Aries: A Transaction Recovery Method

Slides modified by Rachel Pottinger from slides from “Database Management Systems” by Ramakrishnan and Gehrke

ACID Properties

- ***Atomicity:*** Either all actions in the Xact occur, or none occur.
- ***Consistency:*** If each Xact is consistent, and the DB starts in a consistent state, then the DB ends up being consistent.
- ***Isolation:*** The execution of one Xact is isolated from that of other Xacts.
- ***Durability:*** If a Xact commits, then its effects persist.

What happens if the system fails?

- The goal of transaction recovery is to resurrect the db if this happens
- Aries is one example of such a system
- A key tenant of Aries is fine granularity locking for 4 reasons
  1. OO systems make users think in small objects
  2. "Object-oriented system users may tend to have many terminal interactions during …"
  3. More system use → more hotspots → need less tuning
  4. Metadata is accessed often; cannot all be locked at once

The 9 Goals of Aries

1. Simplicity
2. Operation Logging
3. Flexible storage management
4. Partial rollbacks
5. Flexible buffer management
6. Recovery independence
7. Logical undo
8. Parallelism and fast recovery
9. Minimal overhead

Operation logging

“let one transaction modify the same data that was modified earlier by another transaction which has not yet committed, when the two transactions’ actions are semantically compatible”

Partial rollbacks

Support save points and rollbacks to save points in order to be user friendly

Handling the Buffer Pool

- Transactions modify pages in memory buffers
- Writing to disk is more permanent
- When should updated pages be written to disk?
- Force every write to disk?
  - Poor response time.
  - But provides durability.
- Steal buffer-pool frames from uncommitted Xacts? (resulting in write to disk)
  - If not, poor throughput.
  - If so, how can we ensure atomicity?

	No Steal	Steal
Force	Trivial	
No Force		Desired

Flexible buffer management

Make the least number of restrictive assumptions about buffer management policies

Recovery independence

“The recovery of one object should not force the concurrent or lock-step recovery of another object”

Group Discussion on the 9 Goals

Rank the goals from 1 to 9 where 1 is the most important and 9 is the least important

Basic Idea: Logging

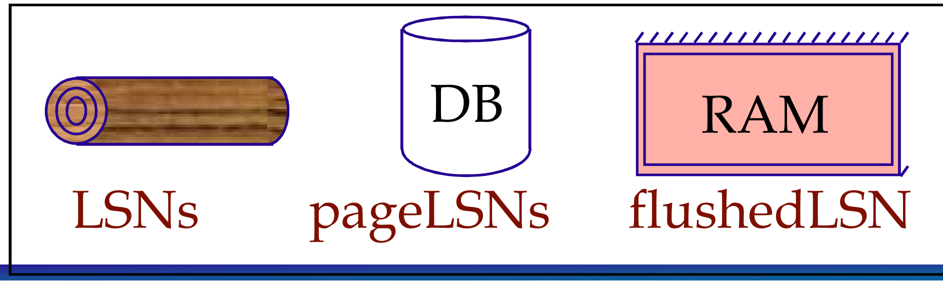
- Record REDO and UNDO information, for every update, in a *log.*
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.
- Log: An ordered list of REDO/UNDO actions
  - Log record contains:

    <XID, pageID, offset, length, old data, new data>
  - and additional control info (which we'll see soon).

Write-Ahead Logging (WAL)

