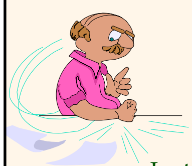


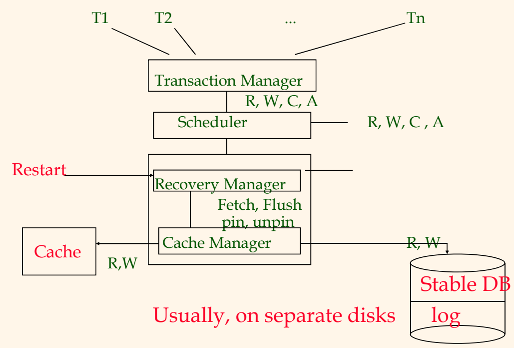
Transaction Management Recovery Chapter 18 (3rd ed)

Instructor: Sharma Chakravarthy

sharma@cse.uta.edu

The University of Texas @ Arlington

ABSTRACT SYSTEM MODEL



Comments

- Scheduler orders the operations so that the execution is serializable and recoverable
- The log contains records of the form [Ti, x, before_val, after_val] – Ti has written value after_val for data item x which had value before_val
- We deal with recovery when the volatile storage fails (abort rollback is easier than recovery)
  - Last committed value of x
  - Committed database state
  - Before (for undo) and after images (for redo) of x

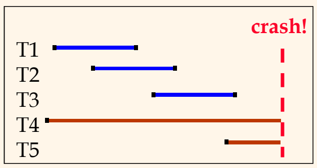
Recoverability

- When is recoverability required
  - When an aborted Tx writes to stable storage (DB)
    - If a system failure occurs at this point, the DB contains the effects of an aborted tx
    - These effects must be undone (UNDO)
  - If a tx commits and does not write all of its updates to the DB
    - This can happen if the buffer has not yet flushed the page to disk!!
    - If a system failure occurs at this point, the DB does NOT contain the effects of the committed tx
    - These effects must be redone (REDO)

Motivation

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

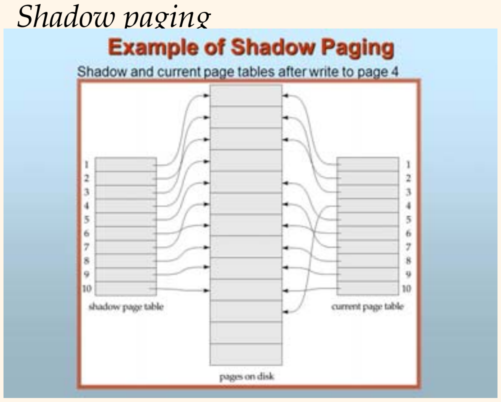
- Desired Behavior after system restarts:
  - T1, T2 & T3 should be durable.
  - T4 & T5 should be aborted (effects not seen).



Recovery Techniques

1. Incremental log with deferred updates (redo / no undo)
   - No writes before commit. Hence no undo. Redo is needed as committed Txs might not have written to disk
2. Incremental log with immediate (in place) updates (redo / undo)
   - If updates are happening in place, uncommitted Txs have written to disk by bm; hence undo.
   - If commit is not atomic, you need redo.
3. Shadow Paging (alternative to log-based crash recovery - no undo/no redo)
   - Pointers are swapped atomically!!

Shadow paging



Recovery

- Tx UNDO: removes all the effects of this Tx only
- Global Undo: when recovering from a system failure, the effects of all incomplete Txs have to be rolled back
- Partial REDO: when recovering from a system failure, results of committed Txs have to be redone (because some of the committed results are still in the buffer)
- Global REDO: archive recovery. Apply all committed Txs to a backup copy to bring it to current state.

Assumptions

- Concurrency control is in effect.
  - Strict 2PL, in particular.
- Updates are happening "in place" (immediate)
  - i.e., data is overwritten on (deleted from) the disk.
  - As we have seen, this is done by the buffer manager and applications/TM do not have any control over it
- Looking for a simple scheme to guarantee Atomicity & Durability?

