleaving of parts of transactions

- **What was the meaning of all-or-nothing atomicity?**



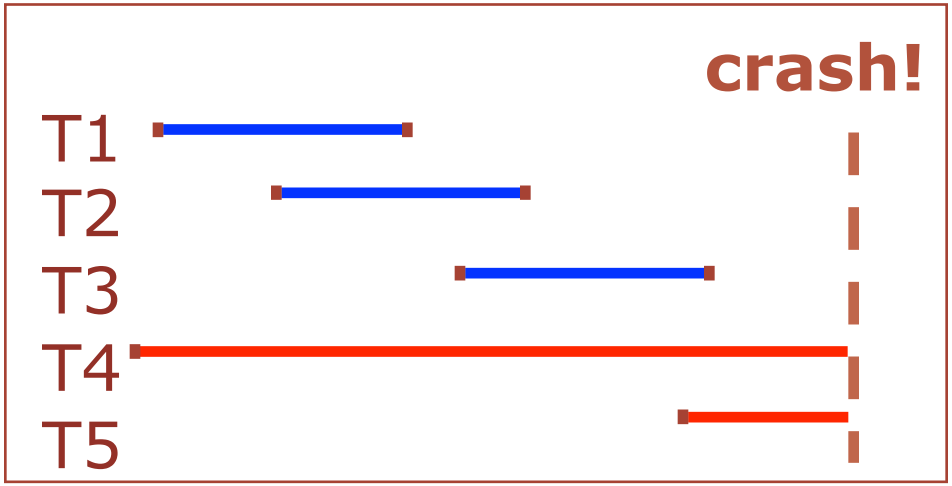
What should we learn today?

- Explain the concepts of volatile, nonvolatile, and stable storage as well as the main assumptions underlying database recovery
- Predict how force/no-force and steal/no-steal strategies for writes and buffer management influence the need for redo and undo
- Explain the notion of logging and the concept of write-ahead logging
- Predict what portions of the log and database are necessary for recovery based on the recovery equations
- Explain how write-ahead logging is achieved in the ARIES protocol
- Explain the functions of recovery metadata such as the transaction table and the dirty page table
- Interpret the contents of the log resulting from ARIES normal operation

Implementing All-or-Nothing Atomicity

- Atomicity
  - Transactions may abort (“Rollback”).
- Durability
  - What if system stops running? (Causes?)

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



Source: Ramakrishnan & Gehrke (partial)

Assumptions

- Concurrency control is in effect
  - Strict 2PL, in particular
- Updates are happening "in place"
  - i.e. data is overwritten on (deleted from) memory using `READ` / `WRITE` interface.
  - We will use a two-level memory with buffer and disk
- Types of failures
  - Crash
  - Media failure
- Always fail-stop!

Source: Ramakrishnan & Gehrke (partial)

Volatile vs. Nonvolatile vs. Stable Storage

- Volatile Storage
  - Lost in the event of a crash
  - Example: main memory
- Nonvolatile Storage
  - Not lost on crash, but lost on media failure
  - Example: disk
- Stable Storage
  - Never lost (otherwise, that's it ☺)
  - How do you implement this one?

Source: Ramakrishnan & Gehrke (partial)

Surviving Crashes: How to handle the Buffer Pool?

Fill in the matrix: When do you need to UNDO changes? When do you need to REDO changes?

- 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

Source: Ramakrishnan & Gehrke (partial)

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.

Source: Ramakrishnan & Gehrke (partial)

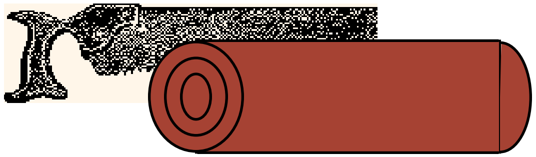
Undo/Redo vs. Force/Steal

	No Steal	Steal
Force	No Redo No Undo	No Redo Undo
No Force	Redo No Undo	$\boxed{\text{Redo Undo}}$

How do we support this option?

Source: Ramakrishnan & Gehrke (partial)

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
  - Logical vs. Physical Logging
  - Example physical log record contains:

    <XID, pageID, offset, length, old data, new data>

  - Good compromise is physiological logging.

