

# EECS 262a

## Advanced Topics in Computer Systems

### ARIES: Logging and Recovery

CS186 Slides by Joe Hellerstein  
Enhanced by Alan Fekete

If you are going to be in the logging business, one of the things that you have to do is to learn about heavy equipment.

~ Robert VanNatta,  
*Logging History of Columbia County*





## Today's Paper

- [ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-ahead Logging,](#)  
C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh and Peter Schwarz. Appears in Transactions on Database Systems, Vol 17, No. 1, March 1992, Pages 94-162
- Thoughts?
- State-of-the-art/historical survey (Ch. 10)



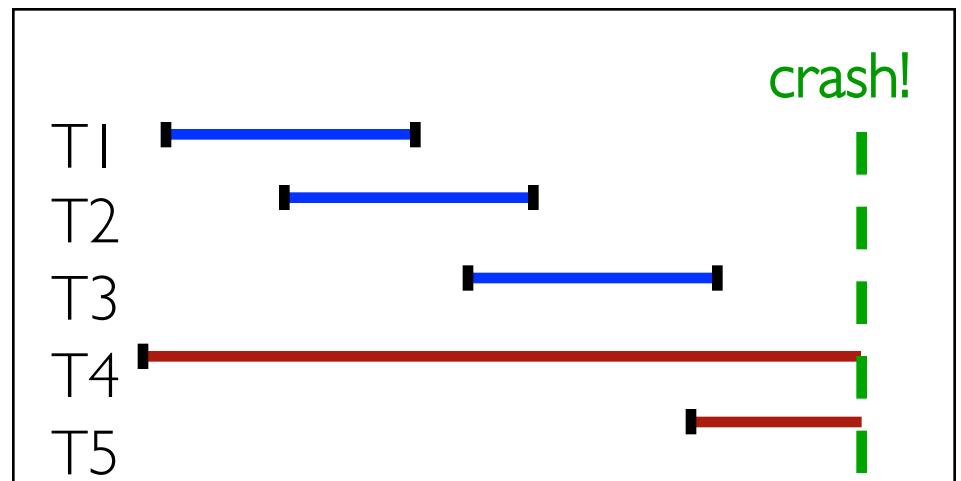
## Review: The ACID properties

- **A tomicity:** All actions in the Transaction happen, or none happen
- **C onsistency:** If each Transaction is consistent, and the DB starts consistent, it ends up consistent
- **I solation:** Execution of one Transaction is isolated from that of other Transactions
- **D urability:** If a Transaction commits, its effects persist
- *The Recovery Manager guarantees Atomicity & Durability*



# Motivation

- Atomicity:
  - Transactions may abort (“Rollback”)
- Durability:
  - What if DBMS stops running? (Causes?)
- ❖ Desired Behavior after system restarts:
  - T1, T2 & T3 should be durable
  - T4 & T5 should be aborted (effects not seen)





## Intended Functionality

- At any time, each (visible) data item contains the value produced by the most recent update done by a transaction that committed



# Assumptions

- Essential concurrency control is in effect
  - For read/write items: Write locks taken and held till commit
    - E.g., Strict 2PL, but read locks not important for recovery
  - For more general types: operations of concurrent transactions commute
- Updates are happening “in place”
  - i.e. data is overwritten on (deleted from) its location
    - Unlike multiversion (e.g., shadow pages) approaches
- Buffer in volatile memory; data persists on disk



## Challenge: REDO

Action	Buffer	Disk
Initially		0
T1 writes I	I	0
T1 commits	I	0
CRASH		0

- Need to restore value I to item
  - Last value written by a committed transaction



## Challenge: UNDO

Action	Buffer	Disk
Initially		0
T1 writes 1	1	0
Page flushed		1
CRASH		1

- Need to restore value 0 to item
  - Last value from a committed transaction



## Handling the Buffer Pool

- Can you think of a simple scheme to guarantee Atomicity & Durability?
- Force write to disk at commit?
  - Poor response time
  - But provides durability
- No Steal of buffer-pool frames from uncommitted Transactions (“pin”)?
  - Poor throughput
  - But easily ensure atomicity

