Recovery

R&G - Chapter 20

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.

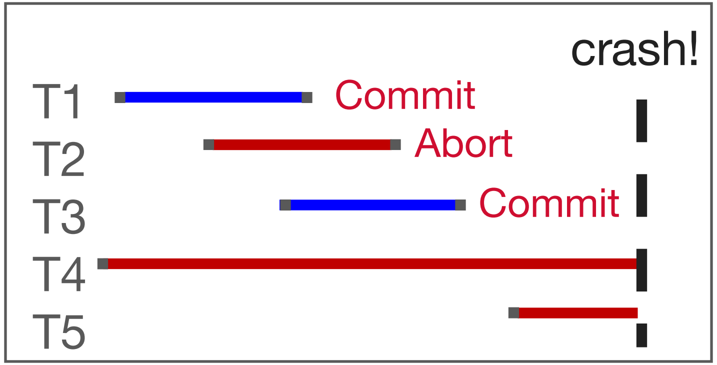
- Recovery Manager
  - Atomicity & Durability
  - Also to rollback transactions that violate Consistency

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?

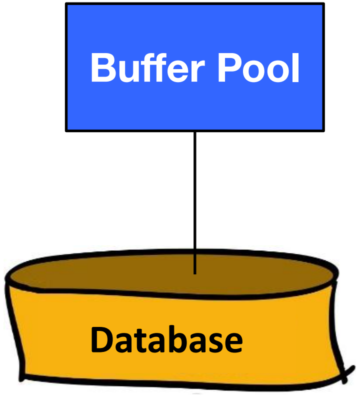
- Operator Error
  - Trip over the power cord
  - Type the wrong command
- Configuration Error
  - Insufficient resources: disk space
  - File permissions, etc.
- Software Failure
  - DBMS bugs, security flaws, OS bugs
- Hardware Failure
  - Media or Server

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.

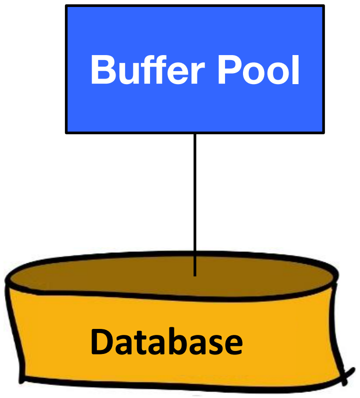
Exercise in Simplicity

- 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 <u>simple</u> 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
- Unfortunately, this doesn't work!



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:

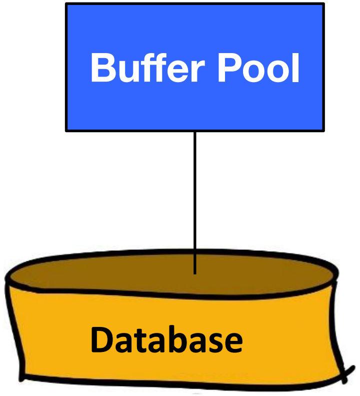
   a. Force dirty pages to disk

   b. Unpin those pages

   c. *Then* we commit

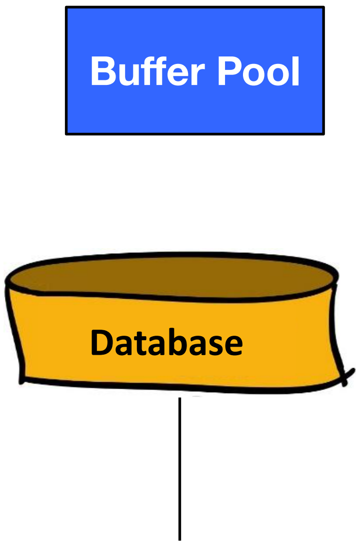
      What if DBMS crashes halfway through step a?

      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	

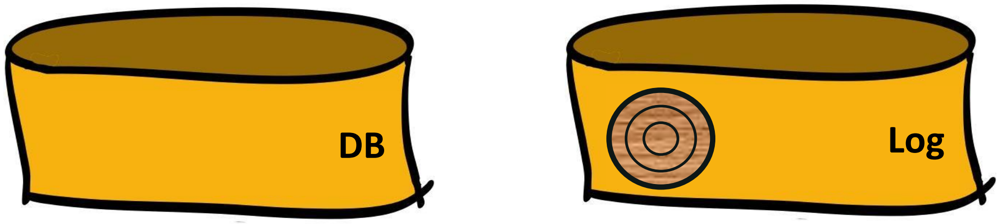
Performance Implications

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

Logging/Recovery Implications

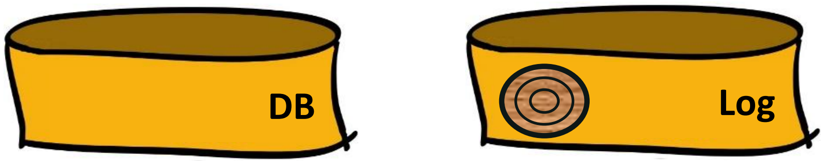
Basic Idea: Logging

- 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).



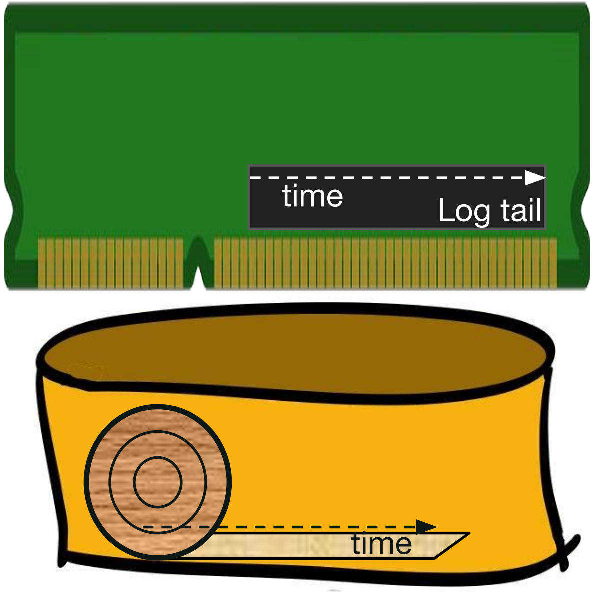
Write-Ahead Logging (WAL)

- 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

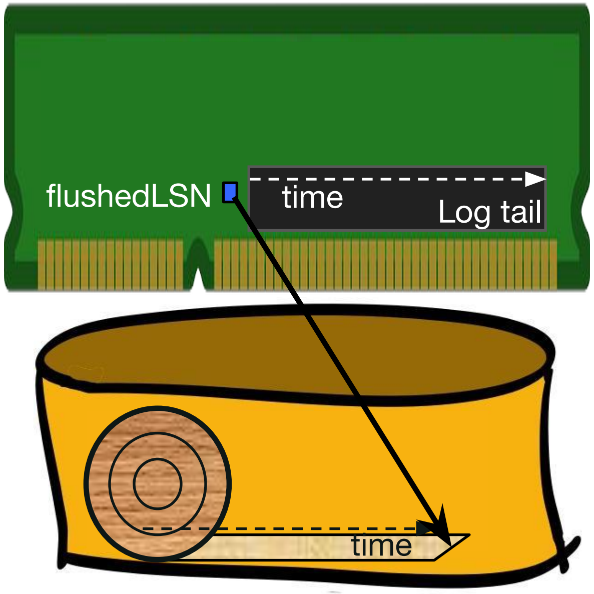
- Log: an ordered file, with a write buffer ("tail") in RAM.
- Each log record has a **Log Sequence Number** (LSN).
    - LSNs unique and increasing.