Recoverability

- When is recoverability required
  - When an aborted Tx writes to stable storage (DB)
    - If a system failure occurs at this point, the DB contains the effects of an aborted tx
    - These effects must be undone (UNDO)
  - If a tx commits and does not write all of its updates to the DB
    - This can happen if the buffer has not yet flushed the page to disk!!
    - If a system failure occurs at this point, the DB does NOT contain the effects of the committed tx
    - These effects must be redone (REDO)

Buffer Management and undo

- Replacement algorithms write dirty pages to disk. This is controlled solely by the buffer manager.
  - If no dirty pages are written back to disk, transaction undo can be limited to buffer operations
- Disadvantage: requires very large buffers.
- Steal: modified buffer pages can be written to disk (due to buffer replacement) at any time (even before commit)
- No Steal: modified pages are kept in buffer at least until the end of the transaction (EOT)
  - Unrealistic; requires large and varying size buffer space!

Buffer Management and Redo

- When a tx commits, all its pages must be written to disk. Otherwise, durability has to be guaranteed in some other way (redo).
  - If all dirty pages are written at the end of Tx (still need to be atomic), no logging is required and hence no redo.
- Disadvantage: requires too many I/O's and waiting for EOT to finish writing.
- Force: All modified buffer pages are written at EOT
  - Commit has to wait for I/O to finish; increases response time
  - System failure can still occur during commit process
- No Force: No writing is triggered at the time of EOT (only decided by the BM)

Handling the Buffer Pool

- Force writes to disk just before commit?
  - Poor response time.
  - But provides durability.
  - No redo
- Steal buffer-pool frames from uncommited Txs?
  - If not, need very large buffers
  - If so, how can we ensure atomicity?
  - undo

	No Steal	Steal
Force	Trivial (if commit is atomic)	Undo/ no redo
No Force	Redo/ no undo	Desired Need both Redo/undo

More on Steal and Force

- **STEAL** (why enforcing Atomicity is hard)
  - *To steal frame F:* Current page in F (say P) may be written to disk; some Xact may hold lock on P.
    - What if the Xact with the lock on P aborts? (locking is different from pinning)
    - Must remember the old value of P at steal time (to support UNDOing the write to page P).
- **NO FORCE** (why enforcing Durability is hard)
  - What if system crashes before a modified page (of a committed Tx) is written to disk?
  - Write as little as possible, in a convenient place (log), at commit time, to support REDOing modifications.

Basic Idea: Logging

- Record REDO and UNDO information, for every update (insert/delete/modify), 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 value, new value>
  - and additional control info (which we'll see soon).

Write-Ahead Logging (WAL)

- The Write-Ahead Logging (WAL) Protocol:
  - Do not flush an uncommitted update to the stable database (or storage) until the log record containing its before image has been flushed to the log.
    - Otherwise, you cannot undo it!!
  - That is, must force the log record for an update *before* the corresponding data page gets to disk.
- Guarantees Atomicity (using undo or rollback)

The Force-at-commit Rule

- The Force-at-Commit Protocol:
  - Do not commit a Tx until the after images of all its updated pages (as log) are in stable storage.
    - Otherwise, cannot do redo!!
  - That is, must write/flush all log records for a Xact *before commit* is declared!
- Guarantees Durability (by doing redo or roll forward)
- NOTE: only write all log records before commit, not data pages!!
  - Log information is much smaller!

Info needed for undo and redo

- All the implementations observe the above two rules to ensure that sufficient (and minimal) information is stored in the log:
  - UNDO Rule (WAL) : log/store the value of x in a stable storage before overwriting it by an uncommitted value (store before image)
    - Each update is logged!
  - REDO Rule (Force-at-Commit) : Before a tx commits, the value it wrote for each data item must be logged/stored in a stable storage (store committed image).