Force

No Force

	No Steal	Steal
Trivial		
Desired		



## More on Steal and Force

- STEAL (why enforcing Atomicity is hard)
  - *To steal frame F:* Current page in F (say P) is written to disk; some Transaction holds lock on P
    - What if the Transaction with the lock on P aborts?
    - Must remember the old value of P at steal time (to support **UNDO**ing the write to page P)
- NO FORCE (why enforcing Durability is hard)
  - What if system crashes before a modified page is written to disk?
  - Write as little as possible, in a convenient place, at commit time, to support **REDO**ing modifications



## 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)
  - For abstract types, have operation(args) instead of old value new value



# Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  1. Must force the log record for an update before the corresponding data page gets to disk
  2. Must write all log records for a Transaction before commit
- #1 (undo rule) allows system to have Atomicity
- #2 (redo rule) allows system to have Durability



# ARIES

- Exactly how is logging (and recovery!) done?
  - Many approaches (traditional ones used in relational systems of 1980s)
  - ARIES algorithms developed by IBM used many of the same ideas, and some novelties that were quite radical at the time
    - Research report in 1989; conference paper on an extension in 1989; comprehensive journal publication in 1992
    - 10 Year VLDB Award 1999



## Key ideas of ARIES

- Log every change (even *UNDOs during Transaction abort*)
- In restart, *first* repeat history without backtracking
  - Even *REDO the actions of loser transactions*
- *Then UNDO actions of losers*
  
- LSNs in pages used to coordinate state between log, buffer, disk

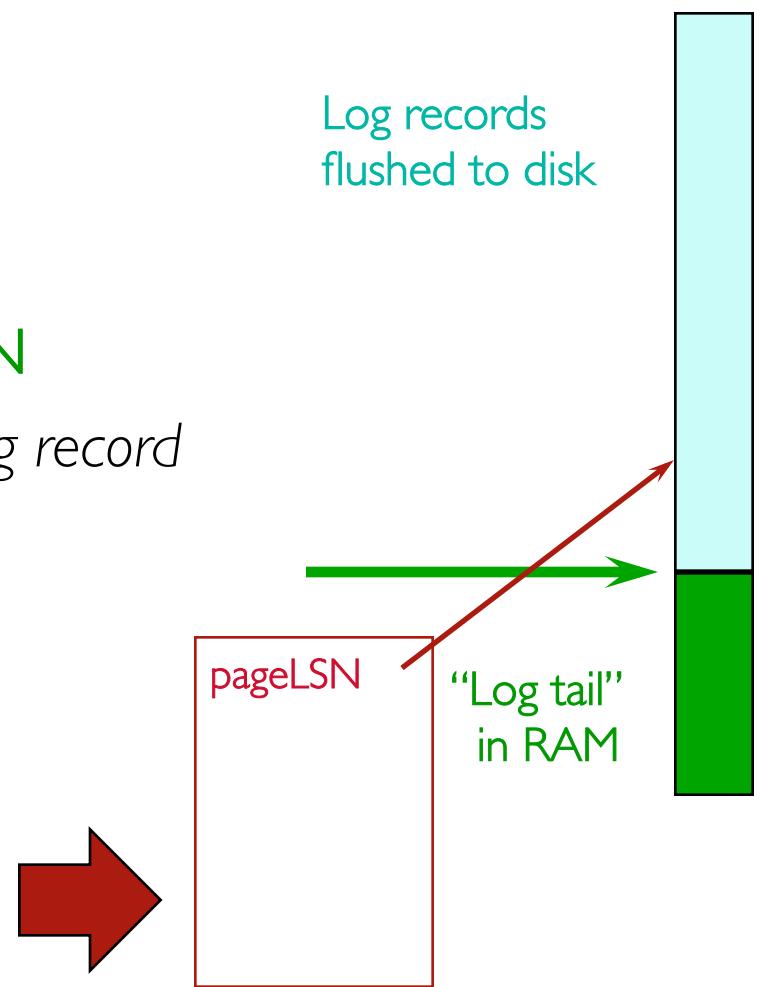
Novel features of ARIES *in italics*



## 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 constraints

- *Before* a page is written,
  - $\text{pageLSN} \leq \text{flushedLSN}$
- Commit record included in log; all related update log records precede it in log