Log records flushed to disk

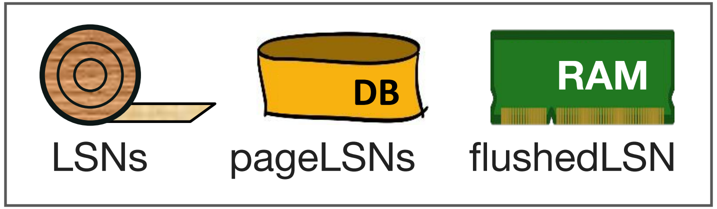
WAL & the Log, Pt 2

- 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

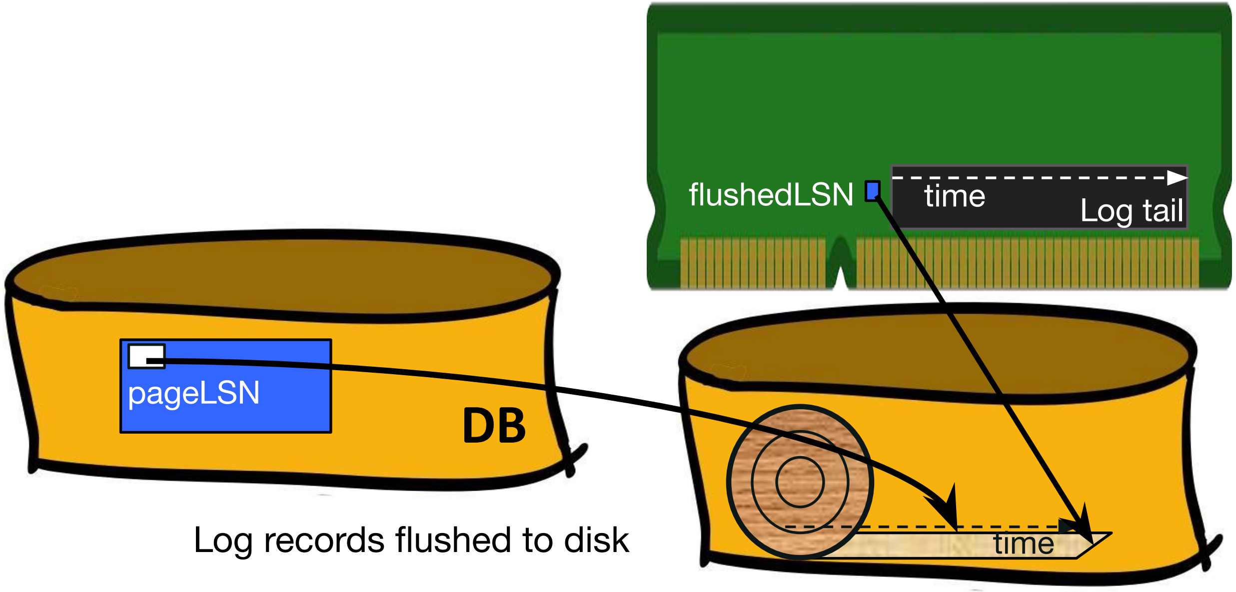


Log records flushed to disk

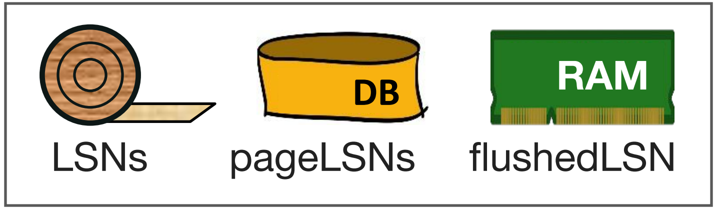
WAL & the Log, Pt 3



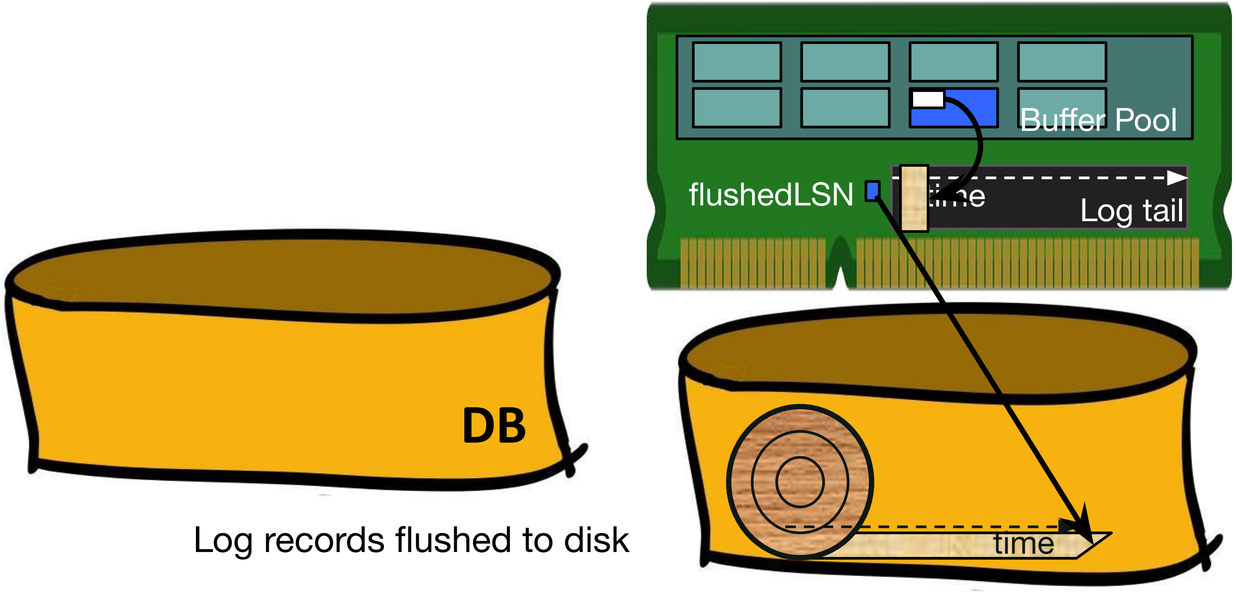
- Each **data page** in the DB contains a pageLSN.
  - A “pointer” into the log
  - The LSN of the most recent log record for an update to that page.



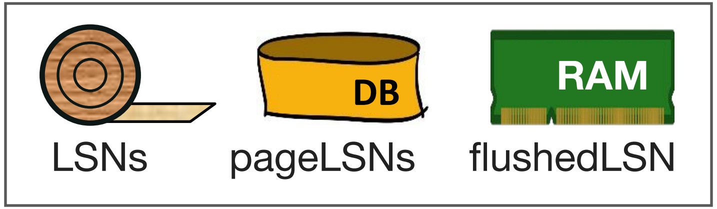
WAL & the Log, Pt 4



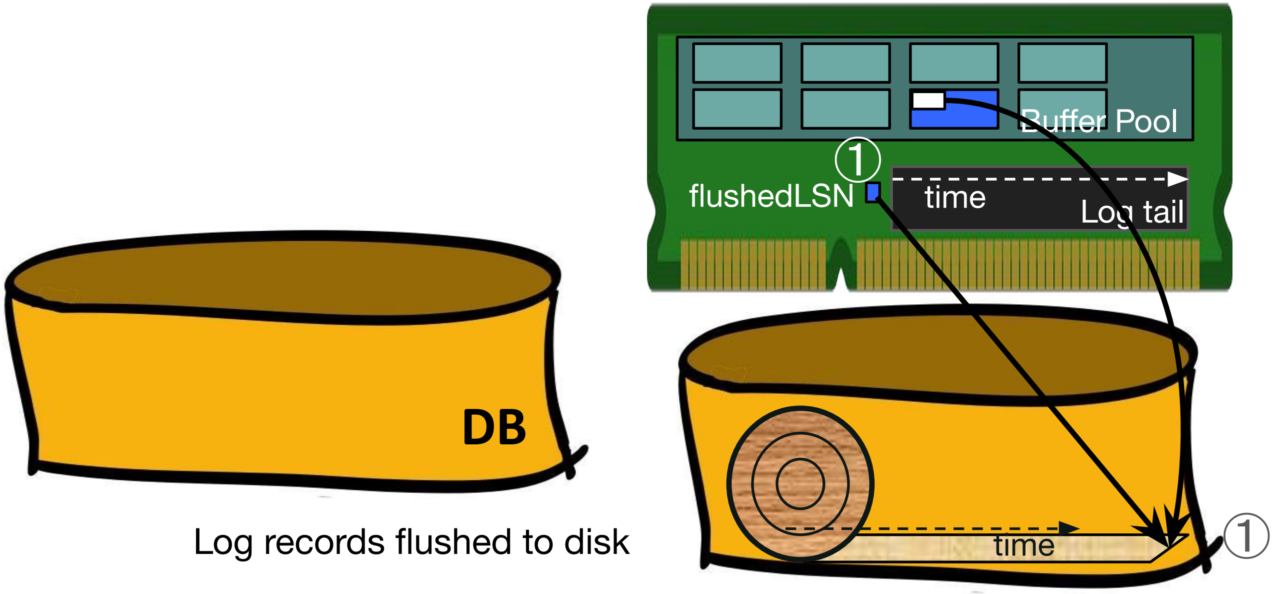
- WAL: Before page i is flushed to DB, log must satisfy:
  - **$\text{pageLSN}_i \le \text{flushedLSN}$**