Types of Recovery algorithms

- Both Undo/Redo
  - corresponds to steal/no force
- Undo/no-redo
  - corresponds steal/Force
- No-undo/redo
  - corresponds to no steal/No force
- No-undo/no-redo
  - corresponds to no steal/force
  - Shadow paging (used in early System R)
- Most/all commercial systems use both undo/redo algorithms

Atomicity of Commit

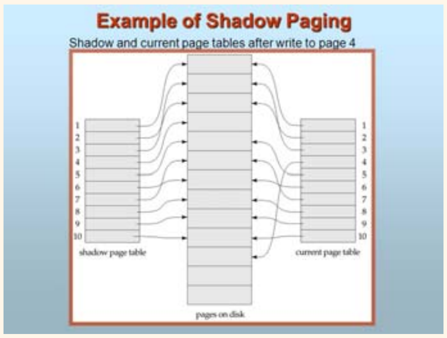
Flush all of T1's updates
Before commit to avoid redo

T1: start …. Write(p) ….. Commit (takes finite amount of time)

Flush all of T1's
Updates only after
Commit to avoid undo

- One of (Undo, Redo) cannot be avoided unless commit is made atomic. That is exactly what shadow paging did by making it an atomic pointers swap. Not realistic for large databases

Shadow paging



Restart

- Restart requires a fair bit of book keeping
  - It needs to know which txs were active at the time of failure – so it can abort them
  - It needs to know which updates of committed Txs were not written to the stable db – so it can redo them
- Moreover, restart must be fault-tolerant. That is, if the system fails when restart is running, it must be possible to re-execute restart (as many times as needed) and get the correct result (DB state)
  - Hence the restart algorithm Must be idempotent
  - property of certain operations in mathematics and computer science, that can be applied multiple times without changing the result beyond the initial application.
- Examples: absolute function, multiplication by 1, max
- Non-idempotent operators: general addition, multiplication

Recovery characteristics

- Recovery should add as little overhead to normal processing as possible
  - Avoid excessive flushing (I/O)
  - Avoid logging too much data
- Recovery should be done quickly
  - So system is down for a short period
  - High availability
- This is even more critical today due to 24/7 availability expectation!

Log Records

LogRecord fields:

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

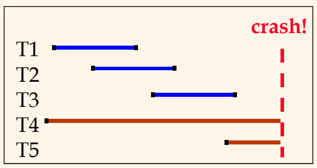
update records only: pageID, length, offset, before-image, after-image

Possible log record types:
- **Update**
- **Commit**
- **Abort**
- **End** (signifies end of commit or abort)

- Each log record has a unique, non-decreasing id
- Both WAL and Force-at-commit are being followed!

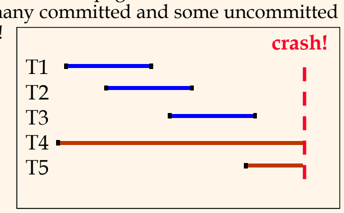
Recovery actions

- In the case of the DB being modified by incomplete Txs (steal policy by BM), to what extent (how far back) does the log have to be processed for UNDO recovery?
- Oldest incomplete transaction



Recovery actions

- If the DBMS does not use a force (no-force by BM) discipline, how much of the log (how far back) has to be processed for redo recovery?
- Depends on when the buffer replaces the page
- You can have a page in the buffer that has been modified by many committed and some uncommitted txs for a long time!



Recovery actions or how much

- Depends on when the buffer replaces the page
- If there is a hot spot, the buffer page for that hot spot will contain changes of many committed transactions.
- Hence redo recovery will have to go back very far in the log (expensive)
- Depends on the interval between crashes
- Higher the availability of the system, the more costly recovery will become.
- Hence, Need checkpoints: to limit the scope of redo
- Note that there is no way to limit scope of undo except to limit Tx to a certain size!

Checkpointing

- Any activity that is done during normal processing to reduce the amount of work to redo after recovery.
- Involves 3 steps
  - Write a begin_checkpoint record to the log file
  - Write all checkpoint data to the log file (shall see soon)
  - Write an end_checkpoint record to the log file
- In fact, SQL allows you to checkpoint a transaction using the CHECKPOINT statement!
- All DBMSs do checkpointing periodically (even if not explicitly specified) to reduce redo

Tx-oriented checkpointing

- FORCE discipline can be seen as checkpointing at the Tx level
  - That is, writing all modified pages at commit!
- Disadvantages:
  - Large number of I/O's at EOT
  - Hotspots have to be written to disk very often.

Tx-consistent checkpointing

- Global in nature
- Creates a transaction consistent database
- Requires all update activity be quiescent
  - All incomplete tx are completed and new ones are not admitted
  - The checkpoint is actually generated when the last update is completed
- After the end_checkpoint record has been written, normal operation resumes
- Stop the pipeline temporarily, empty all Txs, and resume!
- Not realistic!

