

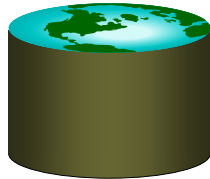
Crash Recovery



R&G - Chapter 20

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.

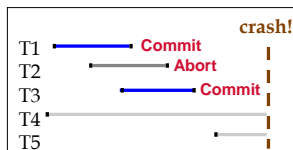
Robert VanNatta,
*Logging History of
Columbia County*



Motivation

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

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



Buffer Mgmt Plays a Key Role

- **Force policy** – make sure that every update is on disk before commit.
 - Provides durability without REDO logging.
 - But, can cause poor performance.
- **No Steal policy** – don't allow buffer-pool frames with **uncommitted** updates to overwrite **committed** data on disk.
 - Useful for ensuring atomicity without UNDO logging.
 - But can cause poor performance.

Of course, there are some nasty details for getting Force/NoSteal to work...



Review: The ACID properties

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

- Question: which ones does the **Recovery Manager** help with?

Atomicity & Durability (and also used for Consistency-related rollbacks)



Assumptions

- **Concurrency control is in effect.**
 - Strict 2PL, in particular.
- **Updates are happening "in place".**
 - i.e. data is overwritten on (deleted from) the actual page copies (not private copies).
- **Can you think of a simple scheme (requiring no logging) to guarantee Atomicity & Durability?**
 - What happens during normal execution (what is the minimum lock granularity)?
 - What happens when a transaction commits?
 - What happens when a transaction aborts?



Preferred Policy: Steal/No-Force

- This combination is most complicated but allows for highest performance.
- **NO FORCE (complicates enforcing Durability)**
 - What if system crashes before a modified page written by a committed transaction makes it to disk?
 - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.
- **STEAL (complicates enforcing Atomicity)**
 - What if the Xact that performed updates aborts?
 - What if system crashes before Xact is finished?
 - Must remember the old value of P (to support UNDOing the write to page P).



Buffer Management summary

	No Steal	Steal		No Steal	Steal
No Force		Fastest	No Force	No UNDO REDO	UNDO REDO
Force	Slowest		Force	No UNDO No REDO	UNDO No REDO
Performance Implications			Logging/Recovery Implications		



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)

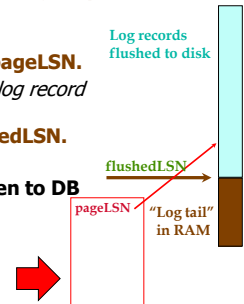
- The **Write-Ahead Logging Protocol**:
 - Must force the log record for an update *before* the corresponding data page gets to disk.
 - Must force all log records for a Xact *before commit*. (I.e. transaction is not committed until all of its log records including its "commit" record are on the stable log.)
- #1 (with **UNDO** info) helps guarantee Atomicity.
- #2 (with **REDO** info) helps guarantee Durability.
- This allows us to implement Steal/No-Force
- Exactly how is logging (and recovery!) done?
 - We'll look at the ARIES algorithms from IBM.



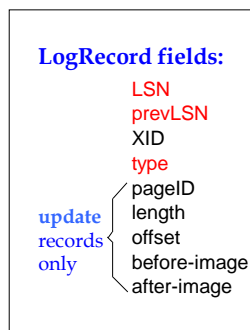
WAL & the Log



- 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 page *i* is written to DB log must satisfy:
 - $pageLSN_i \leq flushedLSN$



Log Records



prevLSN is the LSN of the previous log record written by *this* Xact (so records of an Xact form a linked list backwards in time)

Possible log record types:

- Update, Commit, Abort
- Checkpoint (for log maintenance)
- Compensation Log Records (CLRs)
 - for UNDO actions
- End (end of commit or abort)

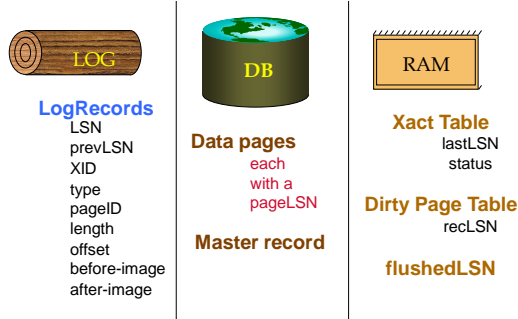


Other Log-Related State

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



The Big Picture: What's Stored Where



Normal Execution of an Xact

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



Transaction Commit

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

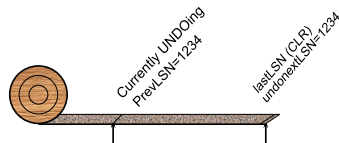


Simple Transaction Abort

- **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.



