CS150A Database

Wenjie Wang

School of Information Science and Technology

ShanghaiTech University

Dec. 2, 2024

Today:

- Recovery :
  - Steal/No-force & WAL
  - Logging
  - Crash Recovery

Readings:

- Database Management Systems (DBMS), Chapter 20

Acknowledgement: Joe Hellerstein@UCBerkeley’s scourse notes

Review: Lock manager

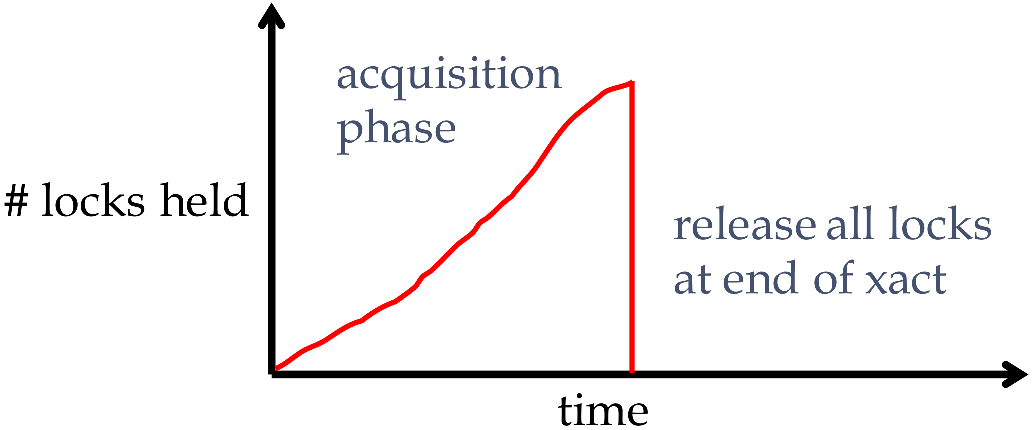
- Rules:
  - Xact must obtain a S (shared) lock before reading, and an X (exclusive) lock before writing.
  - **Xact cannot get new locks after releasing any locks**

Lock Compatibility Matrix

	S	X
S	√	–
X	–	–

Review: Strict 2PL

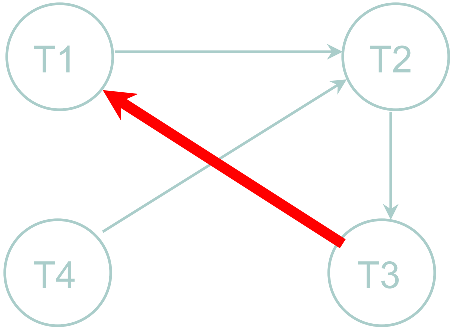
- Same as 2PL, except all locks released together when transaction completes
  - (i.e.) either
    - Transaction has committed (all writes durable), OR
    - Transaction has aborted (all writes have been undone)



Review: Deadlock

Example:

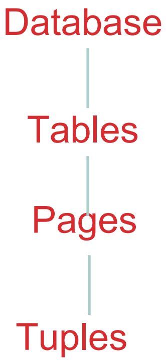
T1:	**S(A)**	**S(D)**		**S(B)**					
T2:			**X(B)**				**X(C)**		
T3:					**S(D)**	**S(C)**			**X(A)**
T4:								**X(B)**	



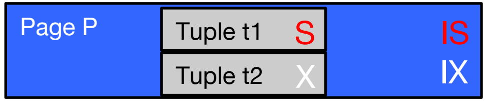
Review : Lock Compatibility Matrix

- IS – Intent to get S lock(s) at finer granularity.
- IX – Intent to get X lock(s) at finer granularity.
- SIX mode: Like S & IX at the same time.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false



Handy simple case to remember:
Could 2 intent locks be compatible?



Review: The ACID properties

- **Atomicity:** All actions in the Xact happen, or none happen.
- **Consistency**: If the DB starts consistent before the Xact…
  it ends up consistent after.
- **Isolation**: Execution of one Xact is isolated from that of other Xacts.
- **Durability:** If a Xact commits, its effects persist.

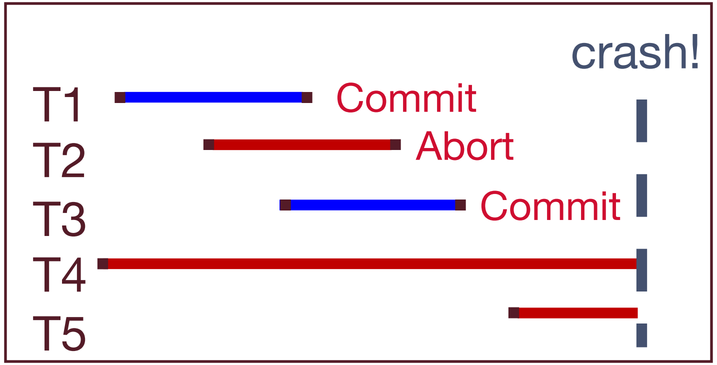
- Recovery Manager
  - Atomicity & Durability
  - Also to rollback transactions that violate Consistency

Motivation

- Atomicity:
  - Transactions may abort ("Rollback").
- Durability:
  - What if DBMS stops running?

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

- Questions:
  - Why do transactions abort?
  - Why do DBMSs stop running?



Atomicity: Why Do Transactions Abort?

- User/Application explicitly aborts
- Failed Consistency check
  - Integrity constraint violated
- Deadlock
- System failure prior to successful commit

Transactions and SQL

- You don’t need SQL to want transactions and vice versa
  - But they often go together
- SQL Basics
  - `BEGIN`
  - `COMMIT`
  - `ROLLBACK`