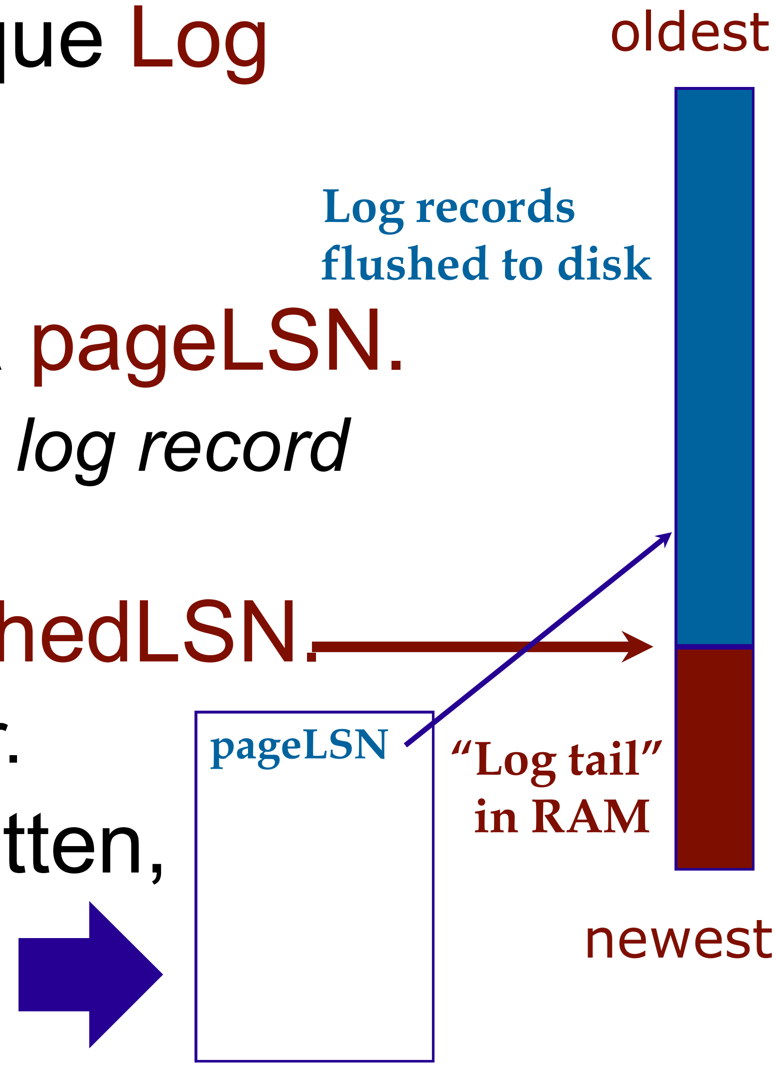
- The Write-Ahead Logging Protocol:
  1. Must force log record for an update *before* the corresponding data page gets to disk.
  2. Must write all log records for a Xact *before commit*.
- #1 guarantees Atomicity.
- #2 guarantees Durability.

WAL & the Log

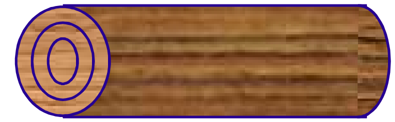


- Each log record has a unique Log Sequence Number (LSN).
  - LSNs always increasing.
- Each *data page* contains a pageLSN.
  - The LSN of the most recent *log record* for an update to that page.
- System keeps track of flushedLSN.
  - The max LSN flushed so far.
- WAL: *Before* a page is written,
  - pageLSN ≤ flushedLSN

I.e., the latest thing on disk must also be written to disk on the log



Log Records



LogRecord fields:

- prevLSN
- transID
- 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**
- **End** (signifies end of commit or abort)
- **Compensation Log Records (CLRs)**
  - for UNDO actions

before and after image are the data before and after the update.

Creating Log Entries

- ***Update :***
  - Inserted when modifying a page.
  - Contains all the fields.
  - pageLSN of that page is set to the LSN of the record (i.e., page updated)
- ***Commit :***
  - When Xact commits a record is written in the log and is <u>forcibly written</u> to stable storage.
- ***Abort :***
  - created when Xact is aborted
- ***End :***
  - created when Xact has completed all work (after commit or abort)
- ***Compensation Log Records (CLR) :***
  - Inserted before undoing an action described by an update log record
  - It happens during aborting or recovery.
  - Contains **undoNextLSN** field: LSN of next log record to be undone.