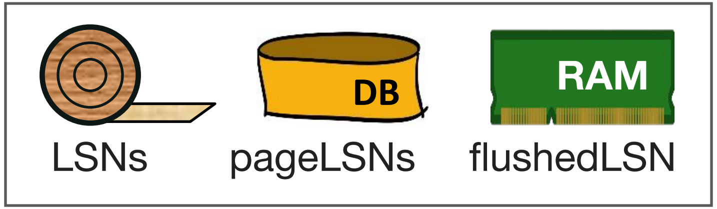
WAL & the Log, Pt 5



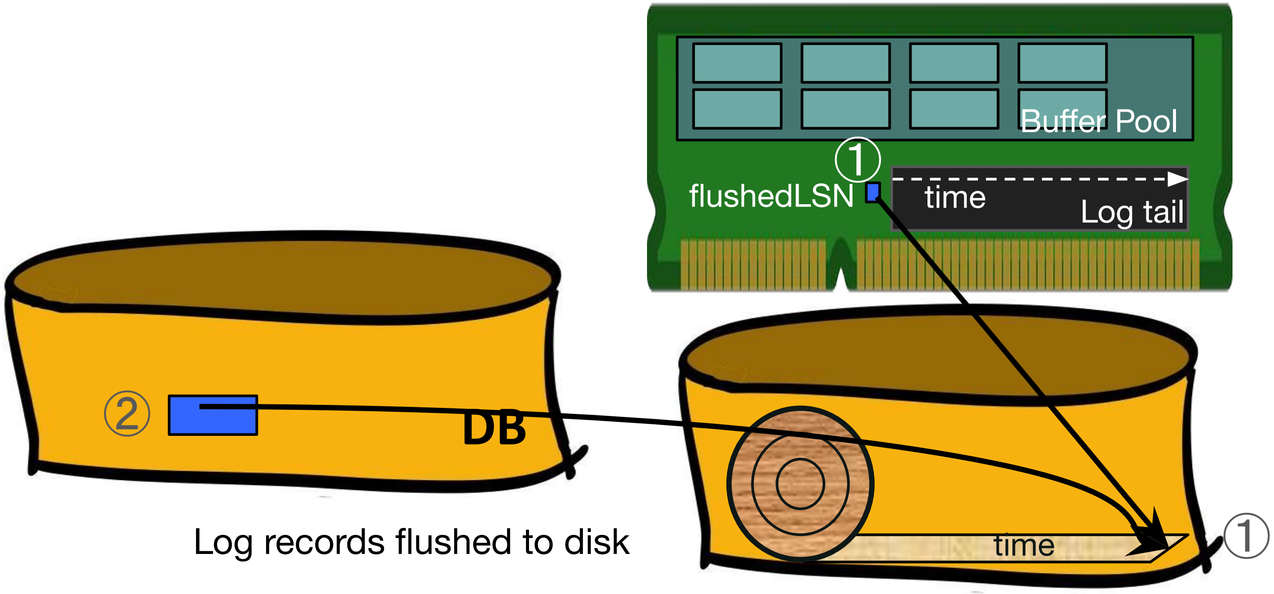
- WAL: Before page i is flushed to DB, log must satisfy:
  - **$pageLSN_i \le flushedLSN$**



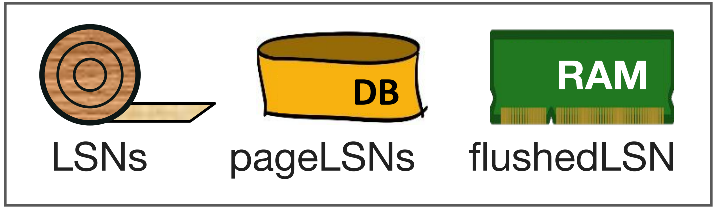
WAL & the Log, Pt 6



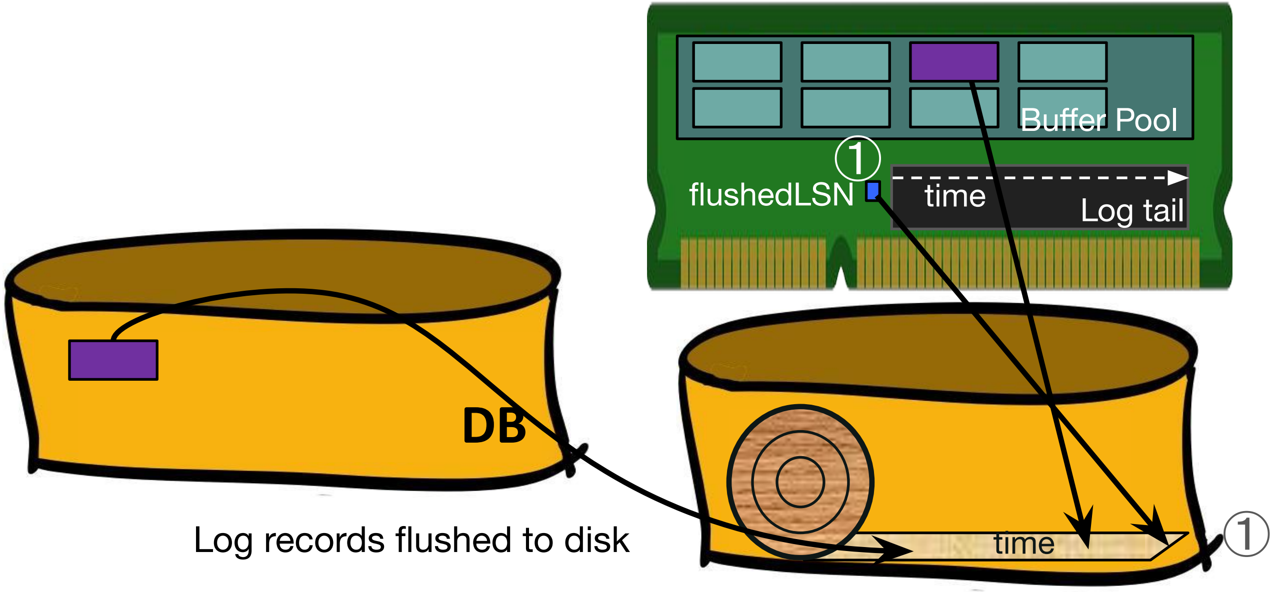
- WAL: Before page i is written to DB, log must satisfy:
  - **$pageLSN_i \le flushedLSN$**