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: **undoneNextLSN**
 - 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.**

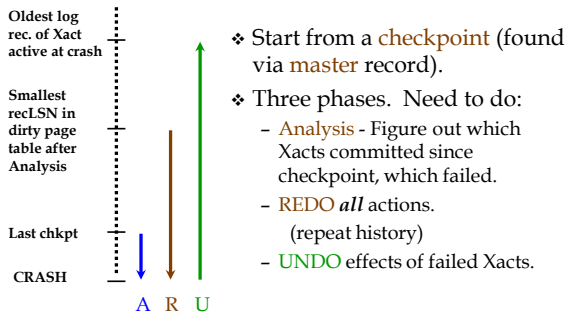


Checkpointing

- **Conceptually, keep log around for all time. Obviously this has performance/implementation problems...**
- **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.
 - Store LSN of most recent chkpt record in a safe place (*master* record).



Crash Recovery: Big Picture



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.
 - All **Other records**: Add Xact to Xact table, set **lastLSN=LSN**, change Xact status on **commit**.
 - also, for **Update** records: If page P not in Dirty Page Table, Add P to DPT, set its **recLSN=LSN**.
- **At end of Analysis...**
 - transaction table says which xacts were active at time of crash.
 - 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 CLR's.
- **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) \geq **LSN**. (this last case requires I/O)
- **To REDO an action:**
 - Reapply logged action.
 - Set **pageLSN** to **LSN**. No additional logging, no forcing!



Phase 3: The UNDO Phase

- **A Naïve solution:**
 - The xacts in the Xact Table are losers.
 - For each loser, perform simple transaction abort.
- Problems?



Phase 3: The UNDO Phase

ToUndo={lastLSNs of all Xacts in the Xact Table}
a.k.a. "losers"

Repeat:

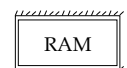
- Choose (and remove) largest LSN among ToUndo.
- If this LSN is a **CLR** and **undonextLSN == NULL**
 - Write an **End** record for this Xact.
- If this LSN is a **CLR**, and **undonextLSN != NULL**
 - Add **undonextLSN** to ToUndo
- Else this LSN is an **update**. Undo the update, write a CLR, add **prevLSN** to ToUndo.

Until ToUndo is empty.

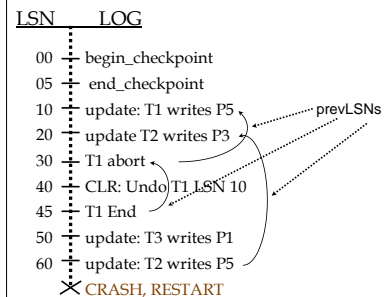
NOTE: This is simply a performance optimization on the naïve solution to do it in one backwards pass of the log!



Example of Recovery

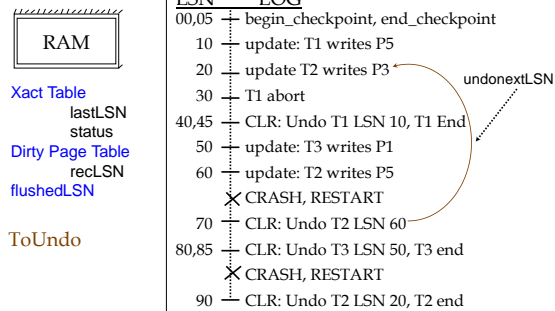


lastLSN
status
recLSN
flushedLSN





Example: Crash During Restart!



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.



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 CLR.**
- **Redo “repeats history”:** Simplifies the logic!