

## Transaction Management: Crash Recovery, part 2

CS634  
Class 21, Apr 20, 2016

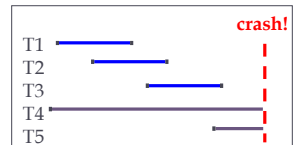
Slides based on "Database Management Systems" 3rd ed, Ramakrishnan and Gehrke

## Motivation

- ▶ **Atomicity:**
  - ▶ Transactions may abort – must **rollback** their actions
- ▶ **Durability:**
  - ▶ What if DBMS stops running – e.g., power failure?

Desired Behavior after system restarts:

- T1, T2 & T3 should be **recoverable**
- T4 & T5 should be **aborted** (effects not seen)



## Logging

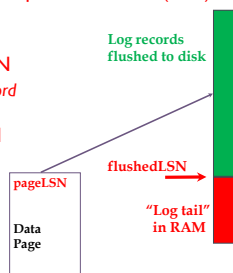
- ▶ **What is in the Log**
  - ▶ Ordered list of REDO/UNDO actions
  - ▶ Update log record contains:
    - <prevLSN, transID, pageID, offset, length, old data, new data>
  - ▶ Old data is called the **before image**
  - ▶ New data called the **after image**
- ▶ The prevLSN provides the LSN of the transaction's previous log record, so it's easy to scan backwards through log records as needed in UNDO processing

## 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 transaction **before commit returns**
- ▶ Property 1 guarantees Atomicity
- ▶ Property 2 guarantees Durability
- ▶ We focus on the **ARIES** algorithm
  - ▶ Algorithms for **R**ecovery and **I**solation **E**xploiting **S**emantics

## How Logging is Done

- ▶ Each log record has a unique **Log Sequence Number (LSN)**
  - ▶ LSNs always increasing
  - ▶ Works similar to "record locator"
- ▶ 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 largest LSN flushed so far
- ▶ **WAL: Before** a page is written, flush its log record such that
  - ▶  $pageLSN \leq flushedLSN$



## Log Records

### LogRecord fields:

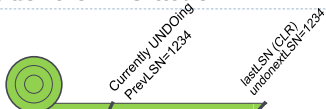
update records only {
 

- prevLSN
- transID
- entryType
- pageID
- length
- offset
- before-image
- after-image

Possible log entry types:

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

## CLR (compensation log record): remember intended/done UNDO action



- ▶ In UNDO processing, before restoring old value of part of a page (say a row), write a CLR to log:
  - ▶ CLR has one extra field: **undonextLSN**
    - ▶ Points to the next LSN to undo (i.e. the prevLSN of the record we're currently undoing).
    - ▶ The undonextLSN value is used, in recovery from a crash, only if this CLR ends up as the last one in the log for a "loser" transaction. Then it points to where in the log to start/resume doing UNDOs of update log records.
  - ▶ CLR's *never* Undone (but they will be Redone if recovery repeats this history).
- ▶ At end of transaction UNDO, write an "end" log record.

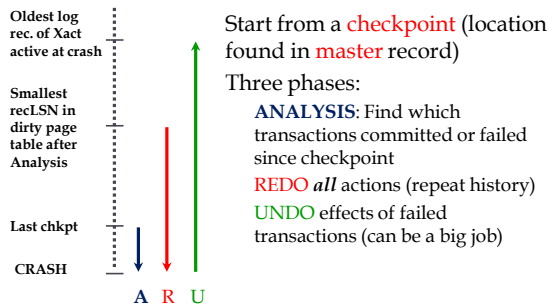
## Other Log-Related State

- ▶ **Transaction Table:**
  - ▶ One entry per active transaction
  - ▶ Contains **transID**, **status** (running/committed/aborted), and **lastLSN** (most recent LSN for transaction)
- ▶ A **dirty page** is one whose disk and buffer images differ
  - ▶ So a dirty page becomes clean at page write, if it stays in buffer
  - ▶ Once clean, can be deleted from dirty page table
  - ▶ And is clean if it gets read back into buffer, even with uncommitted data in it
- ▶ **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 (spec's what part of log relates to redos for this page)
  - ▶ **Earliest recLSN** - important milestone for recovery (spec's what part of log relates to redos for whole system)
- ▶ Both the above are stored in RAM, hence volatile!