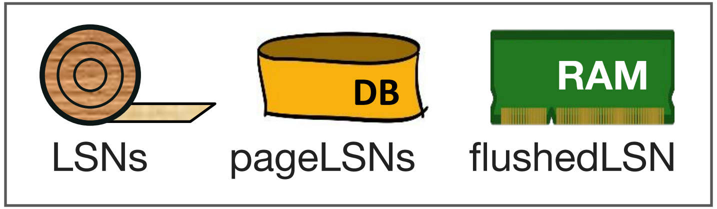
WAL & the Log, Pt 7



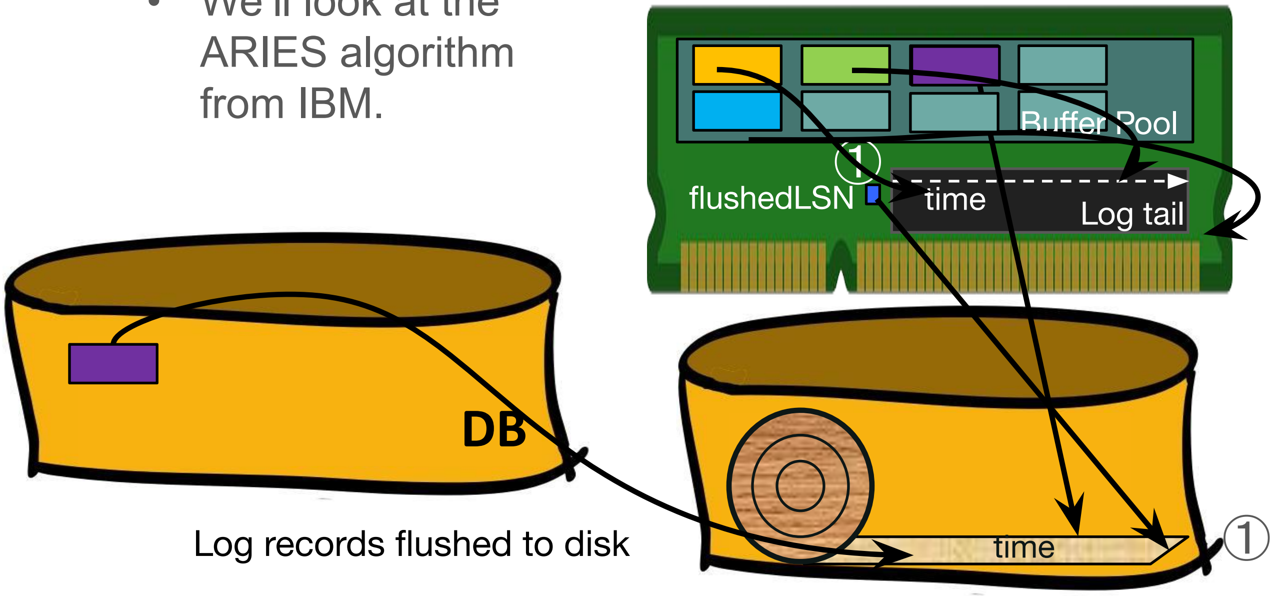
- WAL: Before page i is written to DB, log must satisfy:
  - **$pageLSN_i \leq flushedLSN$**
- Don't need to steal buffer frame if page is hot
  - can write back later



Summary



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



UNDO Logging

FORCE and STEAL

Undo Logging

Log records

- <START T>
  - transaction T has begun
- <COMMIT T>
  - T has committed
- <ABORT T>
  - T has aborted
- <T,X,v>
  - T has updated element X, and its *old* value was v

Undo Logging: Idea

- At recovery time, undo the transactions that have not been committed.
- Leave committed transactions alone.

Undo-Logging (Steal/Force) Rules

Allows STEAL

U1: If T modifies X, then <T,X,v> must be written to disk before the dirty page containing X

>> Want to record the old value before the new value replaces the old value permanently on disk.

U2: If T commits, then dirty pages must be written to disk before <COMMIT T>

>> Want to ensure that all changes written by T have been reflected before T is allowed to commit.

FORCE

- Hence: dirty data page writes are done *early*, before the tra commits

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Crash !

WHAT DO WE DO ?

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH(A)	16	16	16	16	8	Crash !
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

WHAT DO WE DO ?

We UNDO by setting B=8 and A=8

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

What do we do now ?

Crash !

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Crash !

What do we do now ?

Nothing: log contains COMMIT

Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**	**UNDO Log**
						<START T>
FETCH(A)		8			8	
READ(A,t)	8	8				
t:=t*2	16	8			8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

When must we force pages to disk ?

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

RULES: log entry *before* dirty pages *before* COMMIT

Recovery with Undo Log

…
…
<T6,X6,v6>
…
…
<START T5>
<START T4>
<T1,X1,v1>
<T5,X5,v5>
<T4,X4,v4>
<COMMIT T5>
<T3,X3,v3>
<T2,X2,v2>

Crash !

Question 1: Which updates are undone?

All uncommitted txns

Question 2: How far back do we need to read in the log?

Scan entire log for uncommitted txns

Question 3: What happens if there is a second crash, during recovery?

OK: undos are idempotent

However, perf implications fixed by ARIES

Recovery with Undo Log

After system crash, run recovery manager

1. Decide for each transaction T whether it is completed or not
   - <START T>….<COMMIT T>…. = yes
   - <START T>….<ABORT T>……. = yes
   - <START T>……………………… = no

2. Undo all modifications by incomplete transactions

Recovery with Undo Log

Recovery manager:

- Read log from the end; cases:
  - <COMMIT/ABORT T>: mark T as completed
  - <T,X,v>: if T is not completed
    then write X=v to disk
    else ignore /* *committed or aborted txn. */*
  - <START T>: ignore
- How far back do we need to go?
  - All the way to the start!
  - Fixed by checkpointing (later)

Recovery with Undo Log

…	
…	
<T6,X6,v6>	• Write v6 to X6 on disk
…	
…	
<START T5>	
<START T4>	
<T1,X1,v1>	• Write v1 to X1 on disk
<T5,X5,v5>	
<T4,X4,v4>	• Write v4 to X4 on disk
<COMMIT T5>	• Mark T5 as completed
<T3,X3,v3>	• Write v3 to X3 on disk
<T2,X2,v2>	• Write v2 to X2 on disk

REDO Logging

NO-FORCE and NO-STEAL

Redo Logging

Log records

- <START T>
  - transaction T has begun
- <COMMIT T>
  - T has committed
- <ABORT T>
  - T has aborted
- <T,X,v>
  - T has updated element X, and its *new* value was v

Redo Logging: Idea

- At recovery time, redo the txns that have been committed.
- Leave uncommitted txns alone.

Redo-Logging Rules

R1: If T modifies X, then both <T,X,v> and <COMMIT T> must be written to disk before dirty pages are written to disk

NO-STEAL

- Hence: dirty data page writes are done *late*, after commit
- This is WAL!

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

Crash !

How do we recover ?

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

Crash !

How do we recover ?

We REDO by setting A=16 and B=16

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

Crash !

How do we recover ?

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

Crash !

How do we recover ? Nothing to do!

Action	t	Mem A	Mem B		Disk B	**REDO** Log
						<START T>
READ(A,t)	8	8				
t:=t*2	16	8				
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

When must we force pages to disk ?

Action	t	Mem A	Mem B	Disk A	Disk B	**REDO** Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

NO-STEAL

RULE: dirty data writes *after* COMMIT

Recovery with Redo Log

After system crash, run recovery manager

1. Decide for each transaction T whether it is completed or not
   - <START T>….<COMMIT T>…. = yes
   - <START T>….<ABORT T>……. = yes
   - <START T>……………………… = no
2. Read log from the *beginning*, redo all updates of *committed* transactions

Again, this could be slow! Fix with checkpointing (later)

Recovery with Redo Log

Committed transactions: T2

<START T1>	
<T1,X1,v1>	Do Nothing
<START T2>	
<T2, X2, v2>	Write v2 to X2 on disk
<START T3>	
<T1,X3,v3>	Do Nothing
<COMMIT T2>	
<T3,X4,v4>	Do Nothing
<T1,X5,v5>	Do Nothing

Crash !