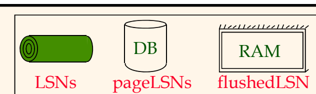
Cache consistent checkpointing

- Also known as action-consistent checkpt
- Stops processing any new operations (temporarily leave an active tx in blocked state)
- Flushes all the dirty pages in the cache/buffer
- Places markers at the end of the log and abort list to indicate that the flushes took place.
- Response time increases, not realistic!

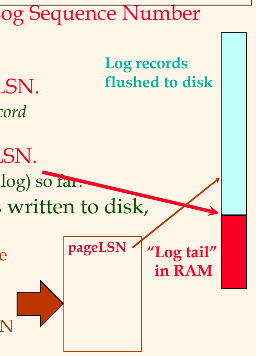
End of log

	ckpt	

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 (written to log) so far.
- WAL protocol: *Before* a page is written to disk,
  - pageLSN ≤ flushedLSN (log record is in stable storage before Flushing the data page)
- Force-at-commit is also in place
  - lastLSN (of that Tx) <= flushedLSN
  - lastLSN: latest log record of a Tx



Goals of ARIES*

- ARIES is an "industrial strength" buffer management and logging/recovery scheme developed at IBM Almaden Research (by C. Mohan and others)
  - Used by most commercial DBMSs today!
  - no constraints on buffer fetch and eviction
    - *steal*
    - support for long-running transactions
  - fast commit
    - *no-force*
  - on-line incremental "fuzzy" checkpointing
    - fully concurrent with automatic log truncation
  - fast recovery, restartable (or idempotent) if the system fails while recovering

*Algorithms for Recovery and Isolation Exploiting Semantics (or ARIES)

Introduction to ARIES

- 1. Every log record is tagged with a monotonically increasing **L**og **S**equence **N**umber (LSN).
  - At recovery, log records can be retrieved efficiently by LSN.
- 2. Keep a *transaction table* in memory, with a record for each active transaction.
  - Keep each transaction's *lastLSN* of its most recent log record.
- 3. Maintain a backward-linked list (in the log) of log records for each transaction (prevLSN)
  - (Write the transaction's current *lastLSN* into each new log record.)
- 4. Each record in the log pertains to exactly one page, whose ID is logged as part of the record.
  - Means cannot have records spanning a page!

Log Records

LogRecord fields:

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

(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

Types of log records

1. Updating a page
   - After modifying a page, an update **record** with before and after image is appended to the log buffer
   - **The pageLSN of the modified page is set to the LSN of the log record**
   - The page must be pinned before modifying the page and unpinned after
2. Commit
   - This log record is written when a commit is encountered/started (not finished)
   - This and prev log records are force written to the log file (force-at-commit rule)
   - Note that commit is not complete until further book keeping actions (e.g., removing Tx from Tx table)

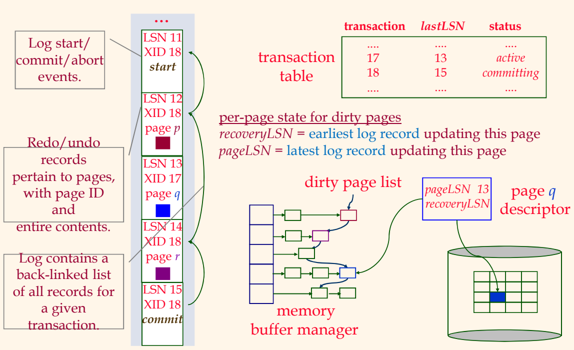
Types of log records (2)

3. Abort
   - This log record is written when a abort is encountered and undo is initiated
   - Remember that abort takes finite amount of time to finish
4. End:
   - When a Tx is aborted and committed, some additional actions are necessary. After these additional actions are done, an end type log record is appended to the log
5. Undoing an update:
   - when a Tx is rolled back (either for abort or for rollback), its updates are undone. When the action described by an update record is undone, a CLR is written. It will have only the before image for redoing it.
     - CLR's are redone but never undone!