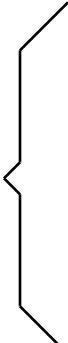


# Log Records

## LogRecord fields:

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

update records only



## Possible log record types:

- Update
- Commit
- Abort
- End (signifies end of commit or abort)
- Compensation Log Records (CLRs)
  - for UNDO actions
  - (and some other tricks!)



## Other Log-Related State

- **Transaction Table:**
  - One entry per active Transaction
  - Contains **XID**, **status** (running/committed/aborted), and **lastLSN**
- **Dirty Page Table:**
  - One entry per dirty page in buffer pool
  - Contains **recLSN** – the LSN of the log record which first caused the page to be dirty



## Normal Execution of a Transaction

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



# 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 *Transaction table* and *dirty page table*. This is a 'fuzzy checkpoint':
    - Other Transactions continue to run; so these tables only known to reflect some mix of state *after 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. (So it's a good idea to periodically flush dirty pages to disk!)
  - Store LSN of chkpt record in a safe place (`master` record)



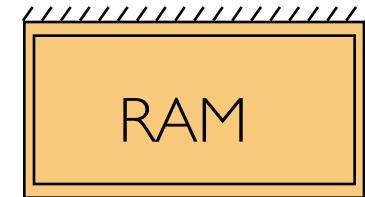
# The Big Picture: What's Stored Where



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



**Data pages**  
each  
with a  
pageLSN  
  
**master record**



**Transaction Table**  
lastLSN  
status

**Dirty Page Table**  
recLSN

**flushedLSN**

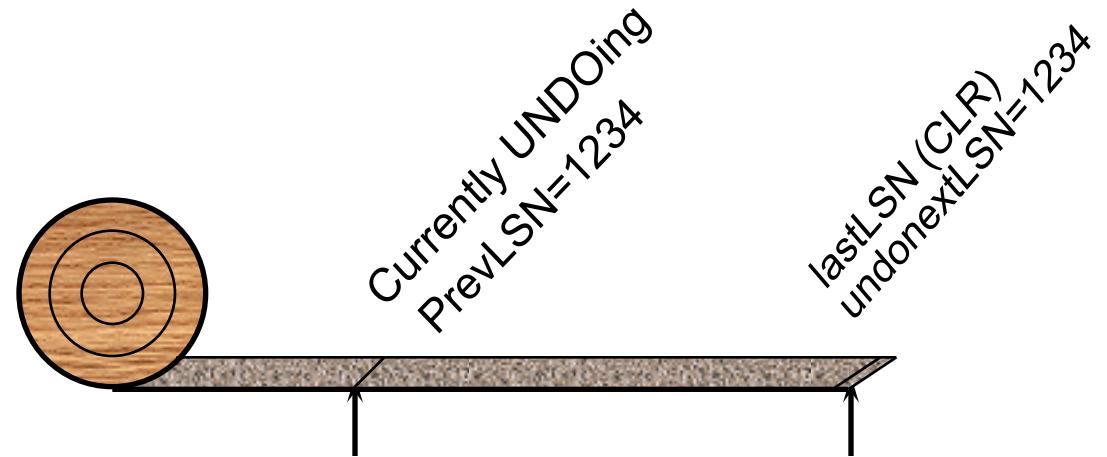


## Simple Transaction Abort

- For now, consider an explicit abort of a Transaction
  - No crash involved
- We want to “play back” the log in reverse order, UNDOing updates.
  - Get **lastLSN** of Transaction from Transaction table
  - Can follow chain of log records backward via the **prevLSN** field
  - Note: before starting UNDO, could write an **Abort log record**
    - Why bother?