Comparison Undo/Redo

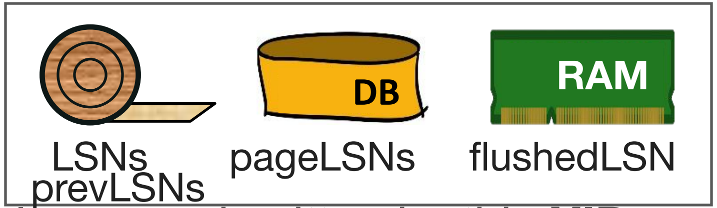
- Undo logging:
  - Dirty data page writes must be done early
  - If <COMMIT T> is seen, T definitely has written all its data to disk (hence, don’t need to undo)

- Redo logging
  - Dirty data page writes must be done late
  - If <COMMIT T> is not seen, T definitely has not written any of its data to disk (hence there is no dirty data on disk)

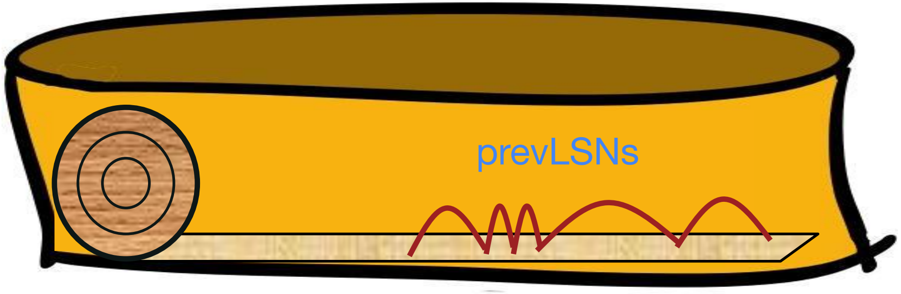
Pro/Con Comparison Undo/Redo

- Undo logging: (Steal/Force)
  - Pro: Less memory intensive: flush updated data pages as soon as log records are flushed, only then COMMIT.
  - Con: Higher latency: forcing all dirty buffer pages to be flushed prior to COMMIT can take a long time.
- Redo logging: (No Steal/No Force)
  - Con: More memory intensive: cannot flush data pages unless COMMIT log has been flushed.
  - Pro: Lower latency: don’t need to wait until data pages are flushed to COMMIT

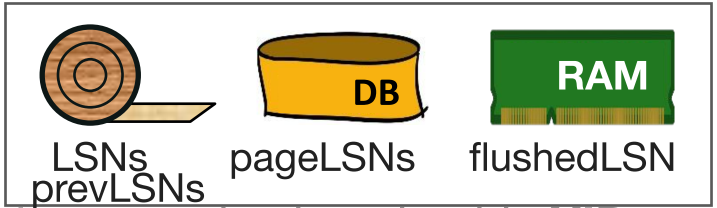
ARIES Log Records



- **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

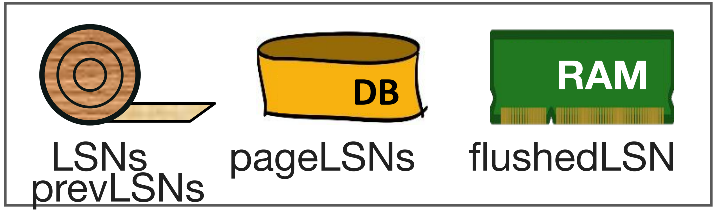
- Possible log record types:
  - Update, Commit, Abort
  - Checkpoint (for log maintainence)
  - Compensation Log Records (CLRs)
    - (for UNDO actions)
  - End (end of commit or abort)

LogRecord fields:

<u>LSN</u>
prevLSN
XID
type

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

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

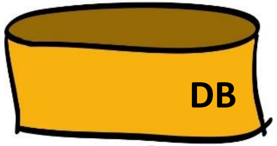
Dirty Page Table

PageID	recLSN
46	11
63	24

ARIES Big Picture: What's Stored Where

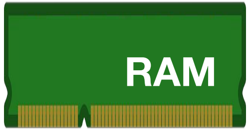
LogRecords

<u>LSN</u>
prevLSN
XID
type
pageID
length
offset
before-image
after-image



Data pages
each with a
pageLSN

Master record



Xact Table
<u>xid</u>
lastLSN
status

Dirty Page Table
<u>pid</u>
recLSN

Log tail
flushedLSN

Buffer pool

ARIES*

- ARIES pieces together several techniques into a comprehensive algorithm
- Developed at IBM Almaden, by C Mohan
- Several variations, e.g., for distributed transactions
- But first, we need to discuss the concept of *checkpoint*

*Algorithms for Recovery and Isolation Exploiting Semantics

Checkpoint

- Idea: save the state the database periodically so that we don't need to always process the entire log during recovery

- During a checkpoint:
  - Stop accepting new transactions
  - Wait until **all** current transactions complete (i.e., commit / abort)
  - Flush log to disk
  - Flush all dirty pages to disk
  - Write a <CKPT> log record, flush log again
  - At this point, changes by committed txns are written to disk, and aborted txns have been rolled back

  - Resume transactions

Undo Recovery with Checkpointing

During recovery,
Stop at first <CKPT>

…
…
<T9,X9,v9>
…
…
(all completed)
<CKPT>
<START T2>
<START T3>
<START T5>
<START T4>
<T1,X1,v1>
<T5,X5,v5>
<T4,X4,v4>
<COMMIT T5>
<T3,X3,v3>
<T2,X2,v2>

All txns here are completed
No need to recover
Can truncate this part of the log

Txns T2,T3,T4,T5 need to be recovered

Fuzzy Checkpointing

- Problem: database freezes during checkpoint
  - Not accepting any new transactions!
- Want DB to keep process txns during checkpoint
- Idea: *fuzzy* checkpointing
  - Save state of all txns and page statuses
    - Some txns can be running and dirty pages not flushed yet!
    - Need data structures to store such info

Fuzzy Checkpointing: Idea

- Keep track of:
  1. txn states (running, committing, etc) – this is the Xact Table
  2. dirty pages and which txn's action first caused page to become dirty – this is the Dirty Page Table

- Save 1 and 2 to disk during checkpoint
- At recovery:
  - Re-create 1 and 2 from the log
  - Re-create running txns and dirty pages in memory
  - Replay rest of the log (will see what this means)

Fuzzy Checkpointing: Data Structures

- Each log record has a **Log Sequence Number** (LSN)
  - A unique integer that's increasing (e.g., line number)
- Each data page has a **Page LSN**
  - The LSN of the most recent log record that updated that page.

Fuzzy Checkpointing: Data Structures

Dirty pages table

pageID	recLSN
P5	102
P6	103
P7	101

Transactions

txnID	lastLSN	Status
T100	104	commit
T200	103	abort

- Dirty Page Table
  - Lists all dirty pages
  - For each dirty page: recoveryLSN (recLSN) = first LSN that caused page to become dirty
- Transactions Table
  - Lists all txn's and their statuses
  - For each txn: lastLSN = its most recent update LSN (if active)

Fuzzy Checkpointing: Data Structures

Berkeley cs186

Log (WAL)

LSN	**prevLSN**	**txnID**	**pageID**	**Log Payload**
101	-	T100	-	START
102	-	T200	-	START
103	102	T200	P6	<old val, new val>
104	101	T100	P5	<old val, new val>

- Write ahead log
  - Store both old and new values in update records
  - New field **prevLSN** = LSN of the previous log record written by this txnID
  - Actions of a transaction form a linked list backwards in time

Fuzzy Checkpointing Example

Berkeley cs186

Dirty pages table (DPT)

pageID	recLSN
P5	102
P6	103
P7	101

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	P7	START
102	-	T200	P5	START
103	102	T200	P6	6, 21
104	101	T100	P5	39, 100

Transactions

txnID	lastLSN	Status
T100	104	commit
T200	103	abort

Buffer Pool

P5	P6	P7
PageLSN=104	PageLSN=103	PageLSN=101
…	…	…

Fuzzy Checkpointing: Protocol

Dirty pages table (DPT)

pageID	recLSN
P5	102
P6	103
P7	101

Transactions

txnID	lastLSN	Status
T100	104	commit
T200	103	abort

- Write a <BEGIN CKPT> to log
- Flush log to disk
- Continue normal operation
- When DPT and Transactions tables are written to the disk, write <END CKPT> to log
- Flush log to disk

ARIES Normal Operation

What to do when a transaction:

- Starts
- Updates a page
- Commits
- Aborts

- What to do when buffer manager:
  - FETCH a page from disk
  - FLUSH a page to disk

