Carnegie Mellon Univ.
Dept. of Computer Science
15-415 - Database Applications

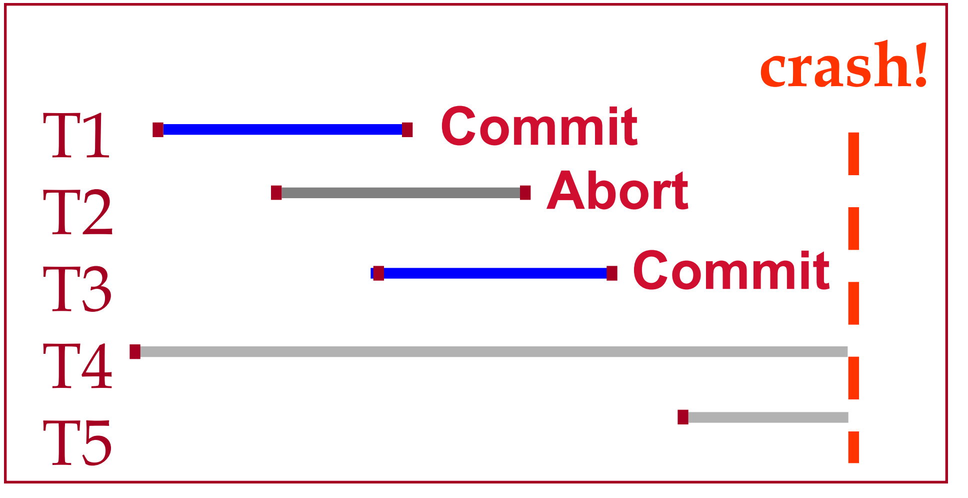
Crash Recovery - part 2
(R&G, ch. 18)

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).



General Overview

- Preliminaries
- Write-Ahead Log - main ideas
- (Shadow paging)
- Write-Ahead Log: ARIES

Main ideas so far:

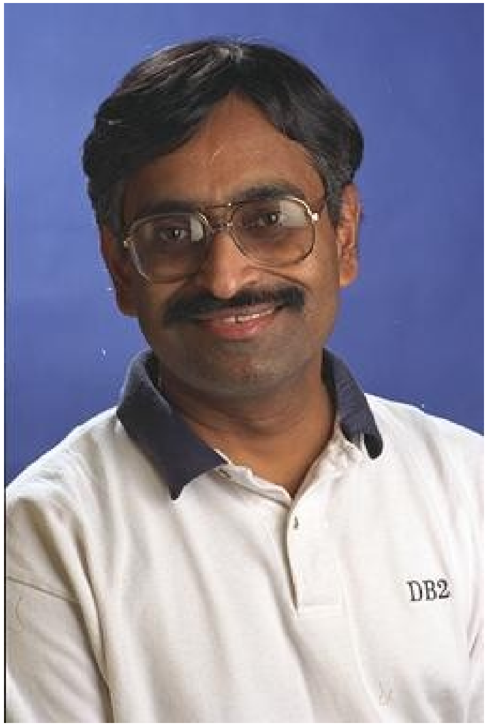
- Write-Ahead Log, for loss of volatile storage,
- with incremental updates (STEAL, NO FORCE)
- and checkpoints
- On recovery: **undo** uncommitted; **redo** committed transactions.

Today: ARIES

With full details on

- fuzzy checkpoints
- recovery algorithm

C. Mohan (IBM)



Overview

- Preliminaries
- Write-Ahead Log - main ideas
- (Shadow paging)
- Write-Ahead Log: ARIES
  - LSN's
  - examples of normal operation & of abort
  - fuzzy checkpoints
  - recovery algo

LSN

- Log Sequence Number
- every log record has an LSN
- Q: Why do we need it?

LSN

A1: e.g, undo T4 - it is faster, if we have a linked list of the T4 log records

A2: and many other uses - see later

<T1 start>

<T2 start>

<T4 start>

<T4, A, 10, 20>

<T1 commit>

<T4, B, 30, 40>

<T3 start>

<T2 commit>

<T3 commit>

~~~~ CRASH ~~~~

Types of log records

Q1: Which types?

A1:

Q2: What format?

A2:

<T1 start>

<T2 start>

<T4 start>

<T4, A, 10, 20>

<T1 commit>

<T4, B, 30, 40>

<T3 start>

<T2 commit>

<T3 commit>

~~~~ CRASH ~~~~

Types of log records

Q1: Which types?

A1: Update, commit, ckpoint, …

Q2: What format?

A2: x-id, type, (old value, …)