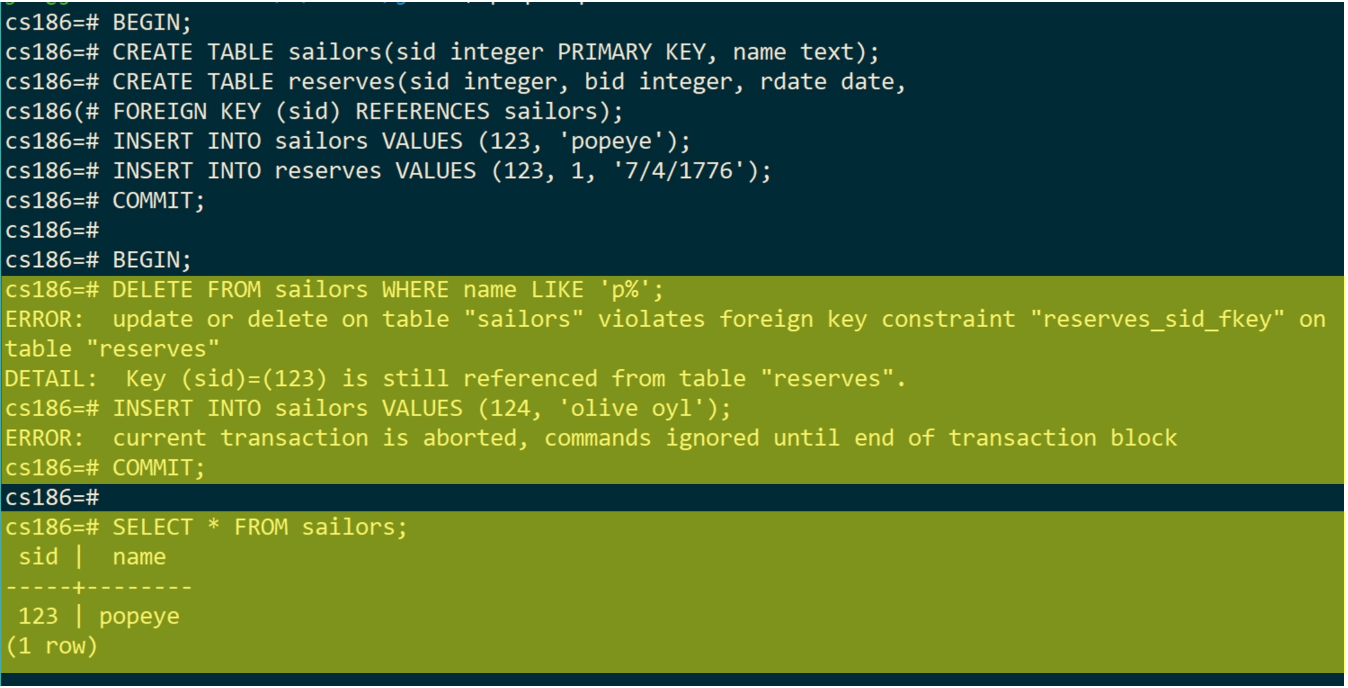
SQL Savepoints

- Savepoints
  - `SAVEPOINT <name>`
  - `RELEASE SAVEPOINT <name>`
    - Makes it as if the savepoint never existed
  - `ROLLBACK TO SAVEPOINT <name>`
    - Statements since the savepoint are rolled back

```
BEGIN;
          INSERT INTO table1 VALUES ('yes1');
          SAVEPOINT sp1;
          INSERT INTO table1 VALUES ('yes2');
          RELEASE SAVEPOINT sp1;
          SAVEPOINT sp2;
          INSERT INTO table1 VALUES ('no');
          ROLLBACK TO SAVEPOINT sp2;
          INSERT INTO table1 VALUES ('yes3');
COMMIT;
```

Example of SQL Integrity Constraints

- Constraint violation rolls back transaction



```
cs186=# BEGIN;
cs186=# CREATE TABLE sailors(sid integer PRIMARY KEY, name text);
cs186=# CREATE TABLE reserves(sid integer, bid integer, rdate date,
cs186(# FOREIGN KEY (sid) REFERENCES sailors);
cs186=# INSERT INTO sailors VALUES (123, 'popeye');
cs186=# INSERT INTO reserves VALUES (123, 1, '7/4/1776');
cs186=# COMMIT;
cs186=#
cs186=# BEGIN;
cs186=# DELETE FROM sailors WHERE name LIKE 'p%';
ERROR:  update or delete on table "sailors" violates foreign key constraint "reserves_sid_fkey" on
table "reserves"
DETAIL:  Key (sid)=(123) is still referenced from table "reserves".
cs186=# INSERT INTO sailors VALUES (124, 'olive oyl');
ERROR:  current transaction is aborted, commands ignored until end of transaction block
cs186=# COMMIT;
cs186=#
cs186=# SELECT * FROM sailors;
 sid |  name
-----+--------
 123 | popeye
(1 row)
```

Durability: Why Do Databases Crash?

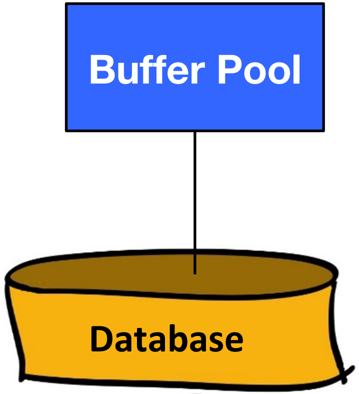
- Operator Error
  - Trip over the power cord
  - Type the wrong command
- Configuration Error
  - Insufficient resources: disk space
  - File permissions, etc.
- Software Failure
  - DBMS bugs, security flaws, OS bugs
- Hardware Failure
  - Media or Server

Assumptions for Our Recovery Discussion

- Concurrency control is in effect.
  - **Strict 2PL**, in particular.
- Updates are happening “in place”.
  - i.e. data is modified in buffer pool and pages in DB are overwritten
    - Transactions are not done on “private copies” of the data.

Exercise in Simplicity

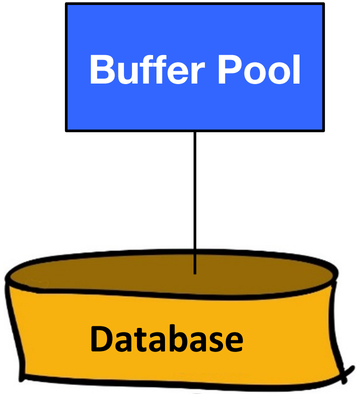
- Devise a simple scheme (requiring no logging) for Atomicity & Durability
- Questions:
  - What is happening during the transaction?
  - What happens at commit for Durability?
  - How do you rollback on abort?
  - How is Atomicity guaranteed?
  - Any limitations/assumptions?



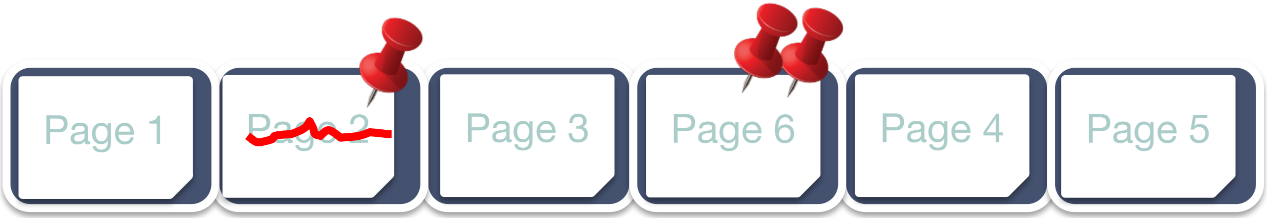
Exercise in Simplicity, cont

- Devise a simple scheme (requiring no logging) for Atomicity & Durability

- Example:
  1. Dirty buffer pages stay pinned in the buffer pool
     - Can't be "stolen" by replacement policy
     - Page-level locking to ensure 1 transaction per page
  2. At commit, we:
     a. Force dirty pages to disk
     b. Unpin those pages
     c. *Then* we commit
- Unfotunately, this doesn't work!



BufMgr State: Illustrated



FrameId	PageId	Dirty?	Pin Count
1	1	N	0
2	2	Y	1
3	3	N	0
4	6	N	2
5	4	N	0
6	5	N	0

Problems with Our Simplistic Solution

1. All dirty pages stay pinned in the buffer pool

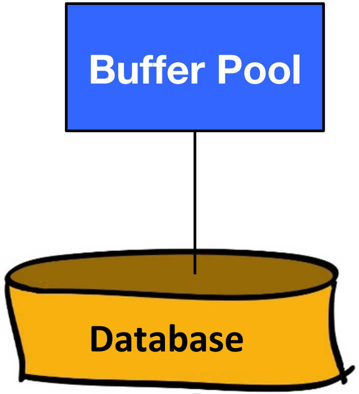
   What happens if buffer pool fills up?

   Not scalable!

2. At commit, we:
   a. Force dirty pages to disk
   b. Unpin those pages
   c. *Then* we commit

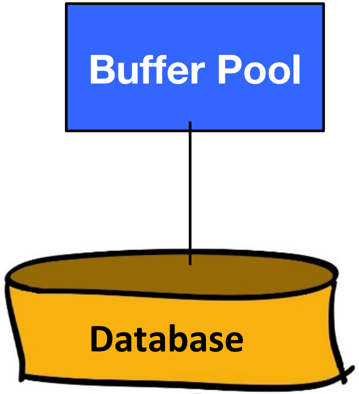
   What if DBMS crashes halfway through step a?

   Not atomic!



