Recovery: ARIES normal operation and crash recovery procedure
Wrap-up of first part of course

ACS, Marcos Vaz Salles

NOTE: Transactional Scale-Up Experiments

- Refer to Part 12 of compendium, DeWitt and Gray paper: transactional scale-up consists of "**N-times as many clients, submitting N-times as many requests against N-times larger database.**"
- How to operationalize evaluation?
  - Find configuration for unit system size (e.g., one core)
  - Scale system size together with configuration

Do-it-yourself-recap: What's Stored Where

Explain each of the ARIES structures below to your colleague!

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

Data pages
each
with a
pageLSN

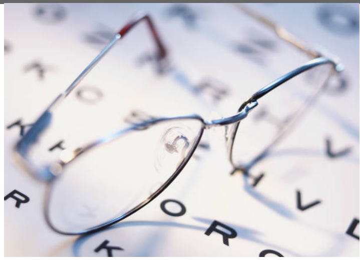
master record

Xact Table
- lastLSN
- status

Dirty Page Table
- recLSN

flushedLSN

Source: Ramakrishnan & Gehrke (partial)



What should we learn today?

- Explain how write-ahead logging is achieved in the ARIES protocol
- Explain the functions of recovery metadata such as the transaction table and the dirty page table
- Interpret the contents of the log resulting from ARIES normal operation
- Explain the three phases of ARIES crash recovery: analysis, redo, and undo
- Predict how recovery metadata, system state, and the log are updated during recovery

+ a discussion of where we got so far, if time allows

Normal Execution of an Xact

Keep in Mind:
It must be OK to crash at **any time**
→ **repeat history!**

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

Transaction Commit

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

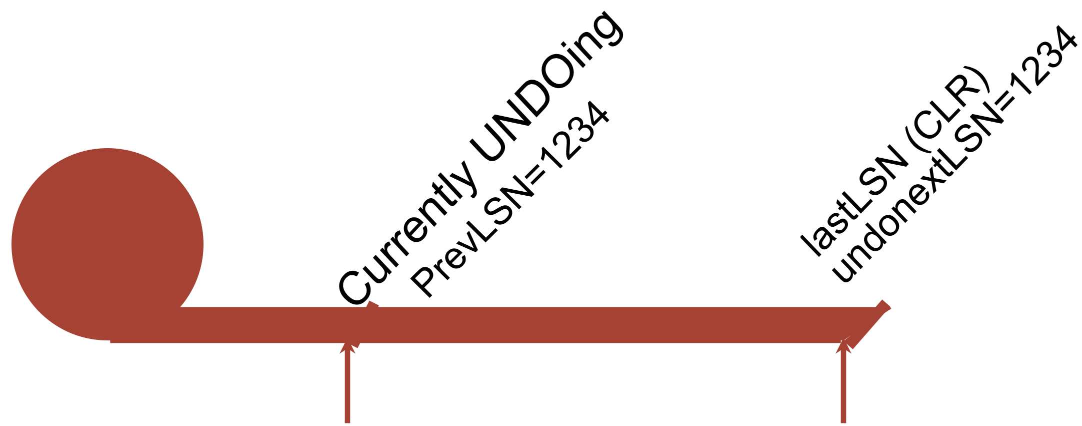
Why?

Source: Ramakrishnan & Gehrke (partial)

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.
  - Can follow chain of log records backward via the prevLSN field.
  - Before starting UNDO, write an *Abort* log record.
    - For recovering from crash during UNDO!

Abort, cont.



- To perform UNDO, must have a lock on data!
  - Strict 2PL enforces this
- 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).
  - CLRs *never* Undone (but they might be Redone when repeating history: guarantees Atomicity!)
- At end of UNDO, write an end log record.

Checkpointing

- Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover in the event of a system crash. Write to log:
  - begin_checkpoint record: Indicates when chkpt began.
  - 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, minDirtyPagesLSN.
    - Use **background process** to flush dirty pages to disk!
  - Store LSN of chkpt record in a safe place (*master* record).

Example

- 10 T1 writes P5
- 20 T2 writes P17
- 30 T1 writes P3

P3 written to disk

(pageLSN for page 3 at this time is 30)

- 40 T1 aborts
- 50 CLR T1 P3 (undonextLSN: 10)
- 60 CLR T1 P5 (undonextLSN: NULL)
- 70 End T1

Source: Ramakrishnan & Gehrke (partial)

A Longer Example

Discussion: Does this example make sense? Can you explain to your colleague what happened?

- 10 T1 writes P3 (prevLSN: NULL)
- 20 T2 writes P4 (prevLSN: NULL)
- 30 T2 writes P5 (prevLSN: 20)
- flushedLSN = 20

P4 gets written to disk (pageLSN for page 4 = 20)
T2 aborts

- 50 Abort T2
- 60 CLR T2 P5 (undoNextLSN = 20), pageLSN(P5)=60

Update P5 in the buffer manager
Flush log up to log record 60
Buffer manager writes P5 to disk.