<T1 start>

<T2 start>

<T4 start>

<T4, A, 10, 20>

<T1 commit>

<T4, B, 30, 40>

<T3 start>

<T2 commit>

<T3 commit>

~~~~ CRASH ~~~~

Log Records

LogRecord fields:

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

(**update** records only: pageID, length, offset, before-image, after-image)

Possible log record types:

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

Overview

- Preliminaries
- Write-Ahead Log - main ideas
- (Shadow paging)
- Write-Ahead Log: ARIES
  - LSN's
  - ➡ examples of normal operation & of abort
  - fuzzy checkpoints
  - recovery algo

Writing log records

- We don’t want to write one record at a time
  - (why not?)
- How should we buffer them?

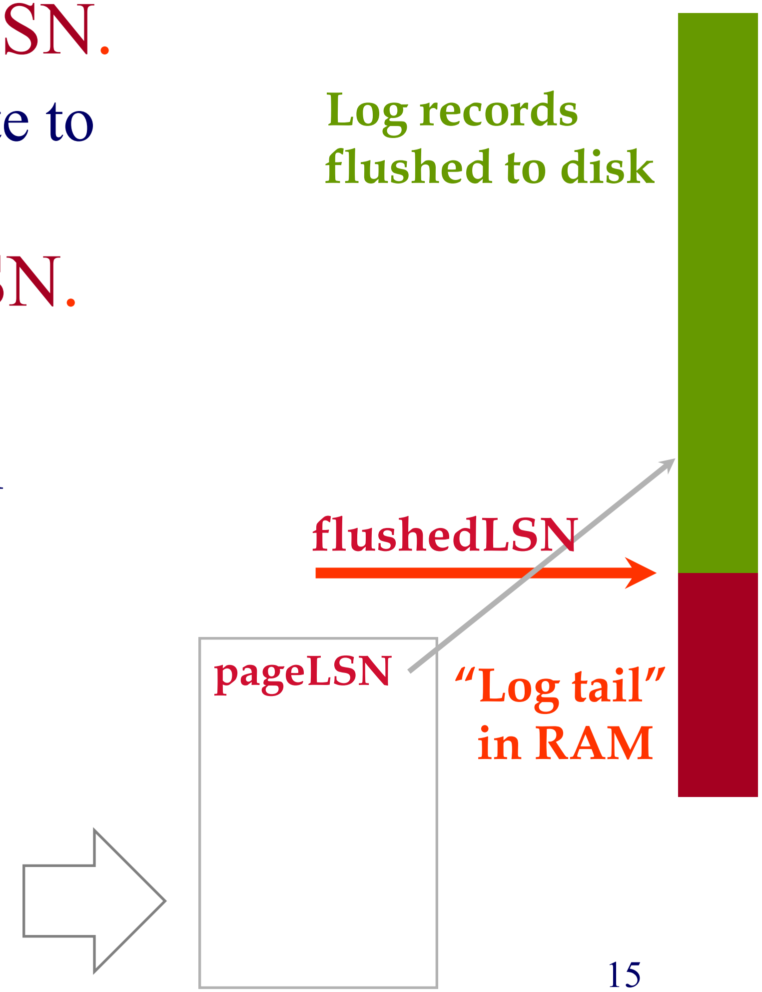
Writing log records

- We don't want to write one record at a time
  - (why not?)
- How should we buffer them?
  - Batch log updates;
  - Un-pin a data page ONLY if all the corresponding log records have been flushed to the log.

WAL & the Log

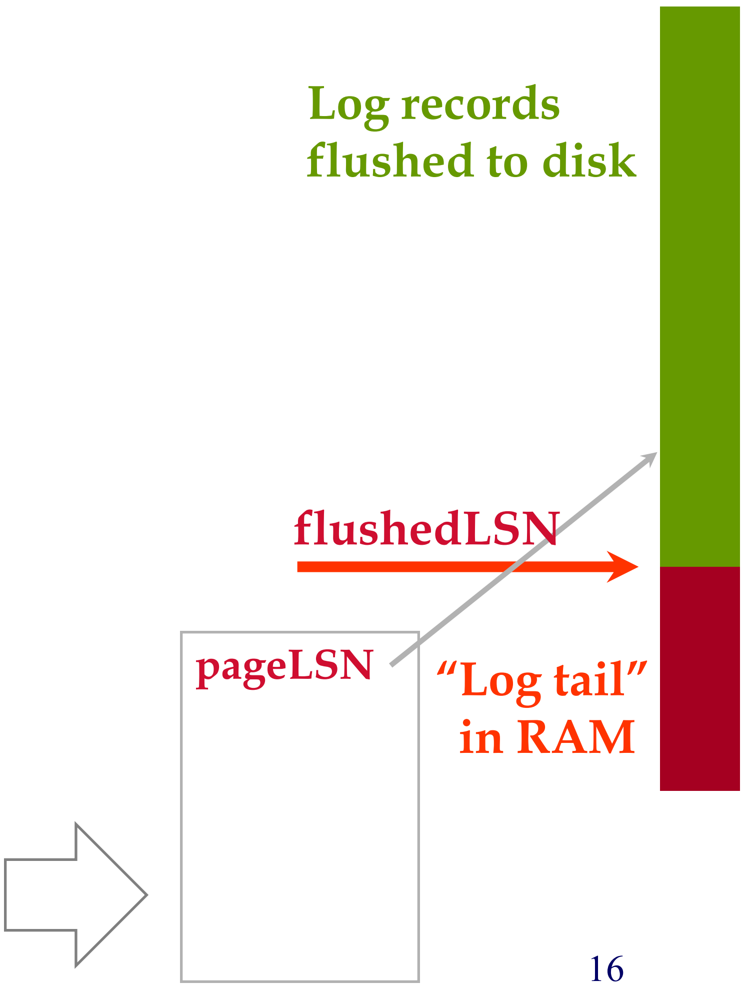
- Each data page contains a pageLSN.
  - The LSN of the **most recent** update to that page.
- System keeps track of flushedLSN.
  - The max LSN flushed so far.
- WAL: For a page *i* to be written must flush log at least to the point where:

  $pageLSN_i \leq flushedLSN$



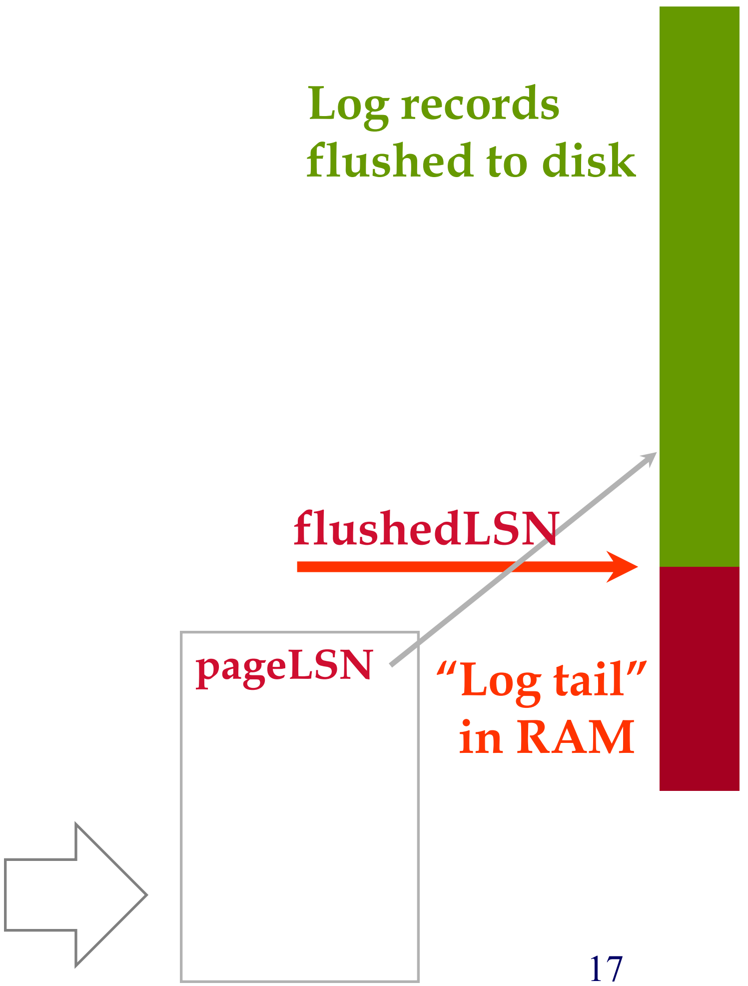
WAL & the Log

- Can we un-pin the gray page?



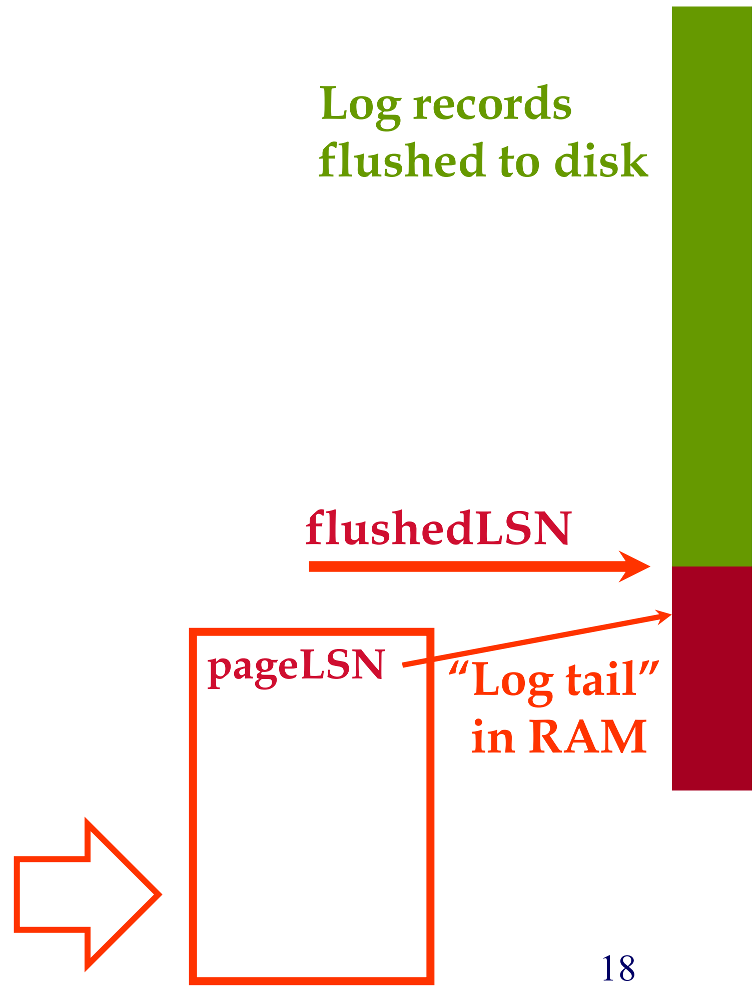
WAL & the Log

- Can we un-pin the gray page?
- A: yes



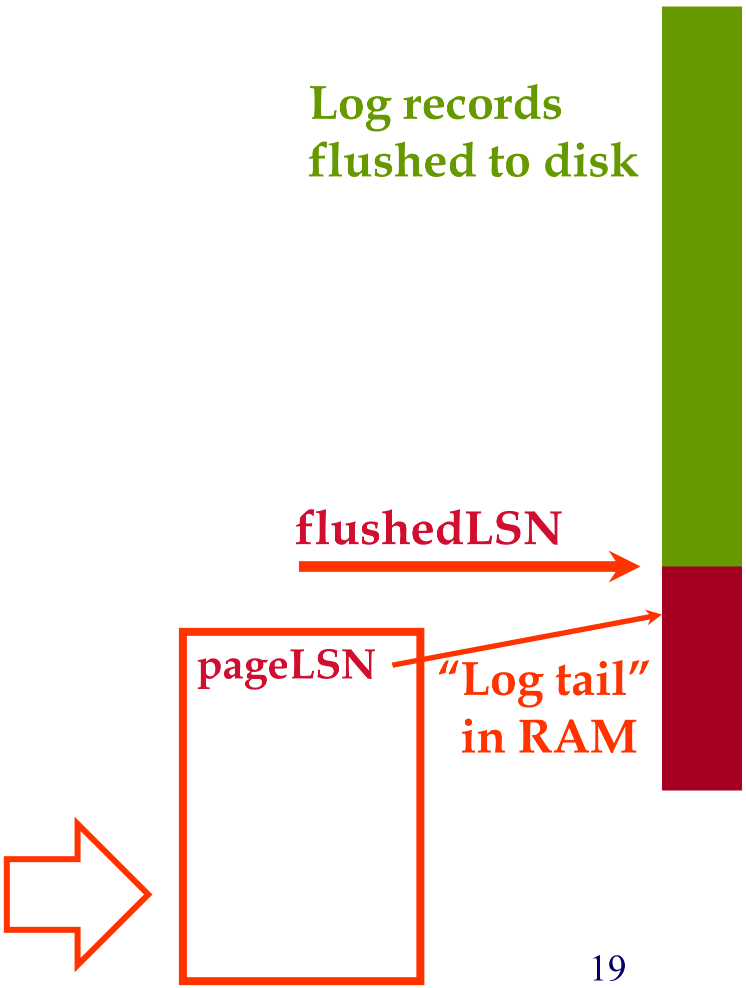
WAL & the Log

- Can we un-pin the red page?

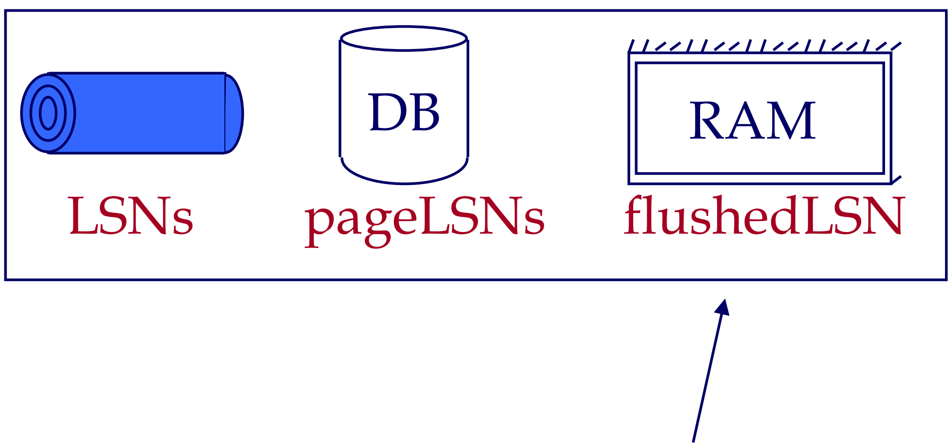