Buffer Management Plays a Key Role

- **NO STEAL policy** – don’t allow buffer-pool frames with uncommited updates to be replaced (or otherwise flushed to disk).
  - Useful for achieving atomicity without UNDO logging.
  - But can cause poor performance (pinned pages limit buffer replacement)
- **FORCE policy**: make sure every update is “forced” onto the DB disk before commit.
  - Provides durability without REDO logging.
  - But, can cause poor performance (lots of random I/O to commit)
- Our simple idea was NO STEAL/FORCE
  - And even that didn’t really achieve atomicity



Preferred Policy: Steal/No-Force

- Most complicated, but highest performance.
- **NO FORCE** (complicates enforcing Durability)
  - Problem: System crash before dirty buffer page of a committed transaction is flushed to DB disk.
  - Solution: Flush as little as possible, in a convenient place, prior to commit. Allows REDOing modifications.
- **STEAL** (complicates enforcing Atomicity)
  - What if a Xact that flushed updates to DB disk aborts?
  - What if system crashes before Xact is finished?
  - Must remember the old value of flushed pages
    - (to support UNDOing the write to those pages).

This is a dense slide … and the crux of the lecture.
Read it over carefully, and return to it later!

Buffer Management summary

	No Steal	Steal
No Force		Fastest
Force	Slowest	

Performance Implications

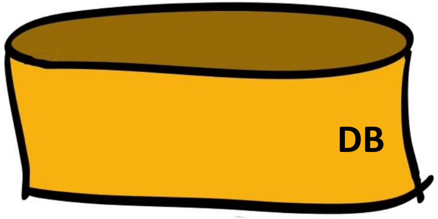
	No Steal	Steal
No Force	No UNDO REDO	UNDO REDO
Force	No UNDO No REDO	UNDO No REDO

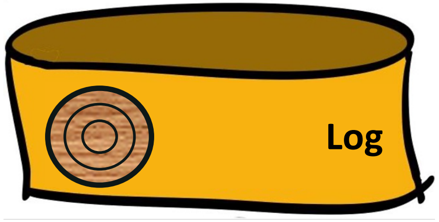
Logging/Recovery Implications

Steal/No-Steal: Whether Disk contains uncommitted data
Force/No-force: Whether disk contains committed data

Basic Idea: Logging

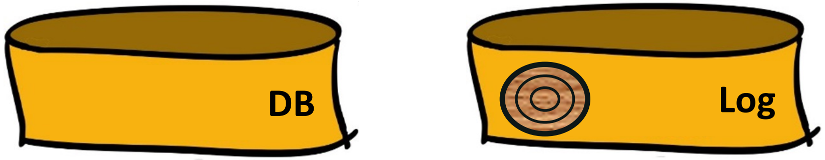
- For every update, record info to allow REDO/UNDO in a log.
  - Sequential writes to log (on a separate disk).
  - Minimal info written to log: pack multiple updates in a single log page.
- Log: An **ordered list** of log records to allow REDO/UNDO
  - Log record contains:
    - **<XID, pageID, offset, length, old data, new data>**
  - and additional control info (which we'll see soon).





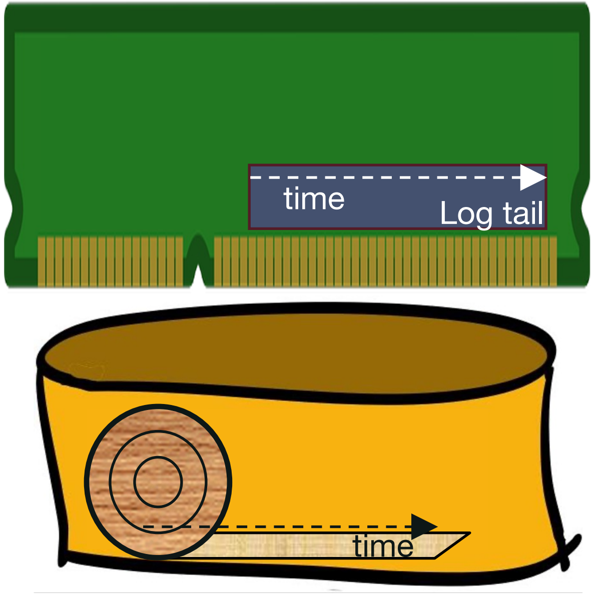
Write-Ahead Logging (WAL)

- The **Write-Ahead Logging Protocol**:
  1. Must **force** the **log record** for an update **<u>before</u>** the corresponding **data page** gets to the DB disk.
  2. Must **force all log records** for a Xact **<u>before commit</u>**.
     - I.e. transaction is not committed until
       all of its log records including its “commit” record are on the stable log.
- #1 (with **UNDO** info) helps guarantee Atomicity.
- #2 (with **REDO** info) helps guarantee Durability.
- This allows us to implement Steal/No-Force



WAL & the Log

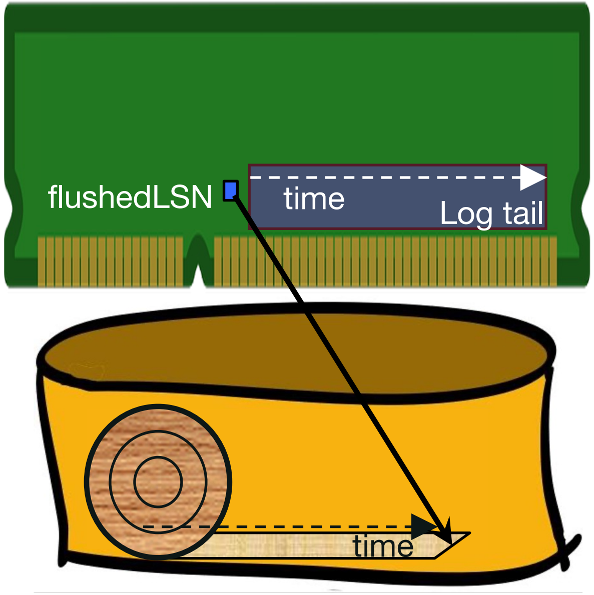
- Log: an ordered file, with a write buffer ("tail") in RAM.
- Each log record has a **Log Sequence Number** (LSN).
  - LSNs unique and increasing.



Log records flushed to disk

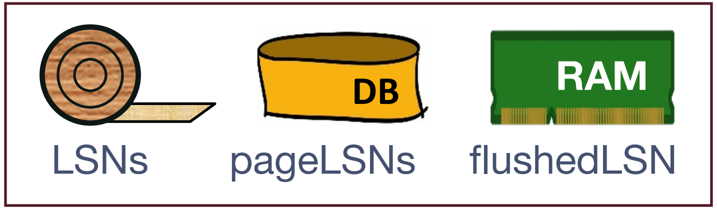
WAL & the Log, Pt 2

- Log: an ordered file, with a write buffer (“tail”) in RAM.
- Each log record has a **Log Sequence Number** (LSN).
  - LSNs unique and increasing.
  - **flushedLSN** tracked in RAM