Source: Ramakrishnan & Gehrke (partial)

Write-Ahead Logging (WAL)

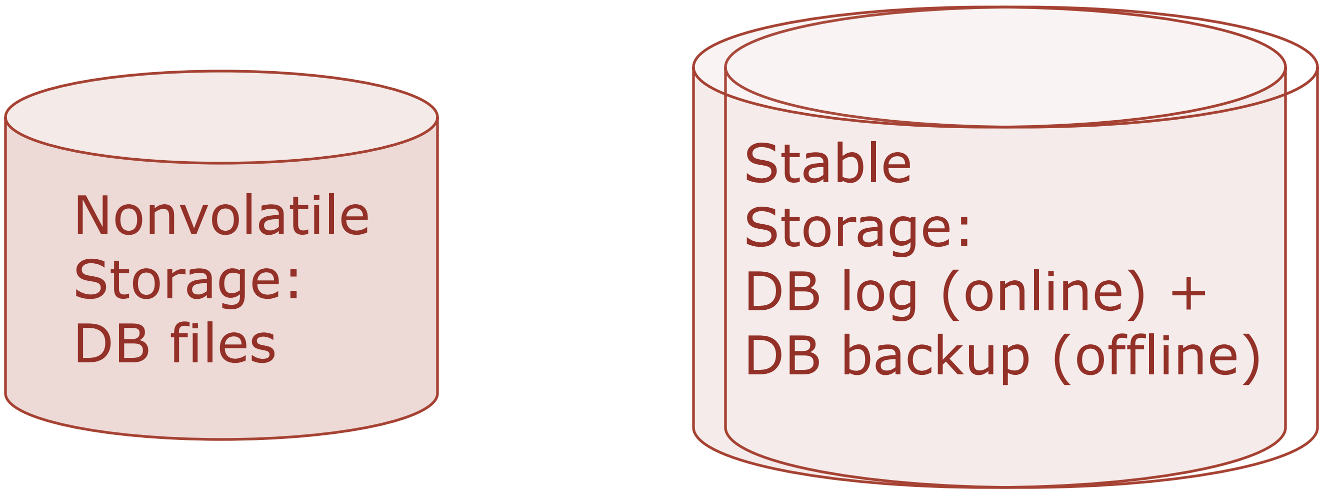
- Golden Rule: Never modify the only copy!

- 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 will study the ARIES algorithms.

Source: Ramakrishnan & Gehrke (partial)

Recovery Equations



Discussion:

- Crash Recovery: volatile memory lost
  - Current DB = DB files + DB log → since when?
- Media Recovery: nonvolatile storage lost
  - Current DB = DB backup + DB log → since when?
- We will focus on crash recovery next

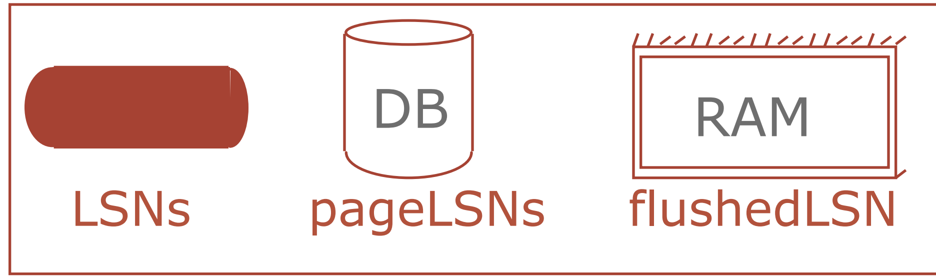
Source: Ramakrishnan & Gehrke (partial), Shasha & Bonnet (partial)