WAL & the Log

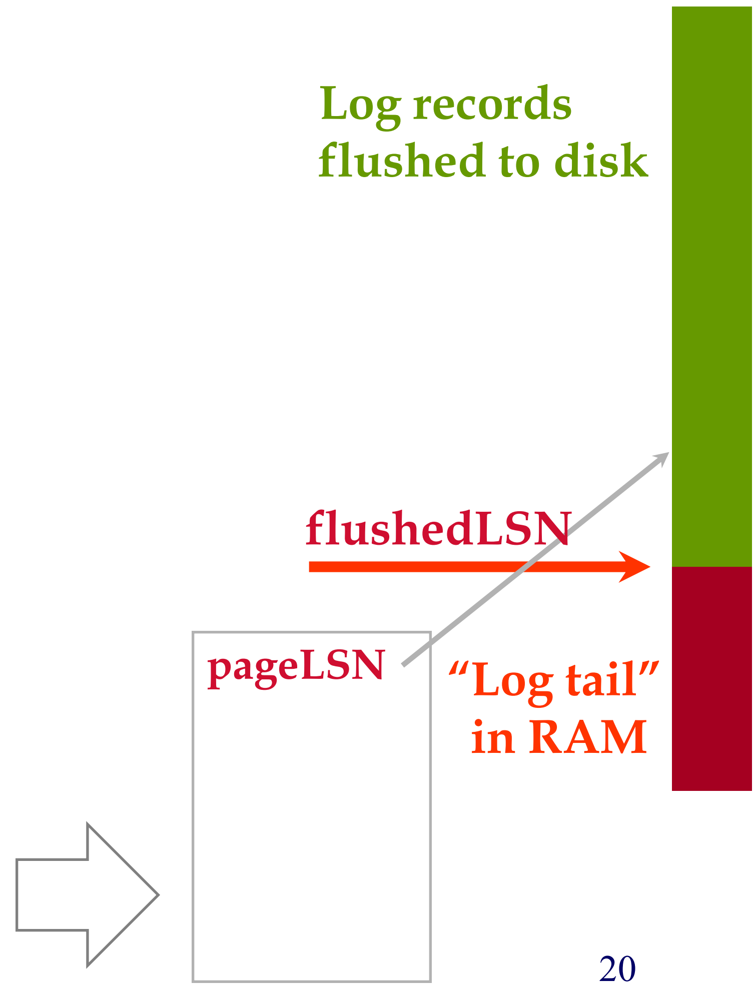
- Can we un-pin the red page?
- A: no



WAL & the Log



Q: why not on disk or log?



Overview

- Preliminaries
- Write-Ahead Log - main ideas
- (Shadow paging)
- Write-Ahead Log: ARIES
  - LSN's
  - examples of **normal operation** & of abort
  - fuzzy checkpoints
  - recovery algo

Normal Execution of an Xact

- Series of reads & **writes**, followed by **commit** or **abort**.
  - We will assume that disk write is atomic.
    - In practice, additional details to deal with non-atomic writes.
- **Strict 2PL**.
- STEAL, NO-FORCE buffer management, with **Write-Ahead Logging.**

Normal execution of an Xact

- Page ‘*i*’ can be written out only after the corresponding log record has been flushed

Transaction Commit

- Write commit record to log.
- All log records up to Xact's commit record are flushed to disk.

Q: why not flush the dirty pages, too?

Transaction Commit

- Write commit record to log.
- All log records up to Xact's commit record are flushed to disk.
  - Note that log flushes are sequential, synchronous writes to disk.
  - Many log records per log page.
- Commit() returns.
- Write end record to log.

Example

LSN	*prevLSN*	*tid*	*type*	*item*	*old*	*new*
10	NULL	T1	update	X	30	40
….						
50	10	T1	update	Y	22	25
…						
63	50	T1	commit			
…						
68	63	T1	end			

dbms flushes
log records
+ some
record-keeping

Overview

- Preliminaries
- Write-Ahead Log - main ideas
- (Shadow paging)
- Write-Ahead Log: ARIES
  - LSN's
  - ➡ examples of normal operation & of **abort**
  - fuzzy checkpoints
  - recovery algo

Abort