## 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: **undoneextLSN**
    - 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

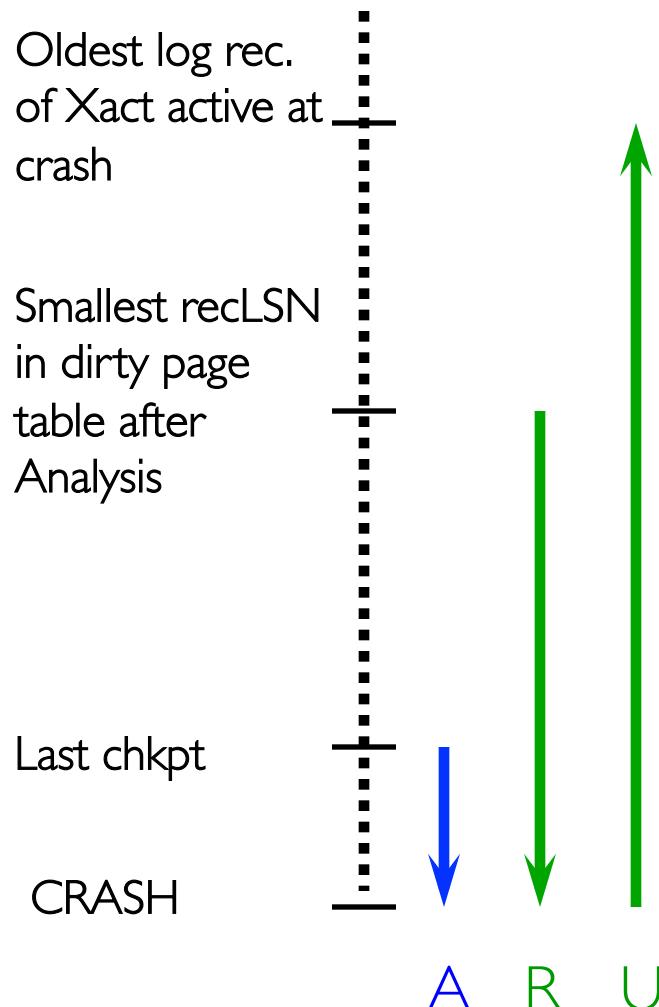


## Transaction Commit

- Write **commit** record to log
- All log records up to Transaction's **lastLSN** are flushed
  - Guarantees that **flushedLSN  $\geq$  lastLSN**
  - Note that log flushes are sequential, synchronous writes to disk
  - Many log records per log page
- Make transaction visible
  - Commit() returns, locks dropped, etc.
- Write **end** record to log



# 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.



## Recovery: The Analysis Phase

- Reconstruct state at checkpoint
  - via `end_checkpoint` record
- Scan log forward from `begin_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`

This phase could be skipped;  
information can be regained in subsequent REDO pass



## 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 page is already more up-to-date than this record:
  - REDO when Affected page is in D.P.T., and has **pageLSN** (in DB) < **LSN**. [if page has **recLSN** > **LSN** no need to read page in from disk to check **pageLSN**]
- To **REDO** an action:
  - Reapply logged action
  - Set **pageLSN** to **LSN**. No additional logging!



## Invariant

- State of page P is the outcome of all changes of relevant log records whose LSN is  $\leq P.\text{pageLSN}$
- During redo phase, every page P has  $P.\text{pageLSN} \geq \text{currently-redoing-LSN}$
- Thus at end of redo pass, the database has a state that reflects exactly everything on the (stable) log