Undo/Redo vs. Force/Steal

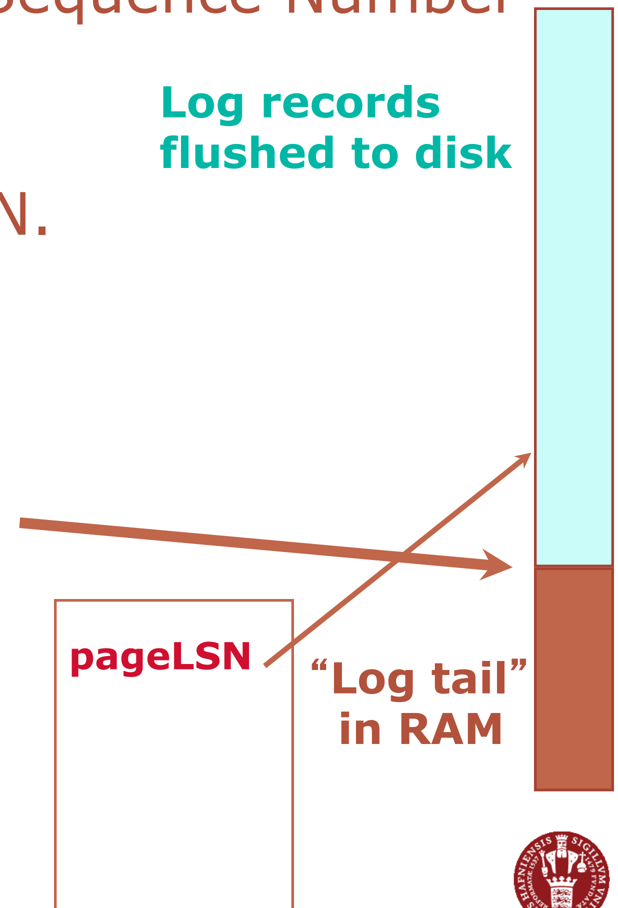
	No Steal	Steal
Force	**No Redo** **No Undo**	**No Redo** **Undo**
No Force	**Redo** **No Undo**	**Redo** **Undo**

How do we support this option?

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

LogRecord fields:

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

update records only

Possible log record types:

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

Note: Format above simplified; in reality, ARIES uses physiological variant

Source: Ramakrishnan & Gehrke (partial)

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

Keep in Mind:
It must be OK to crash at **any time**
→ **repeat history!**

- 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 → concurrency is correctly handled
- 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 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, minDirtyPagesLSN.
    - Use **background process** to flush dirty pages to disk!
  - Store LSN of chkpt record in a safe place (*master* record).

Transaction Commit

- Write commit record to log.
- 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.

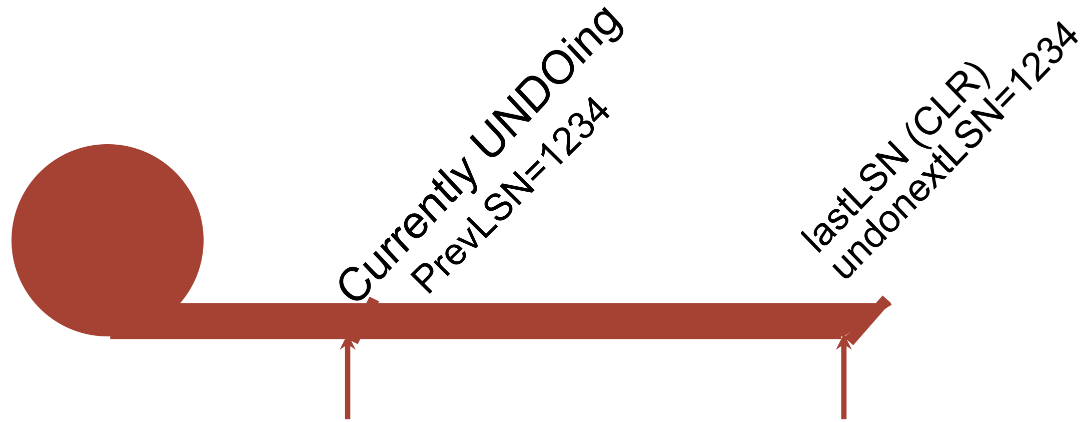
Why?

Source: Ramakrishnan & Gehrke (partial)

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!
  - Strict 2PL enforces this
- 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.

Example

- 10 T1 writes P5
- 20 T2 writes P17
- 30 T1 writes P3

P3 written to disk

(pageLSN for page 3 at this time is 30)

- 40 T1 aborts
- 50 CLR T1 P3 (undonextLSN: 10)
- 60 CLR T1 P5 (undonextLSN: NULL)
- 70 End T1

Source: Ramakrishnan & Gehrke (partial)

A Longer Example

Discussion: Does this example make sense? Can you explain to your colleague what happened?

- 10 T1 writes P3 (prevLSN: NULL)
- 20 T2 writes P4 (prevLSN: NULL)
- 30 T2 writes P5 (prevLSN: 20)
- flushedLSN = 20

P4 gets written to disk (pageLSN for page 4 = 20)
T2 aborts

- 50 Abort T2
- 60 CLR T2 P5 (undoNextLSN = 20), pageLSN(P5)=60

Update P5 in the buffer manager
Flush log up to log record 60
Buffer manager writes P5 to disk.