Other Log-Related Structures

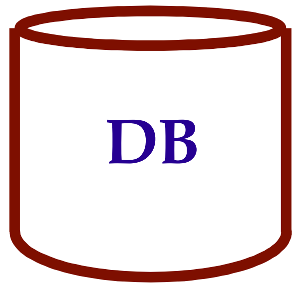
Transaction manager also maintains the following tables

- ***Transaction Table:***
  - Maintained by transaction manager
  - Has one entry per active Xact
  - Contains ***tranID, status*** (running/committed/aborted), and ***lastLSN*** (LSN of most recent log record for it)
  - Xact removed from table when end record is inserted in the log
- ***Dirty Page Table:***
  - Maintained by buffer manager
  - Has one entry per dirty page in buffer pool
  - Contains ***recLSN*** -- LSN of action which ***first*** made the page dirty
  - Entry is removed when page is written to the disk
- Both tables must be reconstructed during recovery.

The Big Picture: What's Stored Where

LogRecords
- prevLSN
- transID
- type
- pageID
- length
- offset
- before-image
- after-image

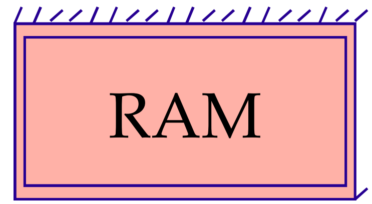
Part of DBMS, but not in db (too slow)