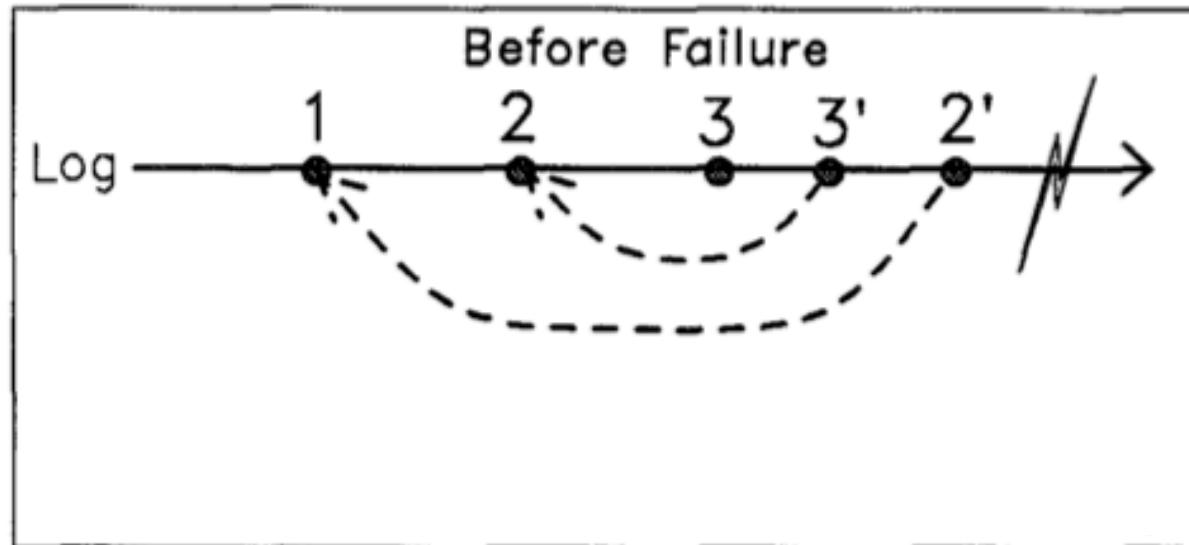


## Recovery: The UNDO Phase

- Key idea: Similar to simple transaction abort, for each loser transaction (that was in flight or aborted at time of crash)
  - Process each loser transaction's log records backwards; undoing each record in turn and generating CLRs
- But: loser may include partial (or complete) rollback actions
- Avoid undo-ing what was already undone
  - undoNextLSN field in each CLR equals prevLSN field from the original action



## UndoNextLSN



I' is the Compensation Log Record for I  
I' points to the predecessor, if any, of I

Fig. 5. ARIES' technique for avoiding compensating compensations and duplicate compensations.

From Mohan et al, TODS 17(1):94-162  
Cs262a-F14 Lecture-06



## Recovery: The UNDO Phase

ToUndo={ | | | 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 Transaction
- If this LSN is a CLR, and undonextLSN != NULL
  - Add undonextLSN to ToUndo
  - (Q: what happens to other CLRs?)
- Else this LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo

Until ToUndo is empty



## Restart Recovery Example

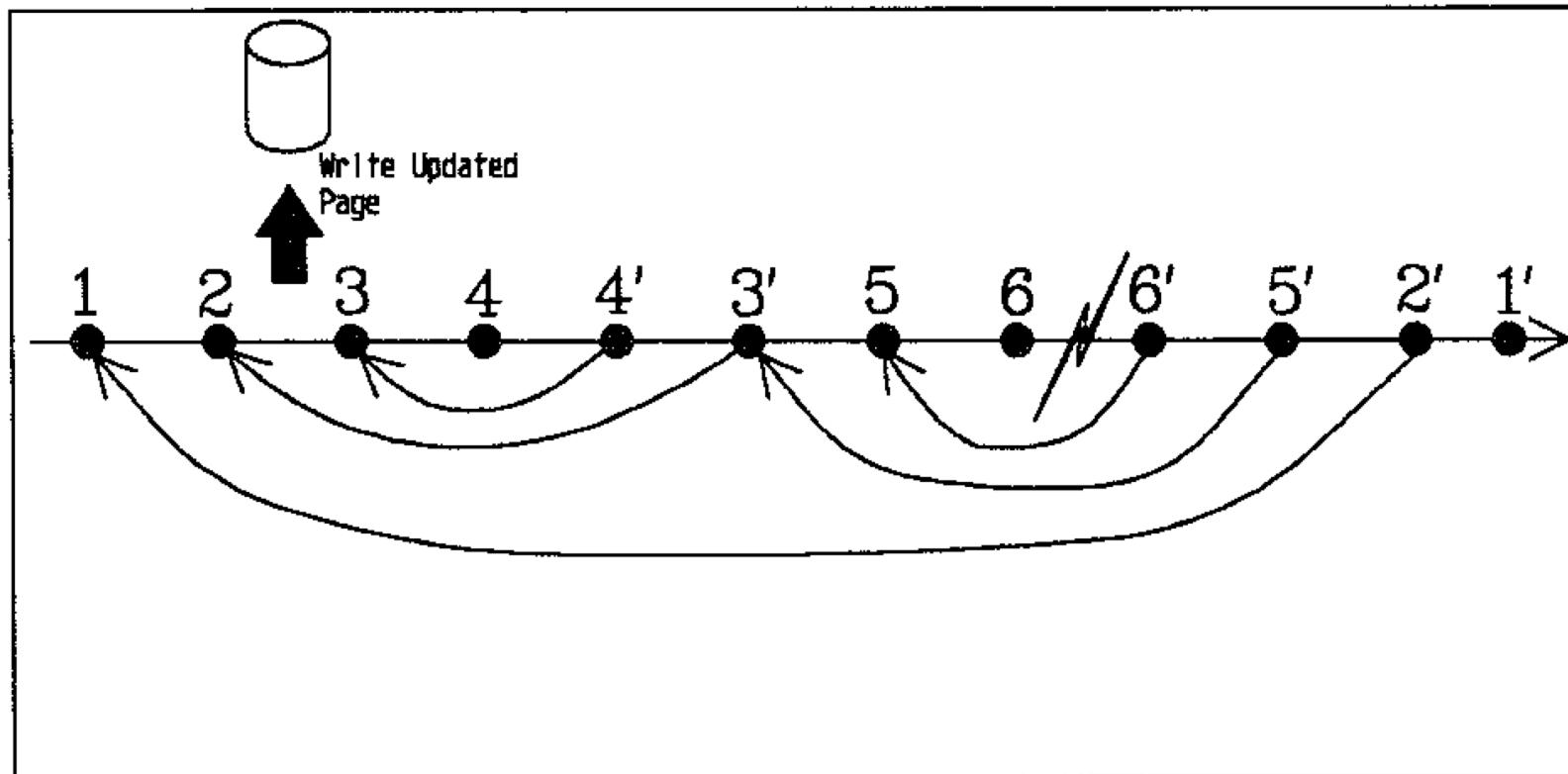
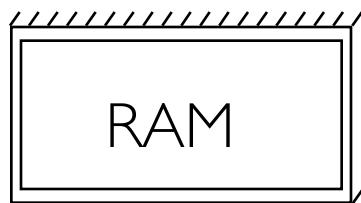


Fig. 13. Restart recovery example with ARIES.