- 70 CLR T2 P4 (undoNextLSN = NULL), pageLSN(P4)=70

Update page P4

- 80 End T2
- 90 T1 commits

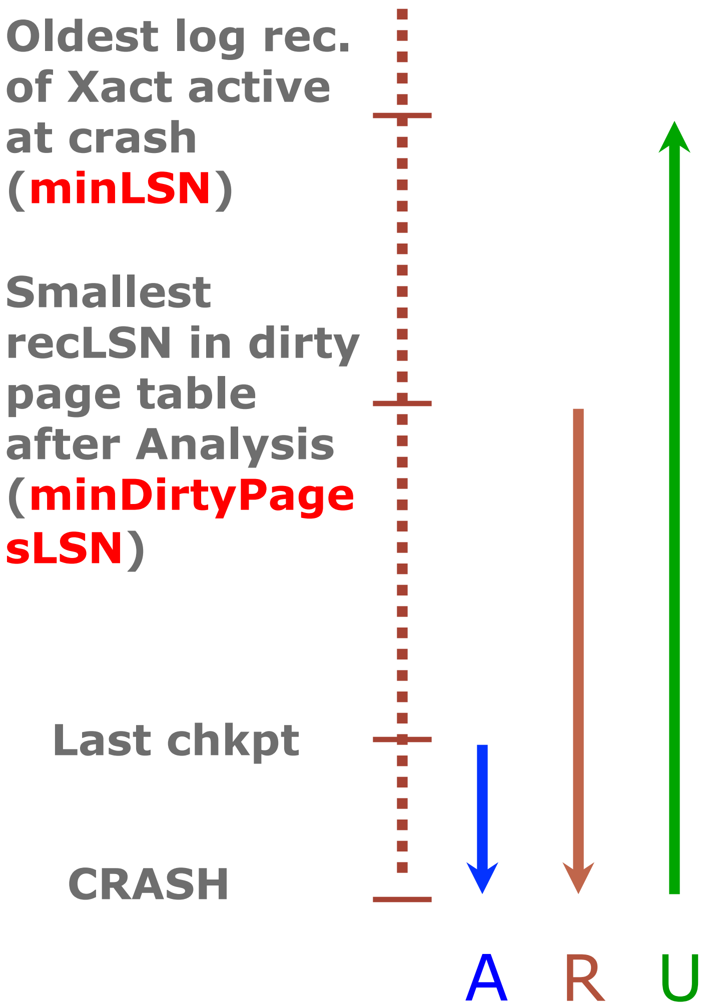
Flush log up to log record 90, then the commit(T1) returns

Source: Ramakrishnan & Gehrke (partial)

Questions so far?

Keep in Mind:
It must be OK to crash at **any time** (including during recovery)

Crash Recovery: Big Picture



- 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.

Source: Ramakrishnan & Gehrke (partial)

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) >= LSN.

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

Source: Ramakrishnan & Gehrke (partial)

Recovery: The UNDO Phase

ToUndo={ *lsn* | *lsn* 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
	CRASH, RESTART

prevLSNs

Source: Ramakrishnan & Gehrke (partial)

Discussion: Explain to your colleague!

Example: Crash During Restart!

RAM

Xact Table
- lastLSN
- status

Dirty Page Table
- recLSN

flushedLSN

ToUndo

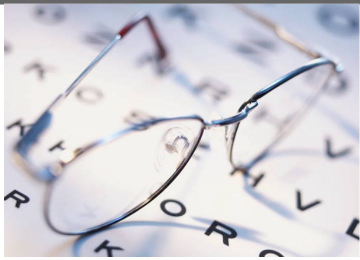
LSN	LOG
00,05	begin_checkpoint, end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40,45	CLR: Undo T1 LSN 10, T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART
64,65,70	T2 abort, T3 abort, CLR: Undo T2 LSN 60
80,85	CLR: Undo T3 LSN 50, T3 end
	CRASH, RESTART
90, 95	CLR: Undo T2 LSN 20, T2 end

undonextLSN

Source: Ramakrishnan & Gehrke (partial)

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.



What should we learn today?

- Explain the concepts of volatile, nonvolatile, and stable storage as well as the main assumptions underlying database recovery
- Predict how force/no-force and steal/no-steal strategies for writes and buffer management influence the need for redo and undo
- Explain the notion of logging and the concept of write-ahead logging
- Predict what portions of the log and database are necessary for recovery based on the recovery equations
- Explain how write-ahead logging is achieved in the ARIES protocol
- Explain the functions of recovery metadata such as the transaction table and the dirty page table
- Interpret the contents of the log resulting from ARIES normal operation