Data pages
each with a pageLSN

master record

Last to update page



Xact Table
- lastLSN
- status

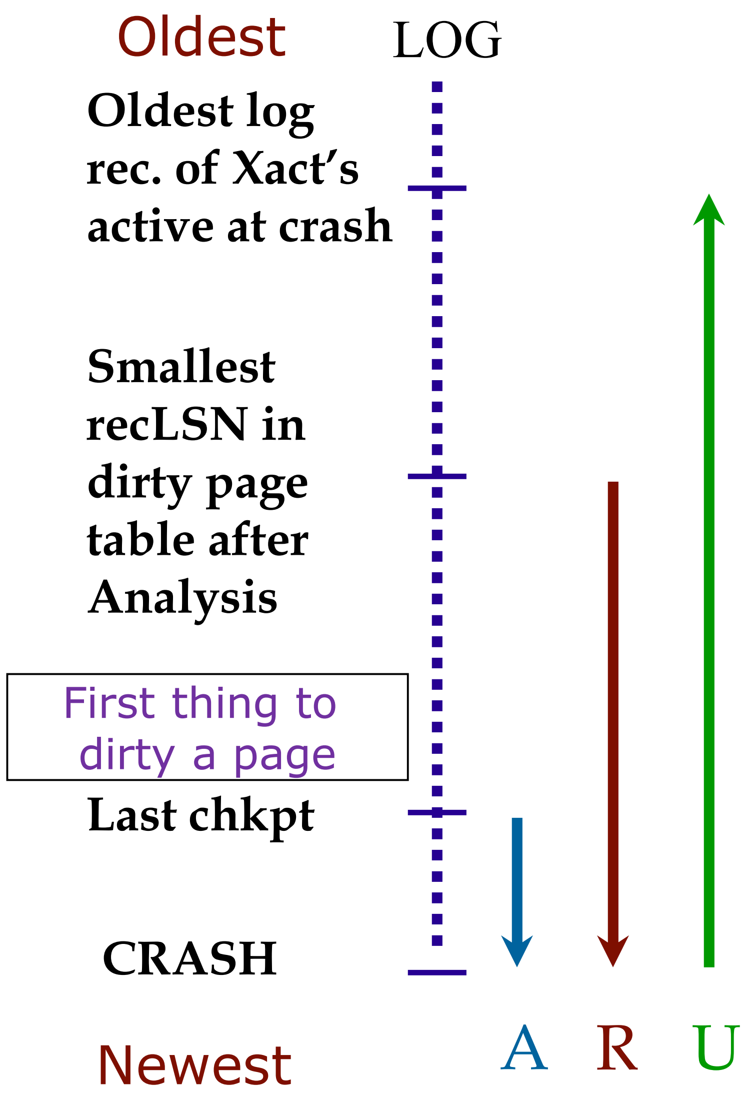
Dirty Page Table
- recLSN

First thing made it dirty

Checkpoints

- Periodically***checkpoint***, to minimize recovery time in system crash. Write to log:
  - ***begin_checkpoint*** record: when checkpoint began
  - ***end_checkpoint*** record: current *Xact table* and *dirty page table*.
- Aries uses a '***fuzzy checkpoint***':
    - Xacts continue to run; so these tables are accurate only as of time of begin_checkpoint
    - Dirty pages are *not* forced to disk;
  - Store LSN of checkpoint record in a safe place (***master*** record).
- When system starts after a crash:
  - Locate the most recent checkpoint
  - Restore Xact table and dirty page table from there.

Crash Recovery: Big Picture



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

Go back far because "fuzzy" checkpoint

Recovery: The Analysis Phase

- Goals:
  - Determine log record that Redo has to start at
  - Determine pages that were dirty at crash
  - Identify Xact's active at crash
- Reconstruct state at checkpoint
  - reconstruct Xact & dirty page tables using **end_checkpoint** record
- Scan log forward from checkpoint
  - End record: Remove Xact from Xact table
  - Other bookkeeping happens

Recovery: The REDO Phase

- We *repeat history* to reconstruct state at crash:
  - Reapply *all* updates (even of aborted Xacts), redo CLRs
- Scan forward from log record containing <u>smallest recLSN in DPT</u>. For each CLR or update log record, REDO the action unless it's clear that it's already been recorded (details omitted)
- To REDO an action:
  - Reapply logged action
  - Set pageLSN to LSN. Know it's done – eventually written
  - No additional logging is required!
- At the end of REDO, and End record is inserted in the log for each transaction with status C which is removed from Xact table.

Recovery: The UNDO Phase

- ***Loser Xact's*** = Xact active at the crash
- Need to undo all records of loser Xact's in reverse order
- ToUndo = set of all lastLSN values of all loser Xact's

Algorithm: Those are the trans. we must undo

Repeat:

- Choose largest LSN among ToUndo
- If this LSN is a **CLR** and **undonextLSN==NULL**
  - write an End record for this Xact. All undone
  - remove record from ToUndo set
- If this LSN is a **CLR**, and **undonextLSN != NULL**
  - add undonextLSN to ToUndo Make sure you undo it
- Else this LSN is an update.
  - undo the update, write a CLR, Undo, log
  - remove record from toUndo We've done it
  - add prevLSN of this record to ToUndo. Undo next for trans.

Until ToUndo is empty

Discussion Questions

If you are designing a system for transaction processing,

- would you redo “loser” transactions?
- would you use selective redo?
- would you do a checkpoint after the analysis phase?

Why or why not?

Example of Recovery

Assume flush at checkpoint

RAM

transaction Table

T# lastLSN

Dirty Page Table — *first* made dirty

T# recLSN

ToUndo

FlushedLSN — Max log flushed

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

Check prev lsn, undo

Delete from transaction table

Example: Crash During Restart!

Still assume flush at checkpoint

RAM

transaction Table

T# lastLSN

Dirty Page Table

T# recLSN

ToUndo

FlushedLSN

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!

Still assume flush at checkpoint

RAM

transaction Table
T# lastLSN

Dirty Page Table
T# recLSN

ToUndo

FlushedLSN

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 Still maybe not on disk
90	CLR: Undo T2 LSN 20, T2 end

undonextLSN

Discussion

The authors claim that the system is simple and efficient. Do you agree or disagree with each claim? Why or why not?

Today's Recovery Algorithms

- Most popular are like ARIES:
  - maintain a log
  - use WAL
- Some Redo phases are different:
  - they don't repeat the whole history
  - they only redo the non-loser transactions – "selective redo"
    - Can lead to trouble because must log undos (for media recovery), then would attempt to redo undo