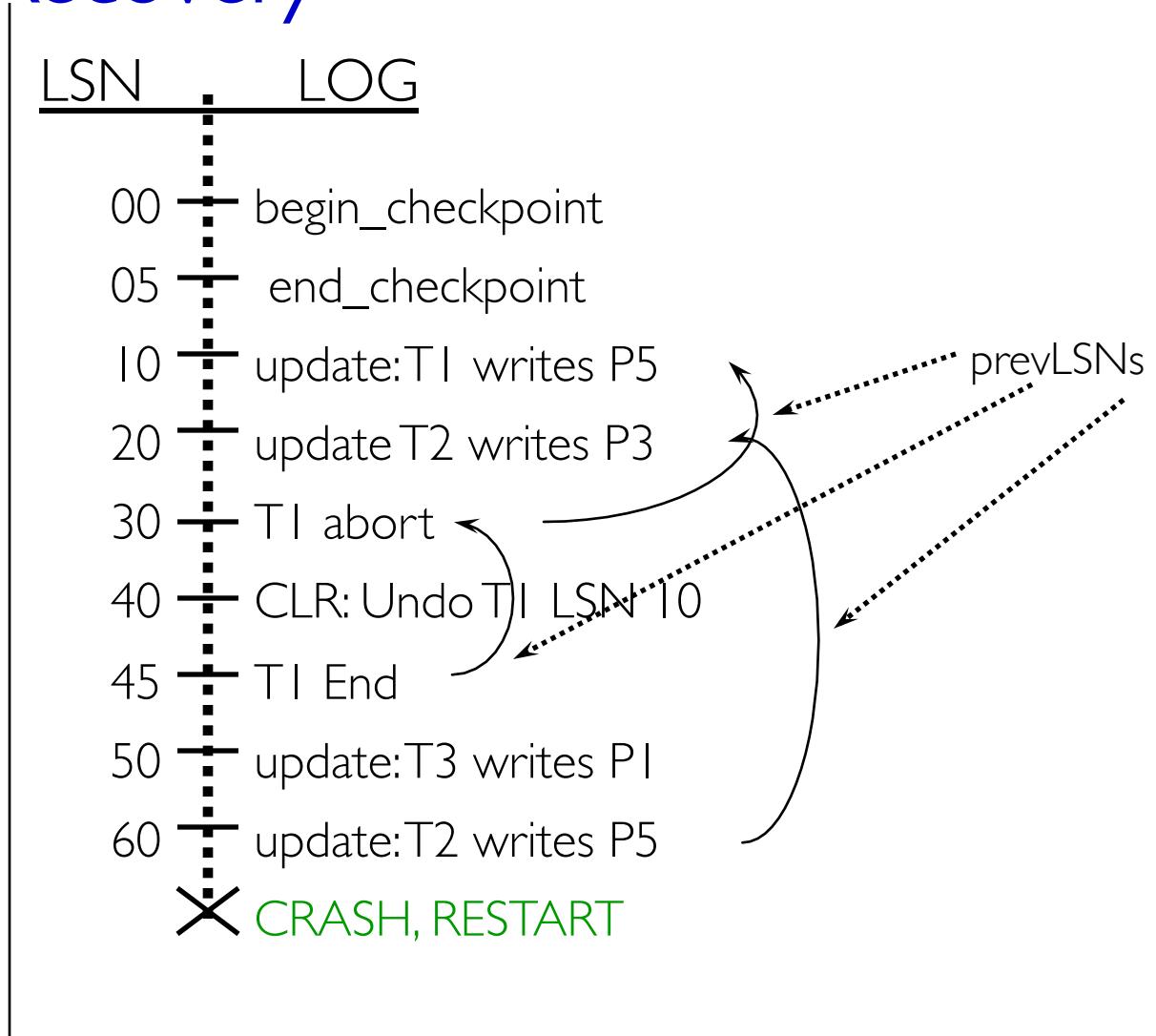


## Example of Recovery



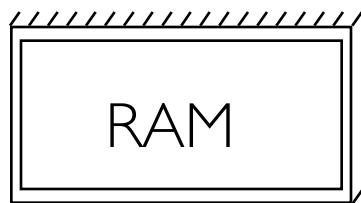
Xact Table  
lastLSN  
status  
Dirty Page Table  
recLSN  
flushedLSN

ToUndo





## Example: Crash During Restart!



Xact Table  
lastLSN  
status  
Dirty Page Table  
recLSN  
flushedLSN

ToUndo

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



## 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.



## Parallelism during restart

- Remember the invariants!
- Activities on a given page must be processed in sequence
- Activities on different pages can be done in parallel



## Log record contents

- What is actually stored in a log record, to allow REDO and UNDO to occur?
- Many choices, 3 main types
  - PHYSICAL
  - LOGICAL
  - PHYSIOLOGICAL



## Physical logging

- Describe the bits (optimization: only those that change)
- Example
  - OLD STATE: 0x47A90E....
  - NEW STATE: 0x632F00...
  - So REDO: set to NEW; UNDO: set to OLD
- Or just delta (OLD XOR NEW)
  - DELTA: 0x24860E...
  - So REDO=UNDO=xor with delta
- Ponder: XOR is not idempotent, but redo and undo must be; why is this OK?