Actually, a special case of the up-coming 'undo' operation,

applied to only one transaction - e.g.:

Abort - Example

LSN	*prevLSN*	*tid*	*type*	*item*	*old*	*new*
10	NULL	T2	update	Y	30	40
...						
63	10	T2	abort			

Abort - Example

LSN	*prevLSN*	*tid*	*type*	*item*	*old*	*new*
10	NULL	T2	update	Y	30	40
...						
63	10	T2	abort			
...						
72	63	T2	CLR	(LSN 10)		
...						
78	72	T2	end			

compensating log record (LSN 72)

Abort - Example

LSN	*prevLSN*	*tid*	*type*	*item*	*old*	*new*	*undoNextLSN*
10	NULL	T2	update	Y	30	40	
…							
63	10	T2	abort				
…							
72	63	T2	CLR	Y	40	30	NULL
…							
78	72	T2	end				

CLR record - details

- a CLR record has all the fields of an 'update' record
- plus the 'undoNextLSN' pointer, to the next-to-be-undone LSN

Abort - algorithm:

- First, write an ‘abort’ record on log and
- Play back updates, in reverse order: for each update
  - write a CLR log record
  - restore old value
- at end, write an ‘end’ log record

Notice: CLR records never need to be undone

Overview

- Preliminaries
- Write-Ahead Log - main ideas
- (Shadow paging)
- Write-Ahead Log: ARIES
  - LSN's
  - examples of normal operation & of **abort**
  - fuzzy checkpoints
  - recovery algo

(non-fuzzy) checkpoints

- they have performance problems - recall from previous lecture:

(non-fuzzy) checkpoints

We assumed that the DBMS:

- stops all transactions, and
- flushes on disk the ‘dirty pages’

Both decisions are expensive

Q: Solution?

<T1 start>

...

<T1 commit>

...

<T499, C, 1000, 1200>

<checkpoint>

<T499 commit>

<T500 start>

before

<T500, A, 200, 400>

<checkpoint>

<T500, B, 10, 12>

crash