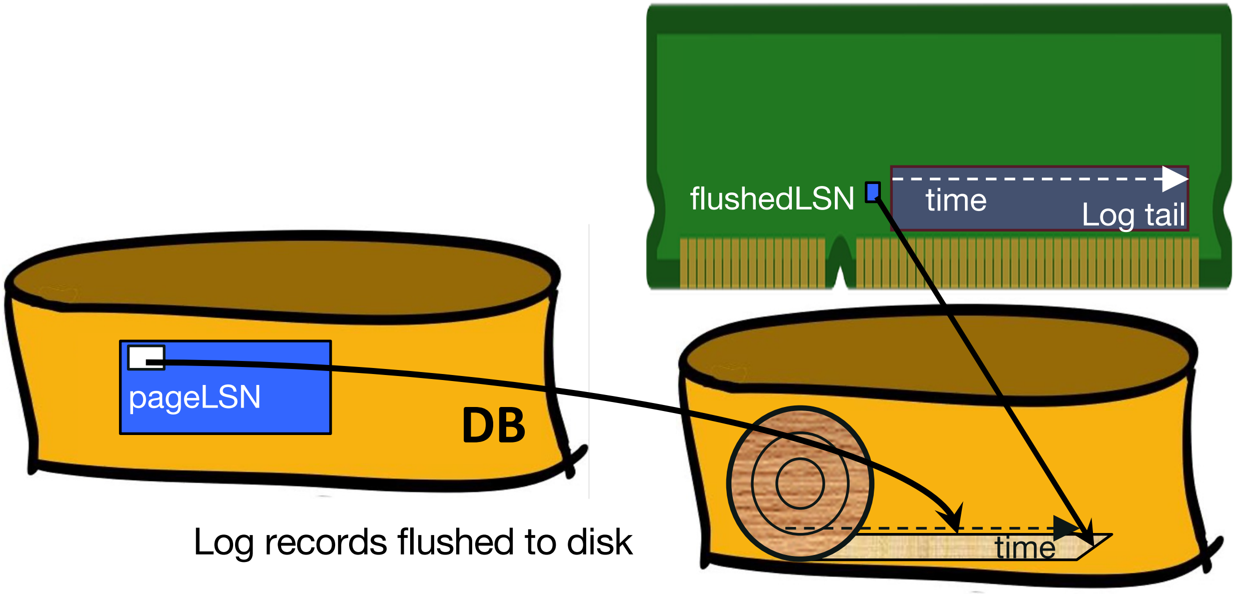


Log records flushed to disk

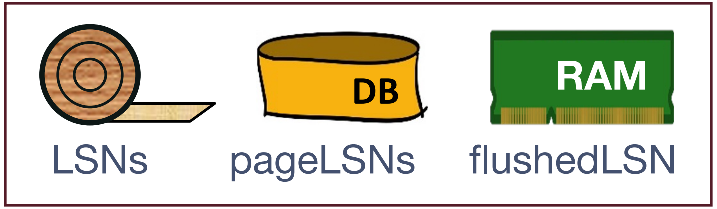
WAL & the Log, Pt 3



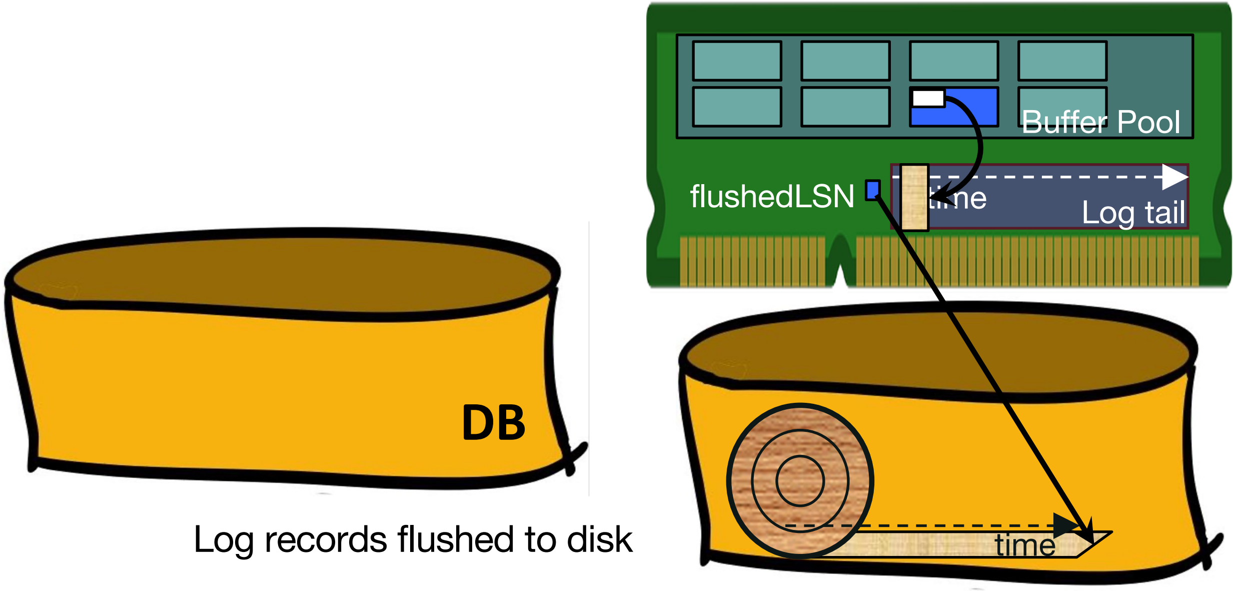
- Each **data page** in the DB contains a pageLSN.
  - A “pointer” into the log
  - The LSN of the most recent log record for an update to that page.



WAL & the Log, Pt 4

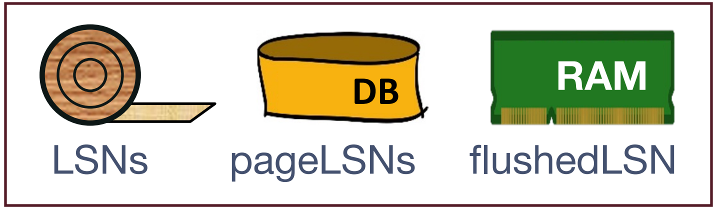


- WAL: Before page i is flushed to DB, log must satisfy:
  - **pageLSN$_i$ ≤ flushedLSN**