## Logical Logging

- Describe the operation and arguments
- E.g., Update field 3 of record whose key is 37, by adding 32
- We need a programmer supplied inverse operation to undo this



# Physiological Logging

- Describe changes to a specified page, logically within that page
- Goes with common page layout, with records indexed from a page header
- Allows movement within the page (important for records whose length varies over time)
- E.g., on page 298, replace record at index 17 from old state to new state
- E.g., on page 35, insert new record at index 20



# ARIES logging

- ARIES allows different log approaches; common choice is:
- Physiological REDO logging
  - Independence of REDO (e.g. indexes & tables)
    - Can have concurrent commutative logical operations like increment/decrement ("escrow transactions")
- Logical UNDO
  - To allow for simple management of physical structures that are invisible to users
    - CLR may act on different page than original action
  - To allow for escrow



## Interactions

- Recovery is traditionally designed with deep awareness of access methods (eg B-trees) and concurrency control
- And vice versa
- Need to handle failure during page split, reobtaining locks for prepared transactions during recovery, etc



## Nested Top Actions

- Trick to support physical operations you do not want to ever be undone
  - Example?
- Basic idea
  - At end of the nested actions, write a dummy CLR
    - Nothing to REDO in this CLR
  - Its UndoNextLSN points to the step before the nested action

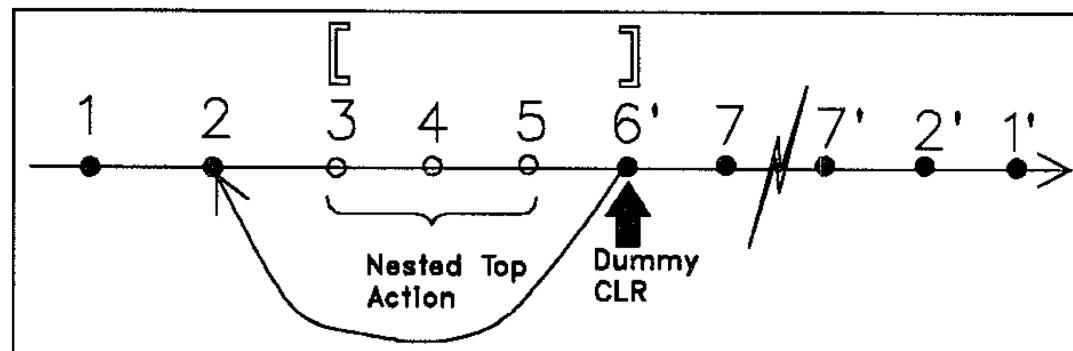


Fig. 14. Nested top action example.



## 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:** A 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 at crash.
- Upon Undo, write CLRs.
- Redo “repeats history”: Simplifies the logic!



## Further Readings

- Repeating History Beyond ARIES,
  - C. Mohan, Proc VLDB'99
  - Reflections on the work 10 years later
- Model and Verification of a Data Manager Based on ARIES
  - D. Kuo, ACM TODS 21(4):427-479
  - Proof of a substantial subset
- A Survey of B-Tree Logging and Recovery Techniques
  - G. Graefe, ACM TODS 37(1), article 1



## Is this a good paper?

- What were the authors' goals?
- What about the performance metrics?
- Did they convince you that this was a good system?
- Were there any red-flags?
- What mistakes did they make?
- Does the system meet the “Test of Time” challenge?
- How would you review this paper today?