Transactions

txnID	lastLSN	Status
T100	101	running

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	P7	1, 5

Buffer Pool

P5	P6	P7
PageLSN=…	PageLSN=…	PageLSN=101

Dirty pages table

pageID	recLSN
P7	101

ARIES Normal Operation

Transaction starts

- Write START record in **Log**
- Update Transactions table

Transactions

txnID	lastLSN	Status
T100	101	running

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	P7	1, 5

Buffer Pool

P5 PageLSN=…	**P6** PageLSN=…	**P7** PageLSN=101

Dirty pages table

pageID	recLSN
P7	101

ARIES Normal Operation

<u>Transaction starts</u>

- Write START record in **Log**
- Update Transactions table

Ex: T105 starts

- Write **<START,T105>** in **Log**
- Add T105 in Transactions and set lastLSN = null

Transactions

txnID	lastLSN	Status
T100	101	running
T105	**-**	**running**

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	P7	1, 5
102	**-**	**T105**	**-**	**START**

Buffer Pool

P5 PageLSN=…	**P6** PageLSN=…	**P7** PageLSN=101

Dirty pages table

pageID	recLSN
P7	101

ARIES Normal Operation

Transaction updates

- Write update record in **Log**
- Update the following:
  - **prevLSN=lastLSN**
  - **pageLSN**=**LSN**
  - **lastLSN**=**LSN**
  - **recLSN**=if null then **LSN**

Transactions

txnID	lastLSN	Status
T100	101	running

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	P7	1, 5

Buffer Pool

P5	P6	P7
PageLSN=…	PageLSN=…	PageLSN=101

Dirty pages table

pageID	recLSN
P7	101

ARIES Normal Operation

Transaction updates

- Write update record in **Log**
- Update the following:
  - **prevLSN**=**lastLSN**
  - **pageLSN**=**LSN**
  - **lastLSN**=**LSN**
  - **recLSN**=if null then **LSN**

Ex: T100 writes 10 in P7

- Write **<T100,P7,5,10>** in the **Log**
  - New LSN: 102
- Update other tables (see arrows)

Transactions

txnID	lastLSN	Status
T100	102	running

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	P7	1, 5
102	101	T100	P7	5, 10

Buffer Pool

P5 PageLSN=…	**P6** PageLSN=…	**P7** PageLSN=102

Dirty pages table

pageID	recLSN
P7	101

ARIES Normal Operation

Page flushes

- Flush log up to (and incl.) **pageLSN**
- Remove page from **Dirty Pages Table** and **Buffer Pool**

Ex: Buffer manager wants to FLUSH(P7)

- Flush **Log** up to (and incl.) **101**
- Remove P7 from **Dirty Pages Table** and **Buffer Pool**

Transactions

txnID	lastLSN	Status
T100	101	running

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	P7	1, 5

Buffer Pool

P5	P6	P7
PageLSN=…	PageLSN=…	PageLSN=101

Dirty pages table

pageID	recLSN
P7	101

ARIES Normal Operation

Page flushes

- Flush log up to (and incl.) **pageLSN**
- Remove page from **Dirty Pages Table** and **Buffer Pool**

Ex: Buffer manager wants to FLUSH(P7)

- Flush **Log** up to (and incl.) **101**
- Remove P7 from **Dirty Pages Table** and **Buffer Pool**

Transactions

txnID	lastLSN	Status
T100	101	running

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	P7	1, 5

Buffer Pool

P5 PageLSN=…	**P6** PageLSN=…

Dirty pages table

pageID	recLSN

ARIES Normal Operation

Page fetches

- Create entry in **Dirty Pages Table** and **Buffer Pool**

Ex: Buffer manager wants FETCH(P2)

- Create entry in **Dirty Pages** table set **recLSN** = NULL
- Bring page into **Buffer Pool**

Transactions

txnID	lastLSN	Status
T100	101	running

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	P7	1, 5

Buffer Pool

	P2
P5 PageLSN=…	**P6** PageLSN=…

Dirty pages table

pageID	recLSN
P2	-

ARIES Normal Operation

Transaction commits

- Write commit record to Log
- Flush Log up to this entry
  - Txn is now considered committed!
- Update Transactions to commit
- Write end record to Log
- Update Transactions to complete

- Recall we are using WAL!

Transactions

txnID	lastLSN	Status
T100	102	running

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	-	START
102	-	T100	P7	1, 5

Buffer Pool

P5	P6	P7
PageLSN=…	PageLSN=…	PageLSN=101

Dirty pages table

pageID	recLSN
P7	101

ARIES Normal Operation

Transaction commits

Ex: T100 commits

- Write **<COMMIT,T100>**
- Flush Log up to this entry
- Update Transactions to commit
- Write **<END,T100>** in the Log
- Update Transactions to complete

Transactions

txnID	lastLSN	Status
T100	102	complete

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	-	START
102	-	T100	P7	1, 5
103	-	T100	-	COMMIT
104	-	T100	-	END

Buffer Pool

Dirty pages table

pageID	recLSN

ARIES Normal Operation

Transaction aborts

- Write abort record to Log
- Find first action to undo from lastLSN
- Go to log and start undo changes
- Write compensation record (CLR) to Log
  - CLR's undoNextLSN points to next record to undo (part of the payload)
- Follow **prevLSN** to retrace more (if any) actions to undo
- Once done, change Transactions to abort
- Write end record to Log
- Update Transactions to complete

Transactions

txnID	lastLSN	Status
T100	103	running

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	-	START
102	101	T100	P7	1, 5
103	102	T100	P6	2, 6

Buffer Pool

P5
PageLSN=…

P6
PageLSN=103

P7
PageLSN=102

Dirty pages table

pageID	recLSN
P7	101

ARIES Normal Operation

Ex: T100 aborts

- Write **<ABORT,T100>** to Log
- From lastLSN, LSN 103 is first to undo
- Undo 103 and write <CLR, 102, P6, 2>
  - 102 is the next record to undo
- Follow **prevLSN**, LSN 102 is next to undo
- Undo 102 and write <CLR, -, P7, 1>
  - No more actions to undo!
- Change Transactions to abort
- Write **<END,T100>** to Log
- Update Transactions to complete

Transactions

txnID	lastLSN	Status
T100	~~103~~ ~~106~~ 107	~~running~~ ~~abort~~ complete

Log (WAL)

LSN	prevLSN	txnID	pageID	Payload
101	-	T100	-	START
102	101	T100	P7	1, 5
103	102	T100	P6	2, 6
104	**103**	**T100**	**-**	**ABORT**
105	**104**	**T100**	**P6**	**CLR, 102, P6, 2**
106	**105**	**T100**	**P7**	**CLR, -, P7, 1**
107	**106**	**T100**	**-**	**END**