(non-fuzzy) checkpoints

Q: Solution?

Hint1: record state as of the beginning of the ckpt

Hint2: we need some guarantee about which pages made it to the disk

<T1 start>

...

<T1 commit>

...

<T499, C, 1000, 1200>

<checkpoint>

<T499 commit>

<T500 start>

before

<T500, A, 200, 400>

<checkpoint>

<T500, B, 10, 12>

crash

checkpoints

Q: Solution?

A: write on the log:

- the id-s of active transactions and
- the id-s (ONLY!) of dirty pages (rest: obviously made it to the disk!)

<T1 start>

...

<T1 commit>

...

<T499, C, 1000, 1200>

<checkpoint>

<T499 commit>

<T500 start>

<T500, A, 200, 400>

<checkpoint>

<T500, B, 10, 12>

before

crash

(Fuzzy) checkpoints

Specifically, write to log:

- begin_checkpoint record: indicates start of ckpt
- 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.

(Fuzzy) checkpoints

Specifically, write to log:

- begin_checkpoint record: indicates start of ckpt
- 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.

solved both problems of non-fuzzy ckpts!!

(Fuzzy) checkpoints - cont'd

And:

- Store LSN of most recent chkpt record on disk (master record)
  - Q: why do we need that?

(Fuzzy) Checkpoints

More details: Two in-memory tables:

#1) **Transaction Table**

Q: what would you store there?

(Fuzzy) Checkpoints

More details: Two in-memory tables:

#1) **Transaction Table**

- One entry per currently active Xact.
  - entry removed when Xact commits or aborts
- Contains
  - XID,
  - status (running/committing/aborting), and
  - lastLSN (most recent LSN written by Xact).

(Fuzzy) Checkpoints

#2) **Dirty Page Table**:

- One entry per dirty page currently in buffer pool.
- Contains recLSN -- the LSN of the log record which ***first*** caused the page to be dirty.

Overview

- Preliminaries
- Write-Ahead Log - main ideas
- (Shadow paging)
- Write-Ahead Log: ARIES
  - LSN's
  - examples of normal operation & of **abort**
  - fuzzy checkpoints
  - recovery algo

The Big Picture: What's Stored Where

copy on bboard

LogRecords

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

update
CLR

CLR — *undoNextLSN*

Data pages

each with a pageLSN

master record

LSN of most recent checkpoint

Xact Table

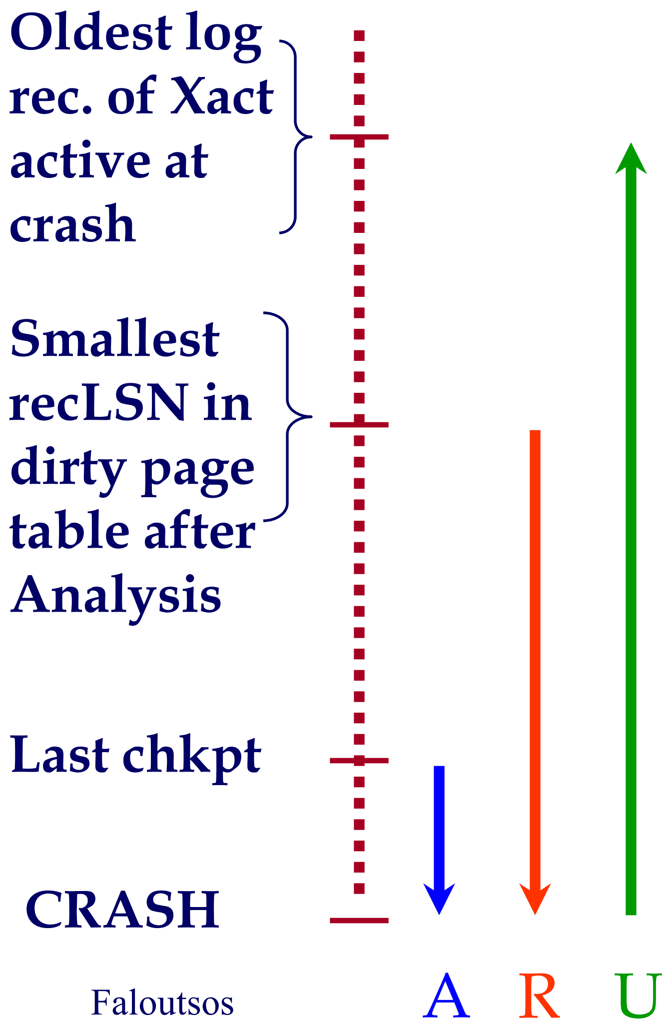
lastLSN
status

Dirty Page Table

recLSN

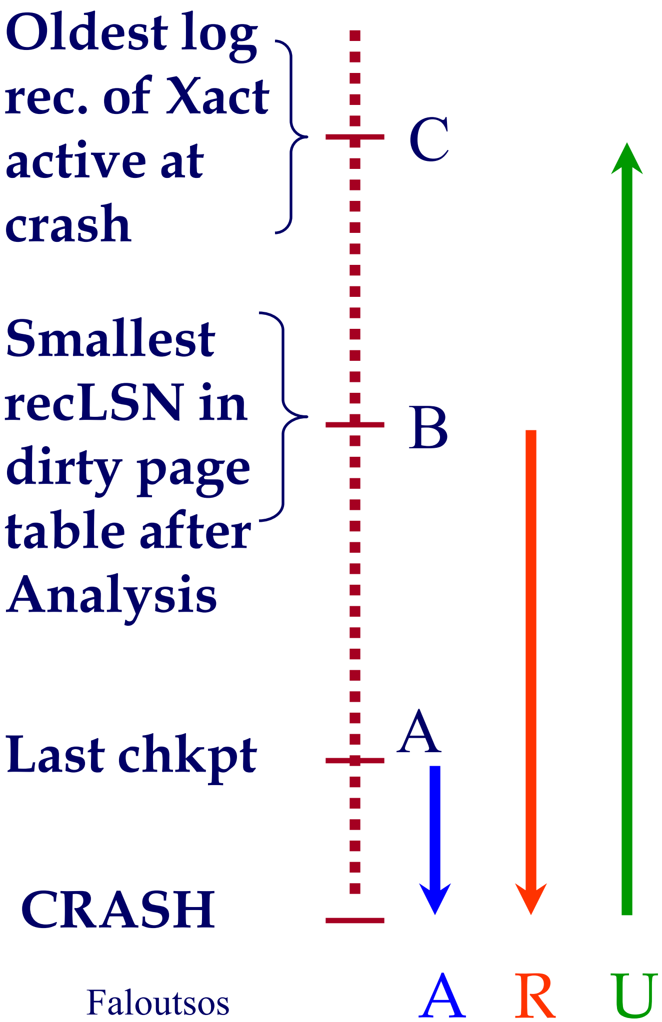
flushedLSN

Crash Recovery: Big Picture



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

Crash Recovery: Big Picture



- Notice: relative ordering of A, B, C may vary!

Recovery: The Analysis Phase

- Re-establish knowledge of state at checkpoint.
  - via transaction table and dirty page table stored in the checkpoint

Recovery: The Analysis Phase

- Scan log forward from checkpoint.
  - End record: Remove Xact from Xact table.
  - All Other records:
    - Add Xact to Xact table, with status 'U' (=candidate for undo)
    - set lastLSN=LSN,
    - on commit, change Xact status to 'C'.
  - also, for Update records: If page P not in Dirty Page Table,
    - add P to DPT, set its recLSN=LSN.

Recovery: The Analysis Phase

- 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: REDO

Goal: *repeat History* to reconstruct state at crash:

- Reapply *all* updates (even of aborted Xacts!), redo CLRs.