## Checkpointing

- ▶ Periodically, the DBMS creates a **checkpoint**
  - ▶ minimize time taken to recover in the event of a system crash
- ▶ Checkpoint logging:
  - ▶ **begin\_checkpoint** record: Indicates when checkpoint began
  - ▶ **end\_checkpoint** record: Contains current **transaction table** and **dirty page table** as of begin\_checkpoint time
  - ▶ So the earliest recLSN is known at recovery time, and the set of live transactions, very useful for recovery
  - ▶ Other transactions continue to run; tables accurate only as of the time of the **begin\_checkpoint** record - **fuzzy** checkpoint
    - ▶ No attempt to force dirty pages to disk!
  - ▶ LSN of **begin\_checkpoint** written in special **master record** on stable storage

## Crash Recovery: Big Picture



## The Analysis Phase

- ▶ Reconstruct state at checkpoint.
  - ▶ from **end\_checkpoint** record
  - ▶ Fill in Transaction table, all with U status (needs undoing)
  - ▶ Fill in DPT
- ▶ Scan log forward from checkpoint
  - ▶ **End** record: Remove T from Transaction table
  - ▶ **Other records:** Add T to transaction table, set **lastLSN=LSN**
  - ▶ If record is **commit** change transaction status to C
  - ▶ **Update** record on page P
    - ▶ If P not in Dirty Page Table, add it & set its **recLSN=LSN**
- ▶ Finished: now all Transactions still marked U are "losers"

## The REDO Phase

- ▶ We **repeat history** to reconstruct state at crash:
  - ▶ Reapply **all** updates (even of aborted transactions), redo CLR's.
- ▶ Redo Update, basic case:
  - ▶ Read in page if not in buffer
  - ▶ Apply change to part of page (often a row)
  - ▶ Leave page in buffer, to be pushed out later (lazy again)
- ▶ Redo CLR:
  - ▶ Do same action as original UNDO:
    - ▶ Read in page if not in buffer, apply change, leave page in buffer
- ▶ But sometimes we don't need to do the redo, check conditions first...

## The REDO Phase in detail

- ▶ We **repeat history** to reconstruct state at crash:
  - ▶ Reapply **all** updates (even of aborted transactions), redo CLR's.
- ▶ Scan forward from log rec containing smallest **reclSN** in Dirty page Table
- ▶ For each CLR or update log rec **LSN**, REDO the action unless:
  - ▶ Affected page is not in the Dirty Page Table (DPT) , or
  - ▶ Affected page is in DPT, but has **reclSN > LSN** or **pageLSN** (in DB)  $\geq$  **LSN** (page is already more up-to-date than this action)
- ▶ To **REDO** an action:
  - ▶ Reapply logged action (read page if not in buffer; change part)
  - ▶ Set **pageLSN** on the page to **LSN**. No additional logging!

## The UNDO Phase, simple case, no rollbacks in progress at crash

In this case, losers have no CLR's in the old log  
**ToUndo** = set of **lastLSN**s for "loser" transactions  
 (ones active at crash)

### Repeat:

- ▶ Choose largest LSN among **ToUndo**
- ▶ This LSN is an **update**. Undo the update, write a **CLR**, add **prevLSN** to **ToUndo**

### Until **ToUndo** is empty

- ▶ i.e. move backwards through update log records of all loser transactions, doing **UNDOS**
- ▶ End up with a bunch of **CLR**'s in log to document what was done

## The UNDO Phase, general case

Hard to understand from algorithmic description (pg. 592).