- 70 CLR T2 P4 (undoNextLSN = NULL), pageLSN(P4)=70

Update page P4

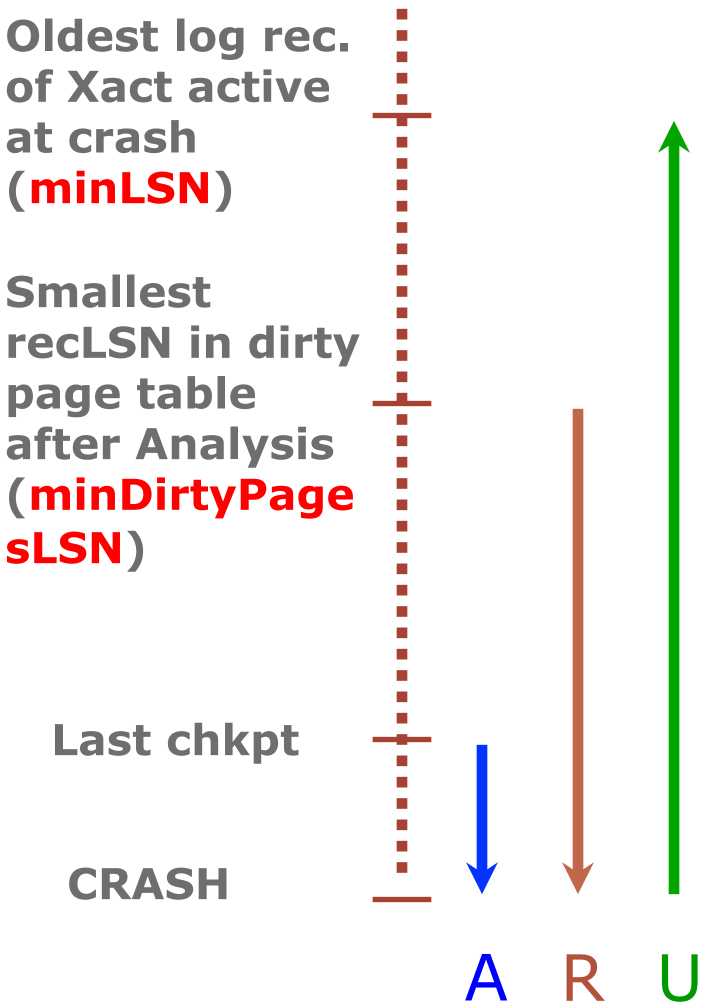
- 80 End T2
- 90 T1 commits

Flush log up to log record 90, then the commit(T1) returns

Questions so far?

Crash Recovery: Big Picture

Keep in Mind:
It must be OK to crash at **any time** (including during recovery)



- 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.

Source: Ramakrishnan & Gehrke (partial)

Recovery: The Analysis Phase

- Reconstruct state at checkpoint.
  - via end_checkpoint record.

- Scan log forward from checkpoint.
  - End record: Remove Xact from Xact table.
  - Other records: Add Xact to Xact table, set lastLSN=LSN, change Xact status on commit.
  - Update record: If P not in Dirty Page Table,
    - Add P to D.P.T., set its recLSN=LSN.

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 rec containing smallest recLSN in D.P.T. For each CLR or update log rec 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.

- To REDO an action:
  - Reapply logged action.
  - Set pageLSN to LSN. No additional logging! (Why?)

Source: Ramakrishnan & Gehrke (partial)

Recovery: The UNDO Phase

ToUndo={ *lsn* | *lsn* a lastLSN of a "loser" Xact}

Repeat:

- Choose 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.

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

Source: Ramakrishnan & Gehrke (partial)

Example: Crash During Restart!

Another example: crash during recovery!

RAM

Xact Table
lastLSN
status
Dirty Page Table
recLSN
flushedLSN

ToUndo

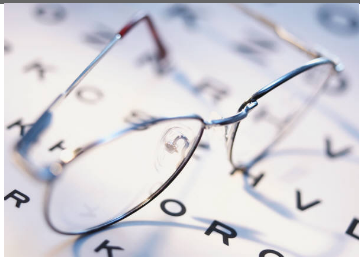
LSN	LOG
00,05	begin_checkpoint, end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40,45	CLR: Undo T1 LSN 10, T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART
64,65,70	T2 abort, T3 abort, CLR: Undo T2 LSN 60
80,85	CLR: Undo T3 LSN 50, T3 end
	CRASH, RESTART
90, 95	CLR: Undo T2 LSN 20, T2 end

undonextLSN

Source: Ramakrishnan & Gehrke (partial)

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.
  - Watch "hot spots"!
- How do you limit the amount of work in UNDO?
  - Avoid long-running Xacts.



What should we learn today?

- Explain how write-ahead logging is achieved in the ARIES protocol
- Explain the functions of recovery metadata such as the transaction table and the dirty page table
- Interpret the contents of the log resulting from ARIES normal operation
- Explain the three phases of ARIES crash recovery: analysis, redo, and undo
- Predict how recovery metadata, system state, and the log are updated during recovery

+ a discussion of where we got so far, if time allows

ACS: Evaluating the Course

- Your qualitative feedback is very important!

Comments on the course?
Syllabus?
Lectures?
TA sessions? Assignments?

Homework discussion: Granularity of Locking

- Suppose application with operations that read on a predicate on whole database
  - E.g., Programming Assignment 2! ☺
- **Locking Approach 1:** Single database lock
- **Locking Approach 2:** Multi-granularity locking
  - For example, one database lock + individual locks per item

- What are the trade-offs between the two approaches?
- Compare them in terms of: (a) Overhead
  (b) Implementation complexity
  (c) Concurrency

	--	IS	IX	S	X
--	√	√	√	√	√
IS	√	√	√	√	
IX	√	√	√		
S	√	√		√	
X	√				

Source: Ramakrishnan & Gehrke (partial)