- (and try to avoid unnecessary reads and writes!)

Specifically:

- Scan forward from log rec containing smallest recLSN in DPT. **Q: why start here?**

Phase 2: REDO (cont'd)

- ...
- 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) ≥ LSN. (this last case requires I/O)

Phase 2: REDO (cont'd)

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

Phase 2: REDO (cont'd)

- ...
- at the end of REDO phase, write 'end' log records for all xacts with status 'C',
- and remove them from transaction table

Phase 3: UNDO

Goal: Undo all transactions that were active at the time of crash ('loser xacts')

- That is, all xacts with 'U' status on the xact table of the Analysis phase
- Process them in reverse LSN order
- using the lastLSN's to speed up traversal
- and issuing CLRs

Phase 3: UNDO

ToUndo={lastLSNs of ‘loser’ Xacts}

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.

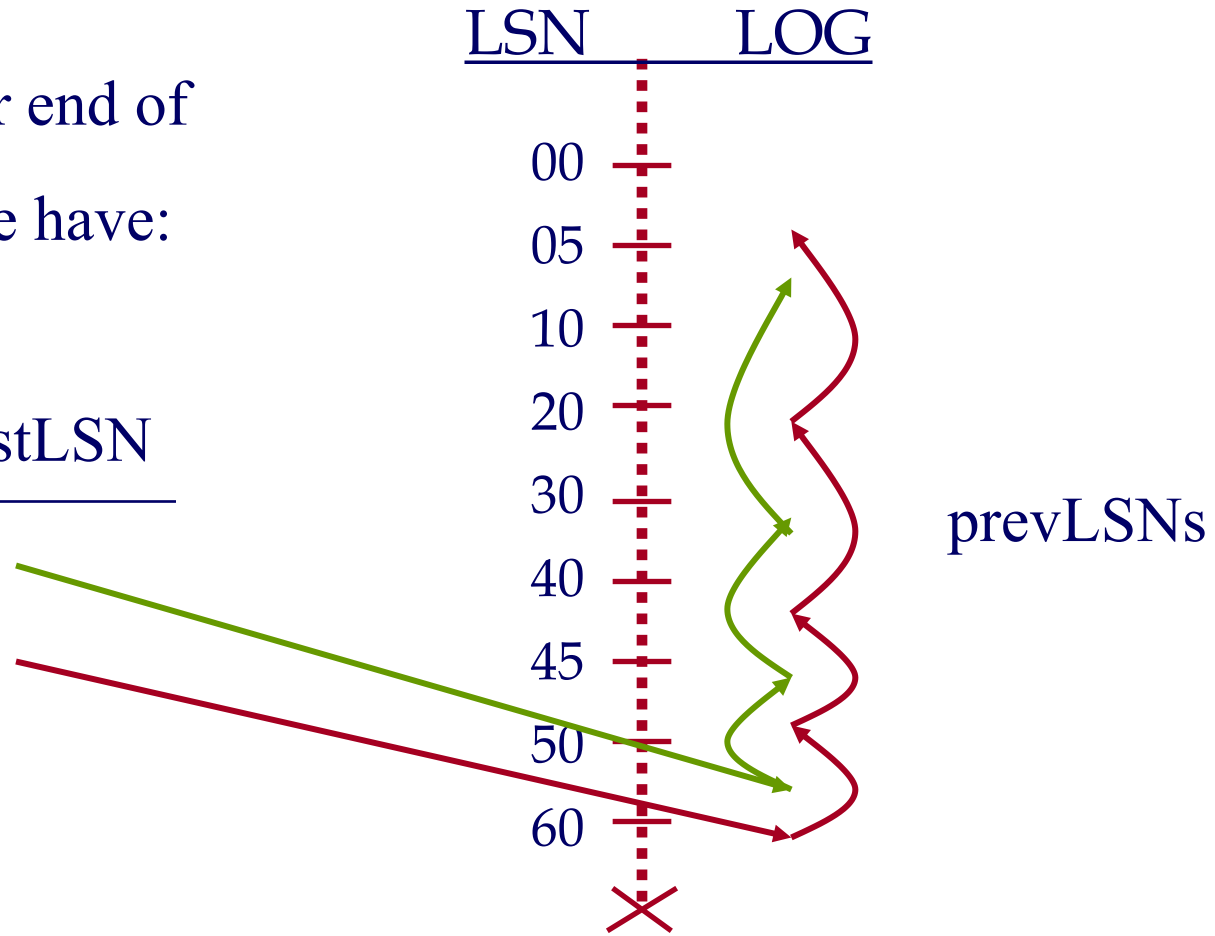
Until ToUndo is empty.

Phase 3: UNDO - illustration

suppose that after end of analysis phase we have:

xact table

xid	status	lastLSN
T32	U	
T41	U	



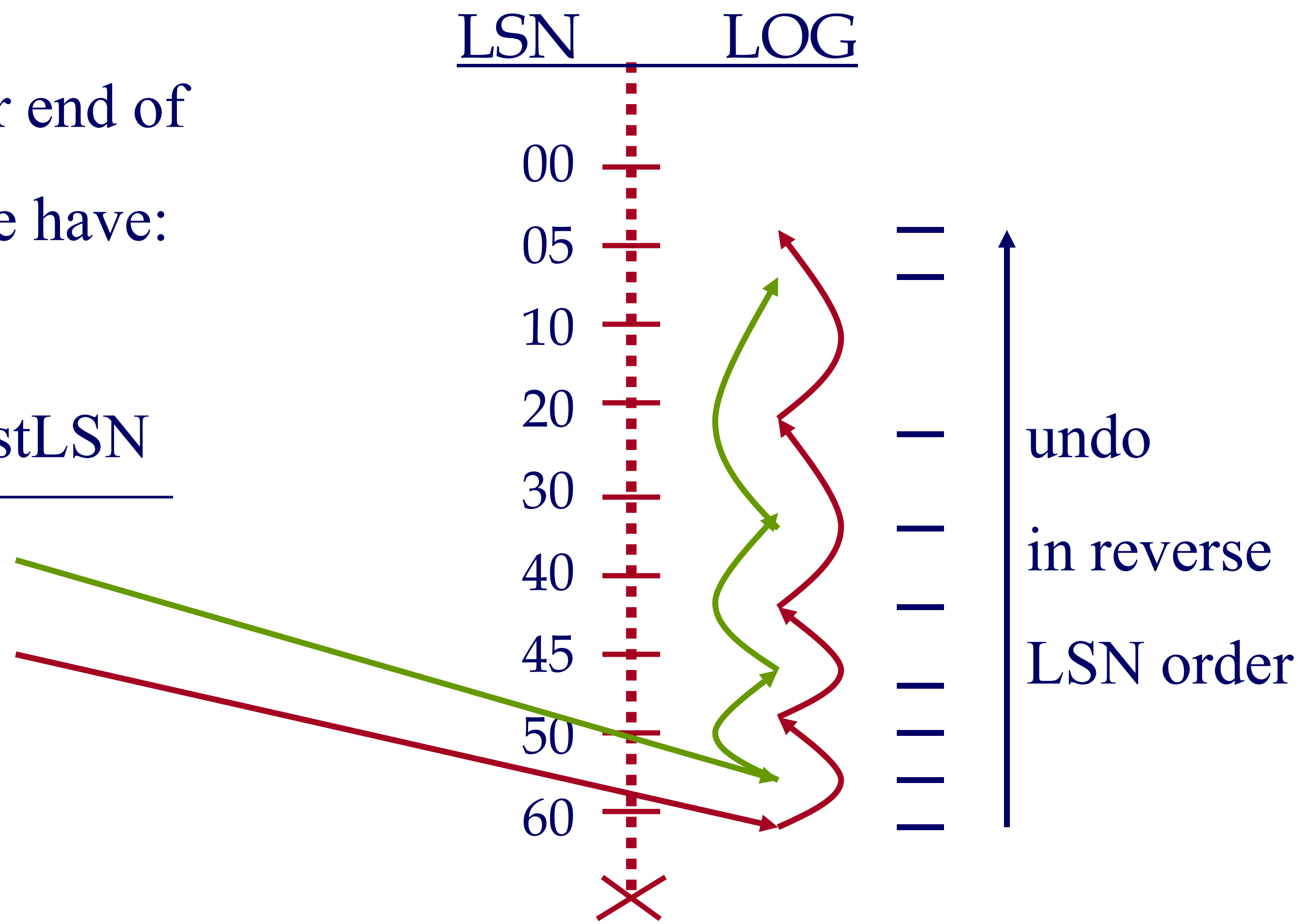
Phase 3: UNDO - illustration

suppose that after end of

analysis phase we have:

xact table

xid	status	lastLSN
T32	U	
T41	U	



Example of Recovery

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

prevLSNs

Questions

- Q1: After the Analysis phase, which are the 'loser' transactions?
- Q2: UNDO phase - what will it do?

Questions

- Q1: After the Analysis phase, which are the 'loser' transactions?
- A1: T2 and T3
- Q2: UNDO phase - what will it do?
- A2: undo ops of LSN 60, 50, 20