Note goals:

- ▶ All actions of losers must be undone
- ▶ These actions must be undone in reverse log order
- ▶ Reason for reverse order:
  - ▶ T1 changes A from 10 to 20, then from 20 to 30
  - ▶ Undo: 30->20, 20->10.
- ▶ Idea: CLR marks that a certain update has been undone, and points to next-earlier update log record that needs attention
- ▶ So last CLR in the log for a transaction tells the story: undo is finished, or processing needs to start undo work with pointed-to update
- ▶ In fact, once that last CLR is processed, the undo processing follows a chain of update entries for that transaction back through the log, never studying an old CLR again, but writing new ones.

## Example of Recovery: after crash 1

Recovery after crash:  
 Analysis: from ckpt,

TxnTable (TT): empty  
 DPT: empty

Scan forward, find:

In TT: T2, T3  
 Losers: T2, T3  
 In DPT: P5, P3, P1  
 Smallest reclSN: 10

Redo:

scan forward from 10:

10, 20: redo updates  
 40: redo CLR  
 45: drop T1 from TT  
 50, 60: redo updates

Undo...

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

## Example: after crash 1, undo phase (Simple case of no Rollbacks in progress at Crash)

Recovery after crash, undo phase:

From analysis,

Losers: T2, T3  
 lastLSN's of losers =  
 ToUndo = {60, 50}

Scan back using ToUndo:

60: undo update, write CLR, put 20 in ToUndo  
 now ToUndo = {50, 20}  
 50: undo update, write CLR, nothing to put in ToUndo, write T3 end record  
 now ToUndo = {20}  
 20: undo update, write CLR, nothing to put in ToUndo, write T2 end record  
 Done with recovery, but we consider a crash before 20 is written to disk...

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

Annotations: *prevLSNs* (pointing to 10, 20, 40), *undonextLSN* (pointing to 20)

## Example: Crash During Restart!

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

Annotations: *undonextLSN* (pointing to 70), *Crash1 recovery undo phase writes these 2 CLR's, then gets interrupted by crash* (pointing to 70, 80)

## Example: Crash During Restart!

	LSN	LOG
<b>Case with undos in progress at crash:</b> Process last CLR to find out where to start UNDOing a transaction	00,05	begin_checkpoint, end_checkpoint
	10	update: T1 writes P5
	20	update: T2 writes P3
<b>From analysis,</b> Losers: T2 lastLSNs of losers = ToUndo = {70}	30	T1 abort
	40,45	CLR: Undo T1 LSN 10, T1 End
	50	update: T3 writes P1
<b>Redo:</b> same as before <b>Undo:</b> Scan back using ToUndo: 70: CLR, put undonextLSN = 20 in ToUndo now, ToUndo = {20} 20: undo update, write CLR, nothing to put in ToUndo, write T2 end record Done with recovery	60	update: T2 writes P5
	×	CRASH, RESTART
	70	CLR: Undo T2 LSN 60
	80,85	CLR: Undo T3 LSN 50, T3 end
	×	CRASH, RESTART

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.
  - ▶ Fix “hot spots” if you can!
- ▶ How do you limit the amount of work in UNDO?
  - ▶ Avoid long-running Xacts. Good idea anyway.

## 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.
- ▶ Without checkpointing, need to process entire log, i.e., back to last DB-start
- ▶ 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 CLR's.
- ▶ Redo “repeats history”: Simplifies the logic!

## Recovering From a Crash

- ▶ There are 3 phases in the *Aries* recovery algorithm:
  - ▶ **Analysis:** Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
  - ▶ **Redo:** Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
  - ▶ **Undo:** The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)

## Logging Logical Operations

- ▶ The log entry studied here has page number, offset on page, number of bytes
- ▶ These are called physical operations, or byte-level operations
- ▶ But the ARIES system also supports *logical operations* like “insert this row (...) into table T”, as discussed on page 596.
- ▶ This works better with B-tree mods
- ▶ The chapter assumes page locks, so a whole page is locked for an insert
- ▶ But current DBs use row locks and logical logging, or physiological logging, which targets pages and uses logical operations in the page.