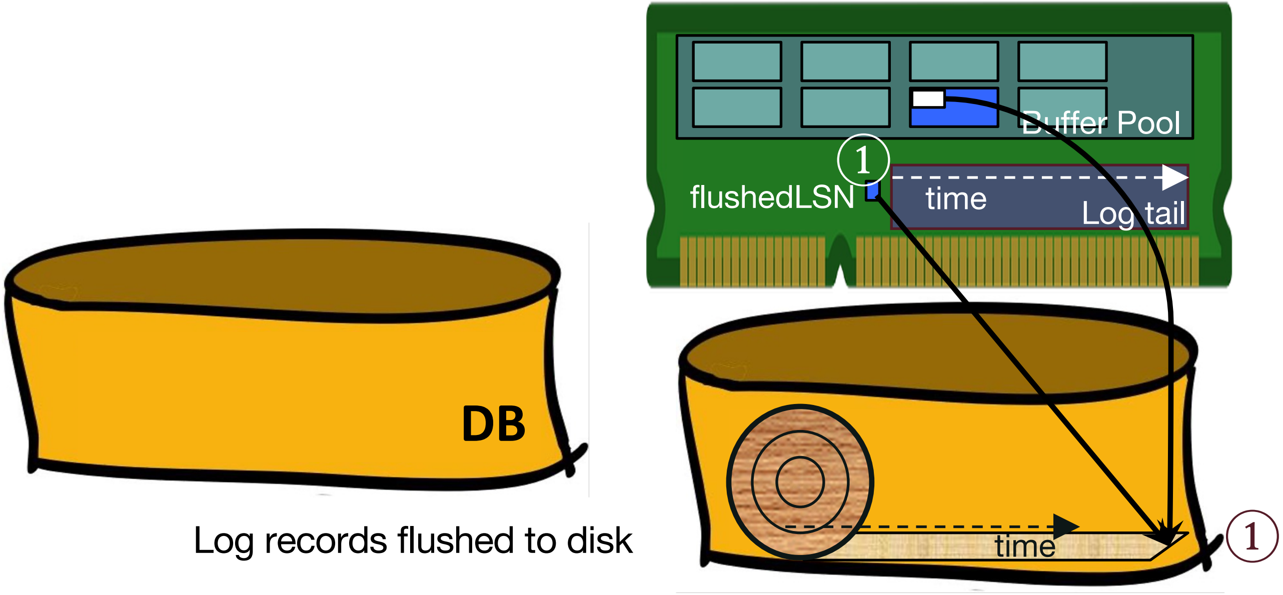


Log records flushed to disk

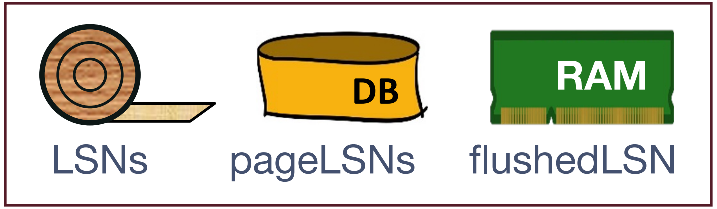
WAL & the Log, Pt 5



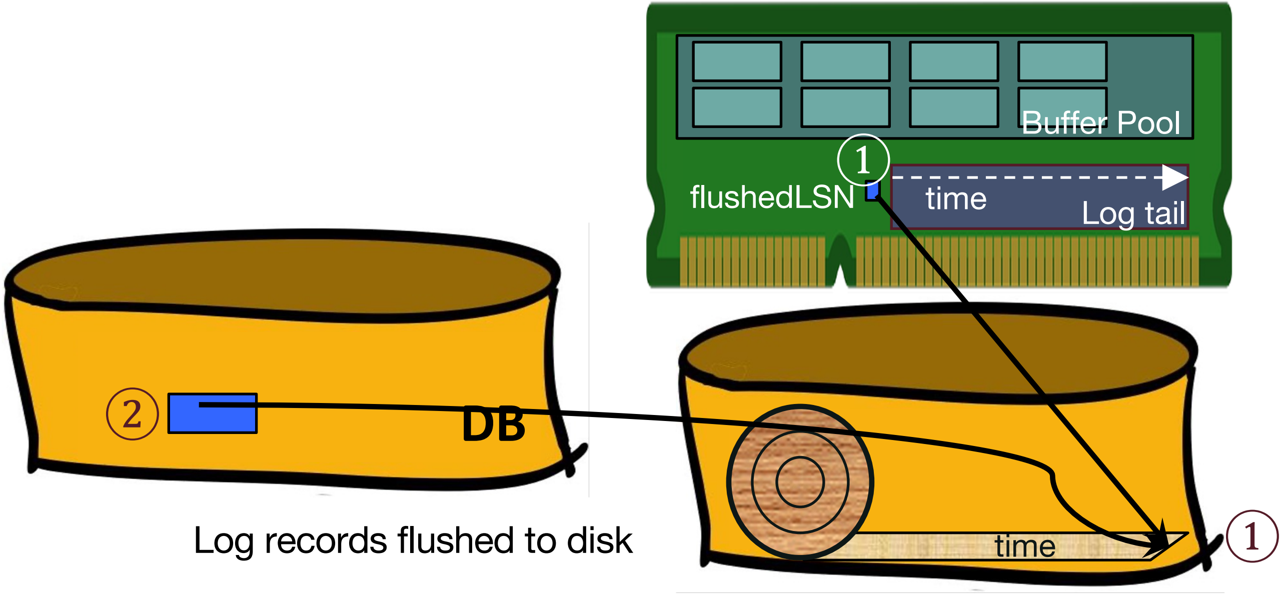
- WAL: Before page i is flushed to DB, log must satisfy:
  - **$\text{pageLSN}_i \le \text{flushedLSN}$**



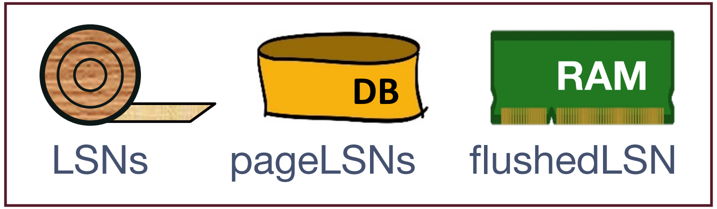
WAL & the Log, Pt 6



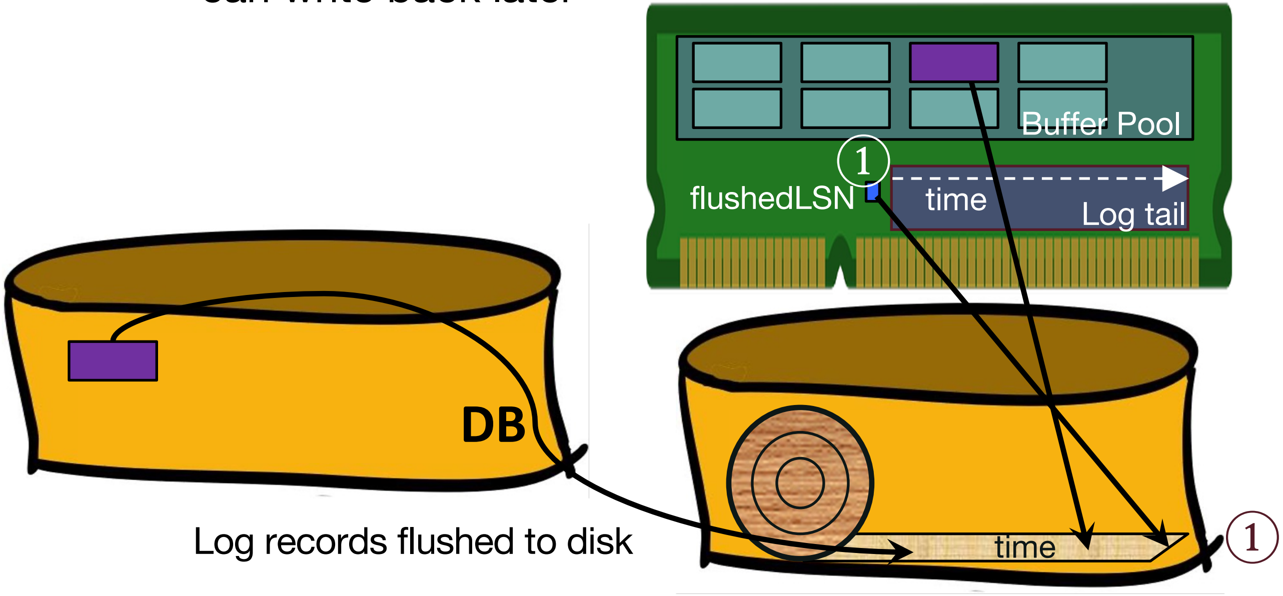
- WAL: Before page i is written to DB, log must satisfy:
  - **$\mathbf{pageLSN_i \le flushedLSN}$**



WAL & the Log, Pt 7

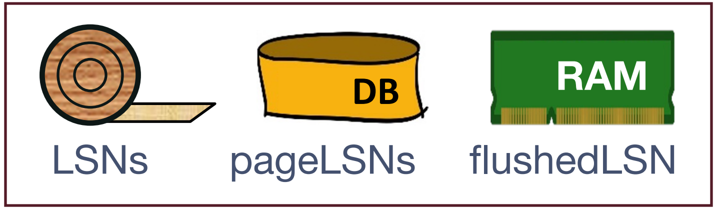


- WAL: Before page i is written to DB, log must satisfy:
  - **$\text{pageLSN}_i \leq \text{flushedLSN}$**
- Don’t need to steal buffer frame if page is hot
  - can write back later