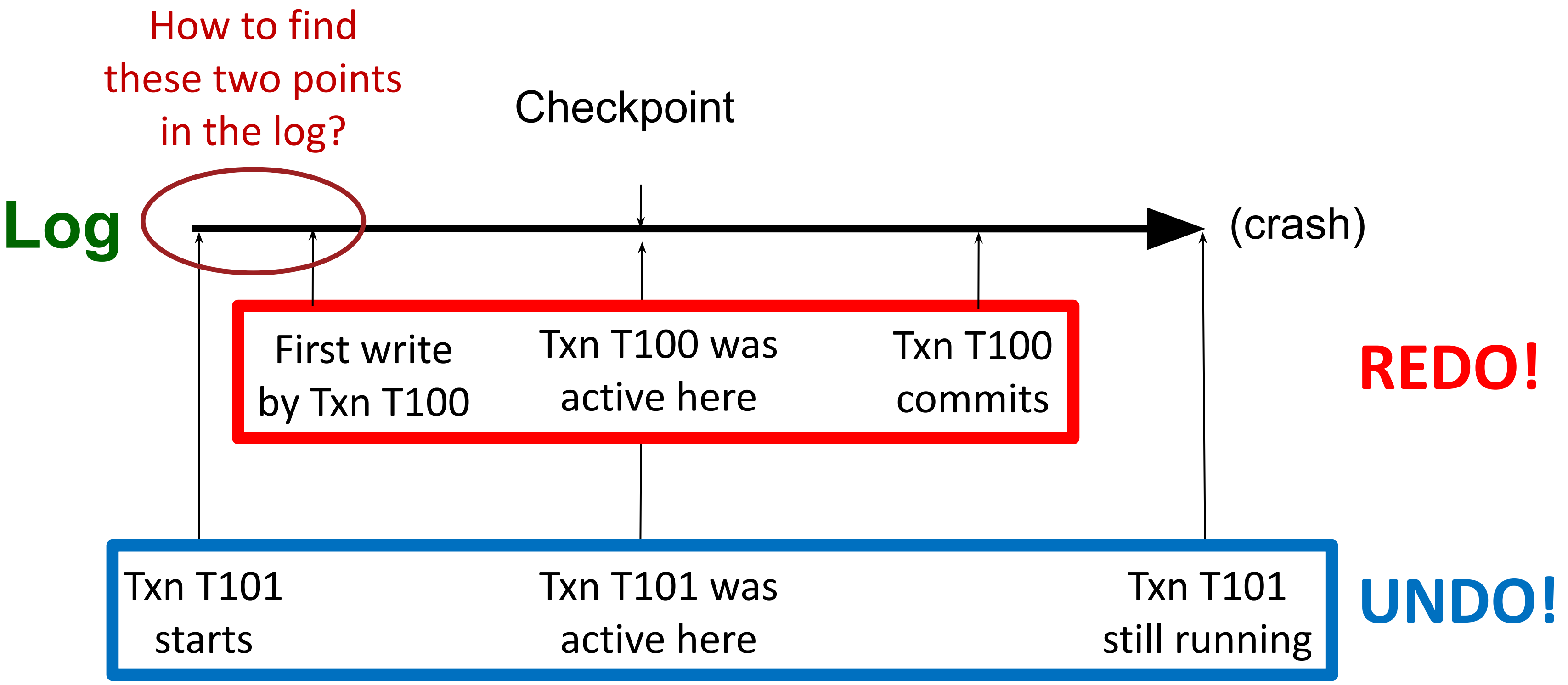
Will see a better algorithm in a few slides!

ARIES Recovery

- Start recovery from the last checkpoint
  - Easy to recover with non-fuzzy checkpoints. Just roll forward!
- With fuzzy checkpoints, we now need to handle:
  - Active transactions when checkpoint was taken
  - Dirty pages that were not flushed to disk yet

- Main principles:
  - Redo all actions before crash and bring DBMS to the exact state right when it crashed
  - Unroll changes from incomplete txns when crash occurred
  - Log all undo changes to ensure changes are not undone

ARIES Recovery



ARIES Recovery

Recovery from a system crash is done in 3 passes:

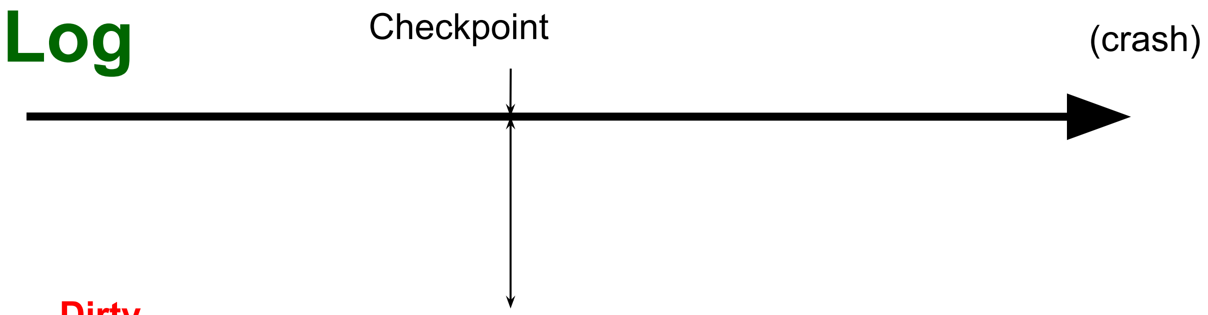
1. **Analysis pass**
   - Recreate list of dirty pages and active transactions
2. **Redo pass**
   - Redo all operations, even for those that were incomplete before crash
   - Goal is to replay DB to the state at the moment of the crash
3. **Undo pass**
   - Unroll effects of all incomplete transactions at time of crash
   - Log changes during undo in case of another crash during undo

1. Analysis Phase

- Goal
  - Determine point in log (**firstLSN**) where to start REDO
  - Determine set of dirty pages when crashed
  - Identify active transactions when crashed

- Approach
  - Rebuild transactions table and dirty pages table
  - Recover these from the last checkpoint in the log
  - Compute: **firstLSN** = smallest of all pages' recoveryLSN
    - This is the earliest point that a write was made to buffer pool that hasn't persisted yet

1. Analysis Phase



Dirty pages

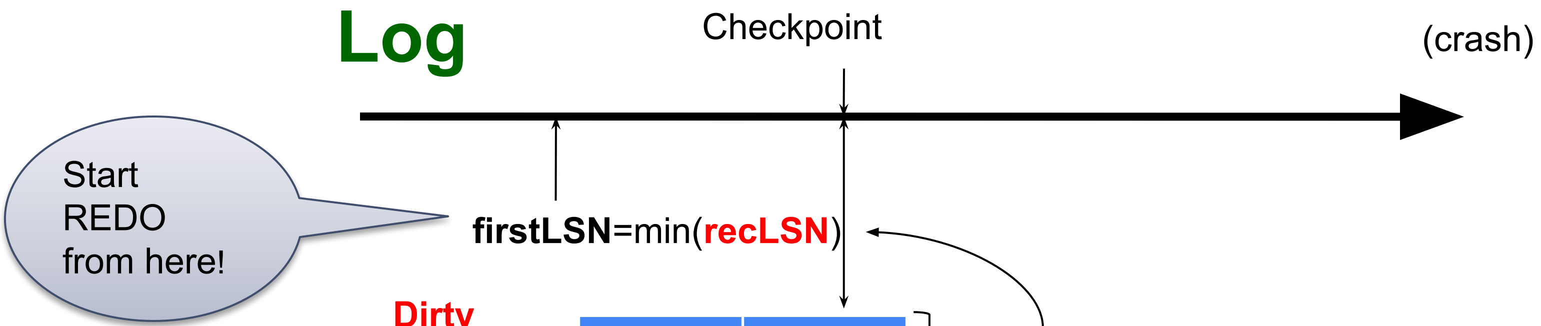
pageID	recLSN
…	…
…	…

Where do we start the REDO phase ?

Txn status

txnID	lastLSN
…	…
…	…

1. Analysis Phase



Dirty pages

pageID	recLSN
…	…
…	…

Txn status

txnID	lastLSN
…	…
…	…

2. Redo Phase

Main principle: replay history

- Process Log forward, starting from **firstLSN**
- Read every log record sequentially
- Redo actions are not recorded in the log

2. Redo Phase: Details

For each Log entry record LSN: **<T,P,old,new>**

- Write new value to page P
- Only redo those that need to be redone
  - How to determine that?

2. Redo Phase: Details

For each Log entry record LSN: **<T,P,old,new>**

- If P is not in Dirty Page then **don’t redo.** How did this happen?
  - P was flushed to DB, removed from DPT before checkpoint
  - *Then* DPT flushed at checkpoint

- P is in DPT, but recLSN > LSN, then **don’t redo.** How did this happen?
  - P was flushed to DB, removed from DPT before checkpoint
  - *Then* P was read in *again* and reinserted in DPT with larger recLSN

- P’s **pageLSN** on disk > LSN, then **don’t redo.** How did this happen?
  - P was updated again and flushed to DB after this log record

- Otherwise redo!

2. Redo Phase: Details

What happens if system crashes during REDO ?

We REDO again! Each REDO operation is *idempotent*: doing it twice is the as doing it once.