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

Example: Crash During Restart!

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

undonextLSN

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

undonextLSN

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

undonextLSN

Questions

- Q3: After the Analysis phase, which are the 'loser' transactions?
- Q4: UNDO phase - what will it do?

Questions

- Q3: After the Analysis phase, which are the 'loser' transactions?
- A3: T2 only
- Q4: UNDO phase - what will it do?
- A4: follow the string of *prevLSN* of T2, exploiting *undoNextLSN*

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

undonextLSN

Questions

- Q5: show the log, after the recovery is finished:

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, 95	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.
- How do you limit the amount of work in UNDO?
  - Avoid long-running Xacts.

Summary of Logging/Recovery

- Recovery Manager guarantees Atomicity & Durability.

Atomicity

Consistency

Isolation

Durability

Summary of Logging/Recovery

ARIES - main ideas:

- WAL (write ahead log), STEAL/NO-FORCE
- fuzzy checkpoints (snapshot of dirty page ids)

} let OS do its best

- redo *everything* since the earliest dirty page; undo 'loser' transactions
- write CLRs when undoing, to survive failures during restarts

} idempotency

Summary of Logging/Recovery

Additional concepts:

- LSNs identify log records; linked into backwards chains per transaction (via prevLSN).
- pageLSN allows comparison of data page and log records.
- (and several other subtle concepts: undoNextLSN, recLSN etc)