Other Log-Related State (*in main memory*)

- Transaction Table (TxT):
  - One entry per active Xact.
  - Contains XID, status (running/committed/aborted), and lastLSN (points to the previous LSN of the SAME transaction)
- Dirty Page Table (DPT):
  - One entry per each dirty page in buffer pool.
  - Contains recoveryLSN – the LSN of the log record which ***first*** caused the page to be dirty (not the same as dirty bit)
- In addition, note that each page has pageLSN – the latest log record that modified that page
  - Useful when it is written to disk; we know that the log was flushed up to that pageLSN before writing the page to disk (dirty page)

ARIES Data Structures

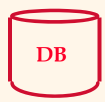


Normal Execution of a Tx

- Series of reads & writes, followed by commit or abort.
  - We will assume that write is atomic on disk.
    - In practice, additional details need to be worked out to deal with non-atomic writes.
    - Some provide atomic writes, Writing and reading back to make sure it is correct!
- Strict 2PL.
- STEAL, NO-FORCE buffer management, with Write-Ahead Logging and force-at-commit.

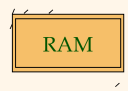
The Big Picture: What's Stored Where

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



Data pages
each
with a
pageLSN

master record
has Last checkpt
Record lsn



Xact Table
- lastLSN
- status

Dirty Page Table
- recLSN

flushedLSN

BM manages pages When in RAM

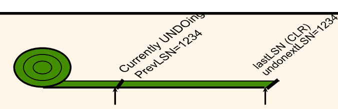
Checkpointing

- 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' bacause:
    - Other Xacts continue to run; so these tables are 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. (So it's a good idea to periodically flush dirty pages to disk!)
  - After writing the end checkpoint record, Store LSN of begin chkpt record in a safe/known place (*master* record). Implication?
- Note the difference between fuzzy checkpointing and others discussed earlier
  - Normal operations continue during checkpointing

Simple Transaction Abort

- For now, consider an explicit abort of a Xact.
  - No crash involved.
  - Memory not lost!!
- We want to "play back" the log in reverse order, UNDOing updates.
  - Before starting UNDO, write an *Abort* log record (not the end record)
    - For recovering from crash during UNDO!
  - Get lastLSN of Xact from Xact table.
  - Follow chain of log records backward via the prevLSN field (until null prevLSN is encountered)

Abort, cont.

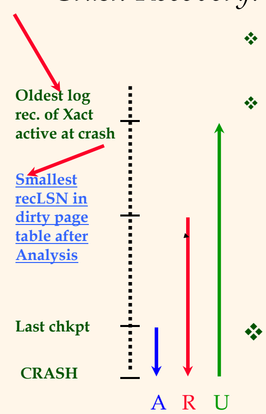


- To perform UNDO, must have a lock on data record!
  - No problem!
  - May need to bring that page from disk! To BM
  - Do you need a lock on the log record page?
  - Do you need to pin it?
- Before restoring old value of a page, write a CLR (has only before image): (we will use this later for redo)
  - You continue logging (CLRs) while you UNDO!!
  - CLRs prevLSN is interpreted as: 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 abort" log record

Transaction Commit

- Write commit record to log. //not to disk
- 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. // commit record is also flushed
- Write end commit record to log.
  - End record need not be flushed!
  - But flushing may save abort of a almost committed Tx
  - Correctness is not affected by not flushing!

Crash Recovery: Big Picture



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

Relative positions of A, R, and U
May be different than shown

Recovery Phases

- Restart-Analysis phase: identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active Txs at the time of crash
  - Starts with the last checkpt record
  - There may be log recs between start and end checkpt
  - Analysis phase may write "end" records if missing for aborted Txs!
- Redo: Repeat all actions, starting from an appropriate point in the log, and restore the db state to what it was at the time of crash
  - Can be skipped if no pages in dirty page table!
- Undo: Undoes the actions of transactions that did not commit, so that the db reflects only the actions of committed transactions. Also writes CLRs
  - Skips logs based on CLR's; CLR's have been handled by redo!