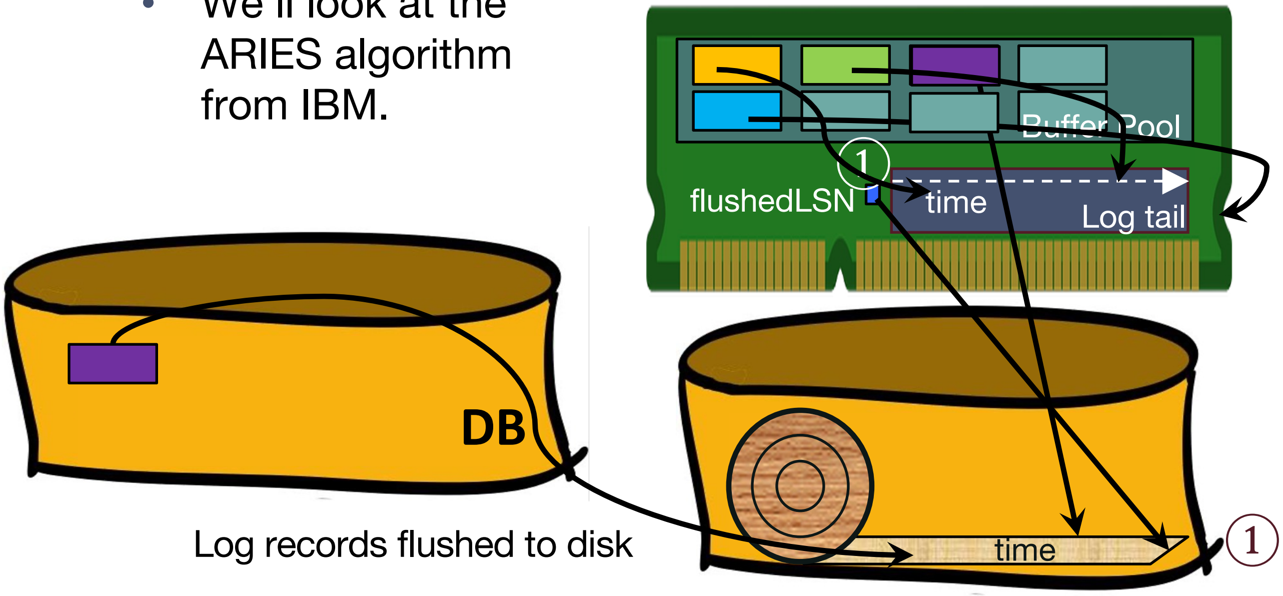


Log records flushed to disk

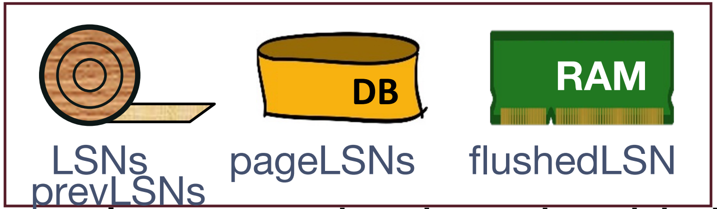
Summary



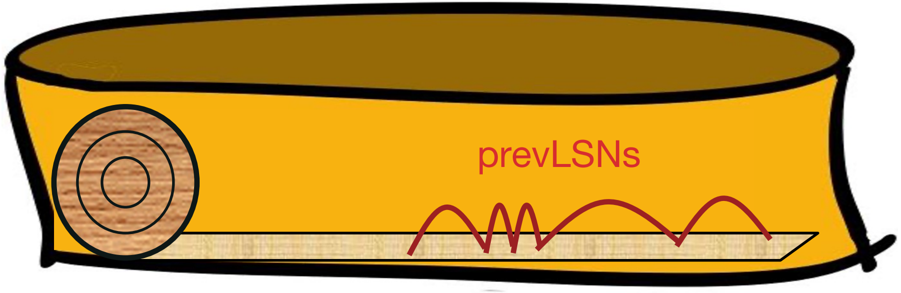
- WAL: Before page i is written to DB, log must satisfy:
  - **$\text{pageLSN}_i \le \text{flushedLSN}$**
- Exactly how is logging (and recovery!) done?
  - We’ll look at the ARIES algorithm from IBM.



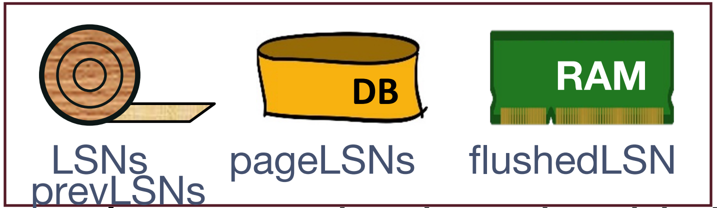
ARIES Log Records



- **prevLSN** is the LSN of the previous log record written by this **XID**
  - So records of an Xact form a linked list backwards in time



Log Records, Pt 2



- **prevLSN** is the LSN of the previous log record written by this **XID**
  - So records of an Xact form a linked list backwards in time

LogRecord fields:

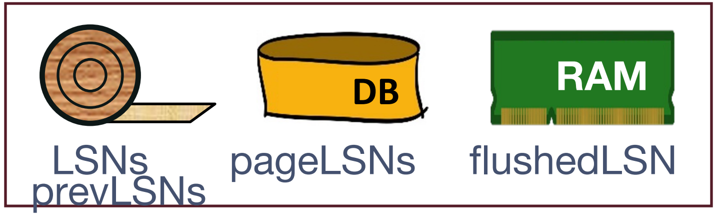
LSN
prevLSN
XID
type

update records only:

pageID
length
offset
before-image
after-image

- Possible log record types:
  - Update, Commit, Abort
  - Checkpoint (for log maintainence)
  - Compensation Log Records (CLRs)
    - (for UNDO actions)
  - End (end of commit or abort)

Log Records, Pt 3



- Update records contain sufficient information for **REDO and UNDO**
  - Our “physical diff” to the left works fine.
  - There are other encodings that can be more space-efficient

LogRecord fields:

LSN
prevLSN
XID
type

update records only:
pageID
length
offset
before-image
after-image

Other Log-Related State

- Two in-memory tables:
- Transaction Table
  - One entry per currently active Xact.
    - removed when Xact commits or aborts
  - Contains:
    - **XID**
    - **Status** (running, committing, aborting)
    - **lastLSN** (most recent LSN written by Xact).
- Dirty Page Table
  - One entry per dirty page currently in buffer pool.
  - Contains **recLSN**
    - LSN of the log record which first caused the page to be dirty.

Transaction Table

XID	**Status**	**lastLSN**
1	R	33
2	C	42

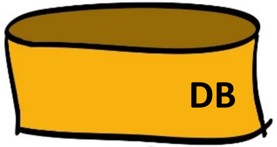
Dirty Page Table

PageID	**recLSN**
46	11
63	24

ARIES Big Picture: What's Stored Where

LogRecords

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



Data pages
each with a
pageLSN

Master record

Xact Table

- xid
- lastLSN
- status

Dirty Page Table

- pid
- recLSN

Log tail
flushedLSN

Buffer pool

LOGGING

Normal Execution of an Xact

- Series of **reads** & **writes**, followed by **commit** or **abort**.
  - For our discussion, the recovery manager sees page-level reads/writes

- STEAL, NO-FORCE buffer management, with Write-Ahead Logging.
  - Update, Commit, Abort log records written to log tail as we go
  - Transaction Table and Dirty Page Table being kept current
  - PageLSNs updated in buffer pool
  - Log tail flushed to disk periodically in background
    - And flushedLSN changed as needed
  - Buffer manager stealing pages subject to WAL

Transaction Commit

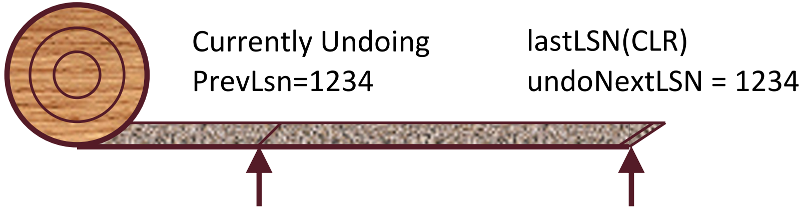
- Write **commit** record to log.
- All log records up to Xact's commit record are flushed to disk.
  - Guarantees that **flushedLSN ≥ lastLSN**.
  - Note that log flushes are sequential, synchronous writes to disk.
  - Many log records per log page.
- Commit() returns.
- Write end record to log.