3. Undo Phase

- A simple solution:
  - All active txns in the Transactions Table are "losers."
  - Just abort each loser transaction
  - Problem?
    - Lots of random I/O in the log following the chain of prevLSNs (see earlier slide)
    - Can we do this in one backwards pass of log?

3. Undo Phase

- Define **ToUndo** = set of lastLSN from the loser txns
  - Get them from the transactions table

Transactions

txnID	**lastLSN**	**Status**
T100	103	complete
T101	**104**	**running**

3. Undo Phase: Details

While **ToUndo** not empty:

- Choose most recent (largest) LSN in **ToUndo**
- If LSN is a regular log record **<T,P,old,new,prevLSN>**:
  - Undo action
  - Write a CLR where CLR.undoNextLSN = LSN.prevLSN
  - If prevLSN is not null then insert prevLSN into **ToUndo**
    - otherwise, write end record in log (we have fully aborted the txn)
- If LSN is a CLR record:
  - Don't undo!
  - But if CLR.undoNextLSN not null, insert in **ToUndo**
  - otherwise, write end record in log (we have fully aborted the txn)

We're done when there are no more transactions to undo

We can use this algorithm to undo a single txn as well during normal operation

3. Undo Phase: Details

What happens if system crashes during UNDO ?

We do not UNDO again! Instead, each CLR is a REDO record: we simply redo the undo

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
    - We will assume that disk write is atomic.
        - In practice, kind of tricky!
- 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

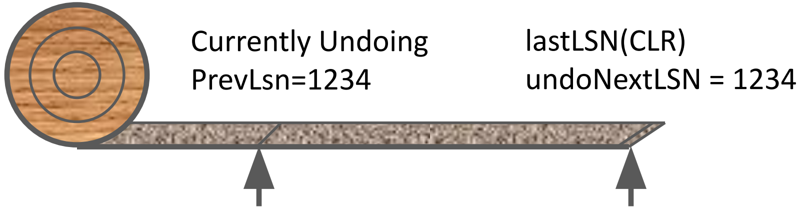
- Write **commit** record to log.
- All log records up to Xact's commit record are flushed to disk.
  - Guarantees that **flushedLSN ≥ lastLSN**.
  - Note that log flushes are sequential, synchronous writes to disk.
  - Many log records per log page.
- Commit() returns.
- Write end record to log.

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

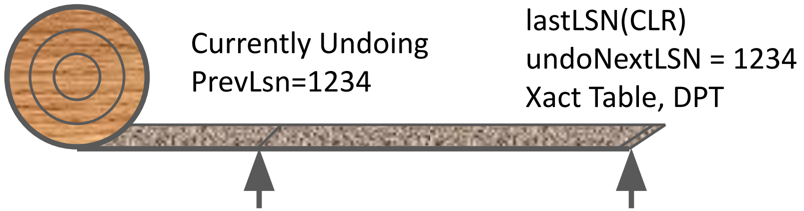
Abort, cont.



- 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

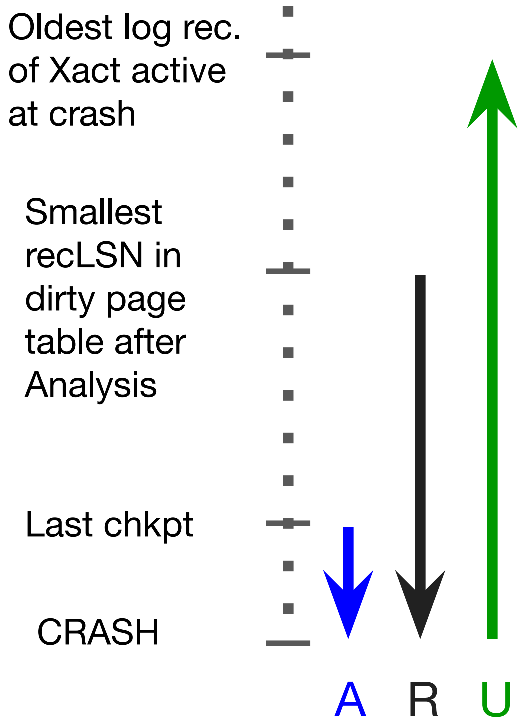


- Conceptually, keep log around for all time.
  - Performance/implementation problems…
- Periodically, the DBMS creates a **checkpoint**
  - Minimizes recovery time after crash. Write to log:
    - **begin_checkpoint** record: Indicates when chkpt began.
    - **end_checkpoint** record: Contains current 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).

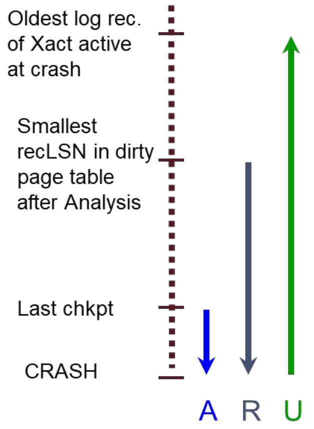
CRASH RECOVERY

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)
    - Reconstruct state of the DB before crash
  - **UNDO** effects of failed Xacts.



Recovery: The Analysis Phase

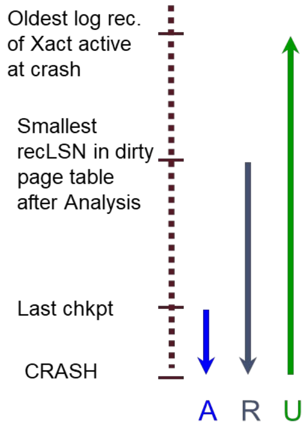


- Re-establish knowledge of state at checkpoint.
  - via transaction table and dirty page table stored in the checkpoint

- 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 or abort.

- 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



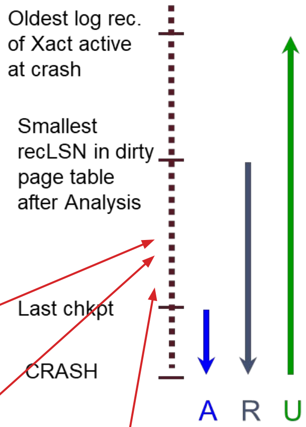
- We **Repeat History** to reconstruct state at crash:
  - Reapply **all** updates (even of aborted Xacts!), redo CLRs.
- Scan forward from log rec containing smallest recLSN in DPT.
  - Q: why start here?

- 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!

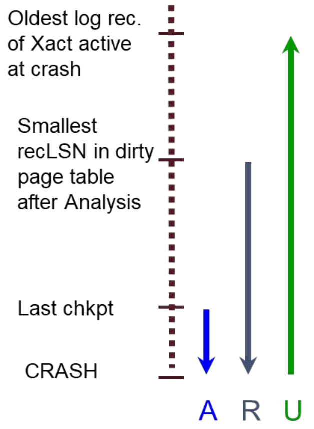
Scenarios When We Do Not REDO

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. *How?*
  - 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). *How?*
  - This page was updated again and flushed to DB
    after this log record

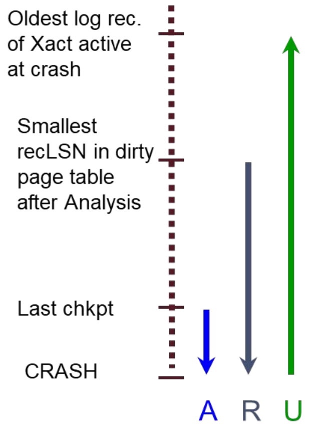


Phase 3: The UNDO Phase



- 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




```
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:
          if thisLR.type == UPDATE:
               write a CLR for the undo in the log
               undo the update in the database
          if thisLR.prevLSN != NULL:
               toUndo.insert(thisLR.prevLSN)
          elif thisLR.prevLSN == NULL:
               write an END record for thisLR.xid
```

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

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!

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!

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
70,80	T2 abort, T3 abort
90	CLR: Undo T2 LSN 60
100,105	CLR: Undo T3 LSN 50, T3 end
	CRASH, RESTART
110,115	CLR: Undo T2 LSN 20, T2 end

undonextLSN

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!