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.
  - Write an **Abort** log record before starting to rollback operations
  - Can follow chain of log records backward via the prevLSN field.
  - Write a “**CLR**” (compensation log record) for each undone operation.

Note: CLRs are a different type of log record we glossed over before

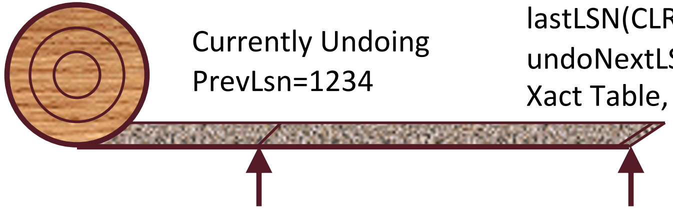
Abort, cont.



- 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
  - CLR contains REDO info
  - CLRs **never** Undone
    - Undo needn't be idempotent (>1 UNDO won't happen)
    - But they might be Redone when repeating history
      - (=1 UNDO guaranteed)
- At end of all UNDOs, write an "end" log record.

Idempotent: can be applied multiple times without changing the result beyond the initial application

Checkpointing



Currently Undoing
PrevLsn=1234

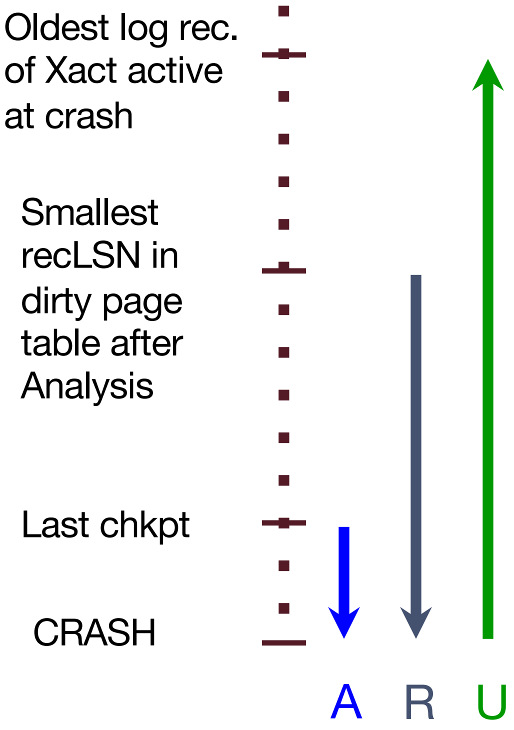
lastLSN(CLR)
undoNextLSN = 1234
Xact Table, DPT

- Conceptually, keep log around for all time.
  - Performance/implementation problems…
- Periodically, the DBMS creates a **<u>checkpoint</u>**
  - Minimizes recovery time after crash. Write to log:
    - **begin_checkpoint** record: Indicates when chkpt began.
    - **end_checkpoint** record: Contains current Xact table DPT
    - . A "**fuzzy checkpoint**": Other Xacts continue to run;
      - So all we know is that these tables are after the time of the begin_checkpoint record.
    - Store LSN of most recent chkpt record in a safe place
    - (**master record**, often block 0 of the log file).

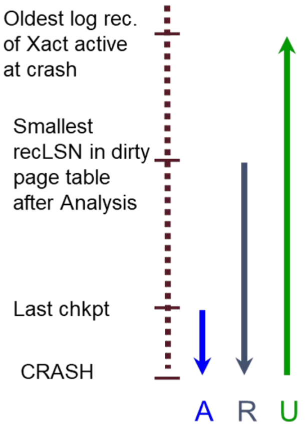
CRASH RECOVERY

Crash Recovery: Big Picture

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

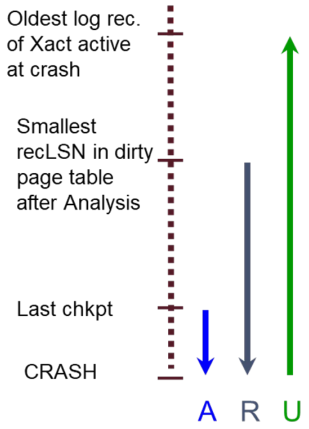

Oldest log rec. of Xact active at crash
Smallest recLSN in dirty page table after Analysis
Last chkpt
CRASH
A
R
U

Recovery: The Analysis Phase


Oldest log rec. of Xact active at crash
Smallest recLSN in dirty page table after Analysis
Last chkpt
CRASH
A
R
U

- Re-establish knowledge of state at checkpoint.
  - via transaction table and dirty page table stored in the checkpoint
- Scan log forward from checkpoint.
  - **End** record:
    - Remove Xact from Xact table
  - **Update** record:
    - If page P not in Dirty Page Table, Add P to DPT, set its **recLSN=LSN.**
  - !**End** record:
    - Add Xact to Xact table
    - set lastLSN=LSN
    - change Xact status on commit.
- At end of Analysis…
  - For any Xacts in the Xact table in Committing state,:
    - Write a corresponding END log record
    - …and Remove Xact from Xact table.
  - Now, Xact table says which xacts were active at time of crash.
    - Change status of running xacts to aborting and write abort records
  - DPT says which dirty pages might not have made it to disk

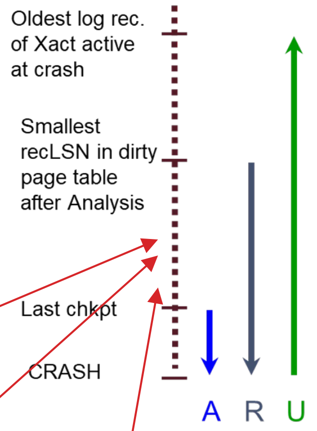
Phase 2: The REDO Phase


Oldest log rec. of Xact active at crash
Smallest recLSN in dirty page table after Analysis
Last chkpt
CRASH
A
R
U

- 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 DPT.
  - Q: why start here?

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

Scenarios When We Do Not REDO

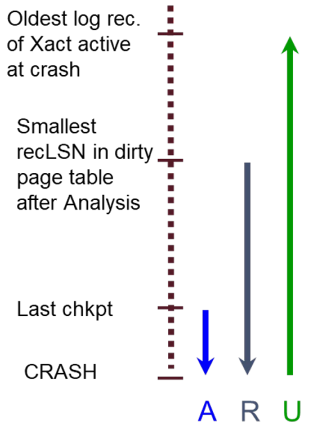


Oldest log rec. of Xact active at crash
Smallest recLSN in dirty page table after Analysis
Last chkpt
CRASH
A
R
U

Given an update log record…

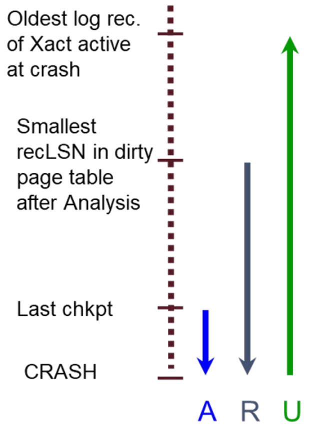
- Affected page is not in the Dirty Page Table. *How did that happen?*
  - This page was flushed to DB, removed from DPT before checkpoint
  - *Then* DPT flushed to checkpoint
- Affected page is in DPT, but has DPT recLSN > LSN. *H.D.T.H.?*
  - This page was flushed to DB, removed from DPT before checkpoint
  - *Then* this page was referenced *again* and reinserted in DPT with larger recLSN
- pageLSN (in DB) >= LSN. (this last case requires DB I/O). *H.D.T.H.?*
  - This page was updated again and flushed to DB after this log record

Phase 3: The UNDO Phase


Oldest log rec. of Xact active at crash
Smallest recLSN in dirty page table after Analysis
Last chkpt
CRASH
A
R
U

- A simple solution:
  - The xacts in the Xact Table are losers.
  - For each loser, perform simple transaction abort (start or continue xact rollback)
  - Problem?
    - Lots of random I/O in the log following undoNextLSN chains.
    - Can we do this in one backwards pass of log?
      - Next slide!

Phase 3: The UNDO Phase, cont


Oldest log rec. of Xact active at crash
Smallest recLSN in dirty page table after Analysis
Last chkpt
CRASH
A
R
U

```
toUndo = {lastLSNs of all Xacts in the Xact Table}
while !toUndo.empty():
        thisLR = toUndo.find_and_remove_largest_LSN()
        if thisLR.type == CLR:
                if thisLR.undoNextLSN != NULL:
                        toUndo.insert(thisLR.undonextLSN)
                else: // thisLR.undonextLSN == NULL
                        write an End record for thisLR.xid in the log
        else:
                write a CLR for the undo in the log
                undo the update in the database
                toUndo.insert(thisLR.prevLSN)
```

Example of Recovery

RAM

Xact Table

lastLSN

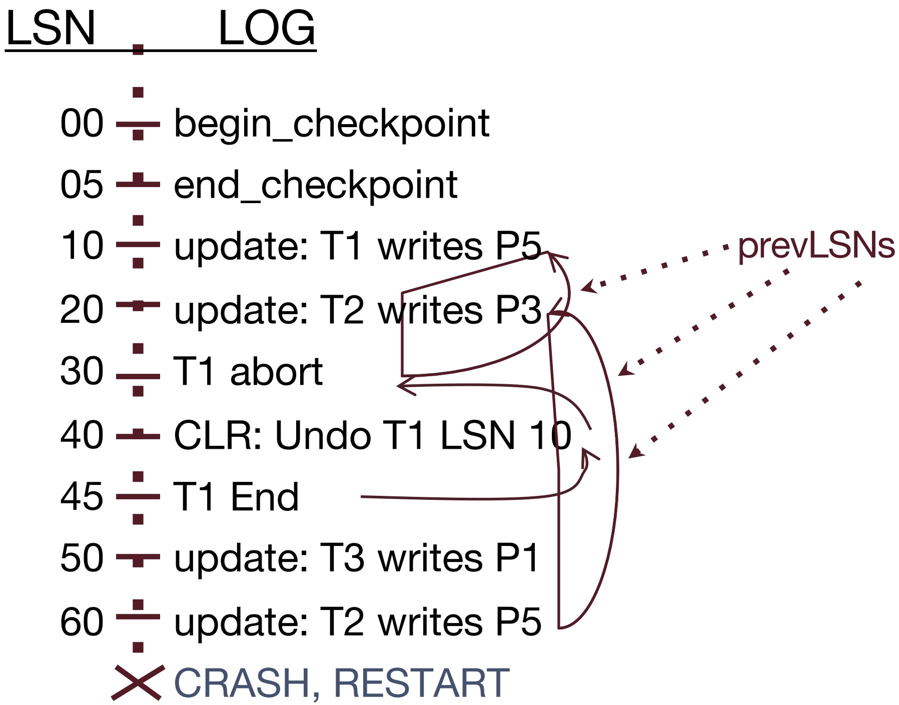
status

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

Using pencil and paper, run the ARIES recovery algorithm on this log, assuming you have access to a master record pointing to LSN 05. Maintain all the state on the left as you go!

Example: Crash During Restart!

RAM

Xact Table

lastLSN

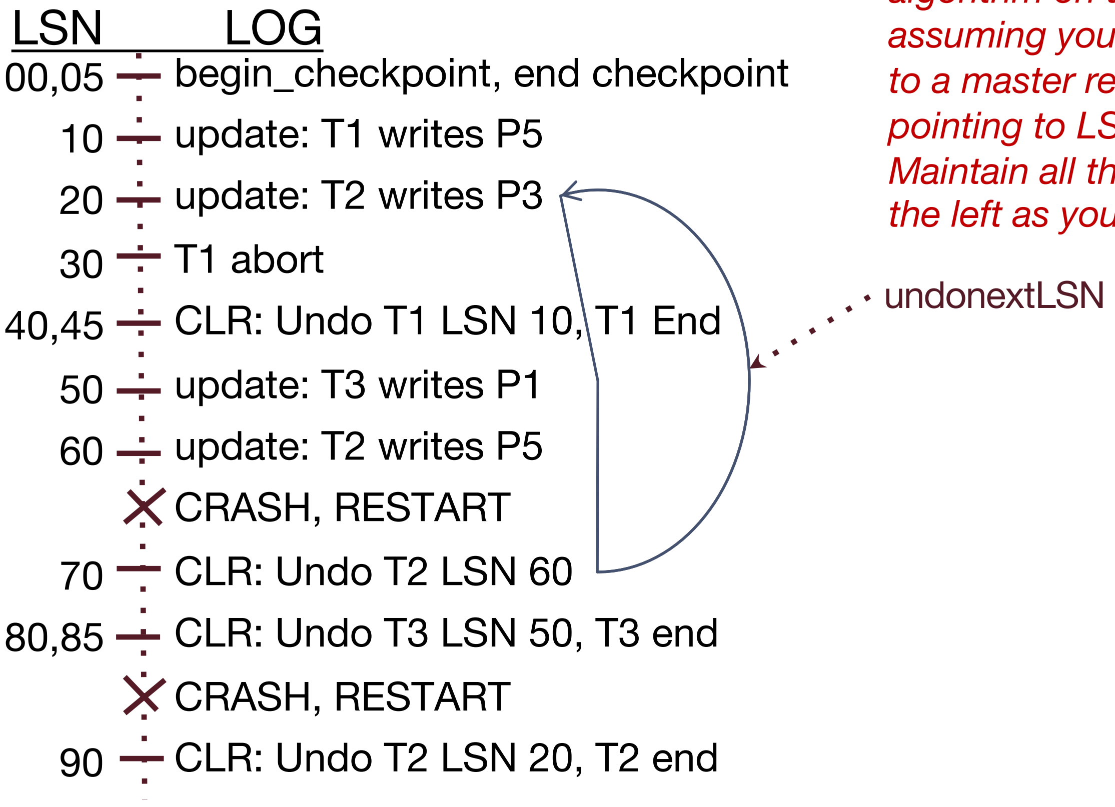
status

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

Using pencil and paper, run the ARIES recovery algorithm on this log, assuming you have access to a master record pointing to LSN 05. Maintain all the state on the left as you go!

Additional Crash FAQs to Understand

Q: What happens if system crashes during Analysis?

A: Nothing serious. RAM state lost, need to start over next time.

Q: What happens if the system crashes during REDO?

A: Nothing bad. Some REDOs done, and we'll detect that next time.

Q: How do you limit the amount of work in REDO?

A: Flush asynchronously in the background. Even"hot" pages!

Q: How do you limit the amount of work in UNDO?

A: 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**: 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 (running, aborting) after Redo.
- Upon Undo, write CLRs.
- Redo “repeats history”: Simplifies the logic!