Log Records

LogRecord fields:

prevLSN
XID
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)
- **CLR** (Compensation Log Records)
  - for UNDO actions

Example of a log

Note: we do not write
A begin tx type log record

LSN	LOG
00	update: T1 writes P5
05	update T2 writes P3
10	T2 commit
20	T2 End
30	update: T3 writes P1
40	update: T3 writes P3
	CRASH

T1 and T3 were active at the time of crash. Hence need to be undone; T2 is a committed Tx, and all its actions need to be written to disk; and P1, P3, and P5 are potentially dirty pages

Recovery: The Analysis Phase

- Reconstruct state at checkpoint via end_checkpoint record. That is,
  - From where to do Redo?
  - Dirty pages at the time of crash
  - Txs to be undone (those active at the time of crash)
- Scan log forward from checkpoint.
  - End record: Remove Xact from Xact table (if there)
  - Other records (c, a, u, clr): Add Xact to Xact table,
    - set lastLSN=LSN, change Xact status to C on commit, else u
  - Redoable (Update or CLR) record:
    - If P not in Dirty Page Table (do nothing if it is in DPT)
      - Add P to DPT., set its recLSN=LSN (because we are coming from the ckpoint side; done only once!)
    - P not in DPT means that it was brought into memory after the checkpoint was written
- At the end of the Analysis phase, we have reconstructed Tx table and DPT in memory!

Example of a log

Txt and DPT are assumed empty at checkpoint

DPT

Page	recLSN
P5	00
P3	05
P1	30

Tx Table

Tx	lastLSN	status	
T1	00	u	
T2	05, 10	c	X
T3	30, 40	u	

LSN	LOG
00	update: T1 writes P5
05	update T2 writes P3
10	T2 commit
20	T2 End
30	update: T3 writes P1
40	update: T3 writes P3
	CRASH

T1 and T3 were active at the time of crash. Hence need to be undone; T2 as a committed Tx, and all its actions need to be written to disk; and P1, P3, and P5 are potentially dirty pages

Recovery: The REDO Phase

recLSN = earliest log record updating this page (present only in the main memory DPT table)
pageLSN = latest log record updating this page (present in every page in disk/buffer)

- 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. For each CLR or update log rec LSN, REDO the action unless:
  1. Affected page is not in the Dirty Page Table, or
  2. Affected page is in DPT (and BM) but has recLSN > LSN, or
  3. (in DPT and not in BM)
     1. pageLSN (in DB) ≥ LSN. (page must be retrieved from DB)
- To REDO an action:
  - Reapply logged action.
  - Set pageLSN to LSN of the log record. No additional logging!
- At the end of REDO phase, end type records are written for all Txs with status C, and are removed from the Tx table. All committed Txs have been restored! Why is this true?

Recovery: The UNDO Phase

ToUndo={ *l* | *l* is the lastLSN of a "loser" Xact}
//from the Tx table constructed

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.

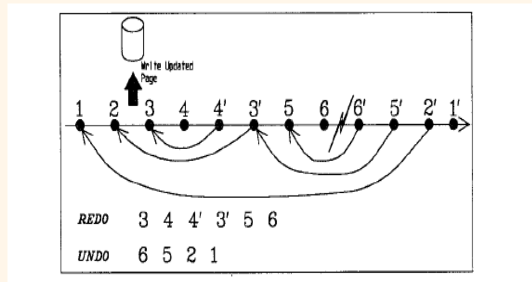


Fig. 13. Restart recovery example with ARIES.

Example of Recovery

Page	recLSN
P5	10
P3	20
P1	50

Tx	lastLSN
~~T1~~	~~30~~
T2	60
T3	50

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 NULL
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART

prevLSNs

UndonextLSN

Example: Crash During Restart!

Page	recLSN
P5	10
P3	20
P1	50

Tx	lastLSN
~~T1~~	~~10~~
T2	70
~~T3~~	~~50~~

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	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

Evaluating ARIES

- The ARIES logging/recovery algorithm has several advantages over other approaches:
  - *steal/no-force* with few constraints on buffer management
    - Steals act as incremental, nonintrusive checkpoints.
  - synchronous "fuzzy" checkpoints are fast and nonintrusive
  - minimizes recovery work
    - makes forward progress in failures during recovery
  - *repeating history* redo supports logical undo logging and alternative locking strategies (e.g., fine-grained locking)
- But: ARIES requires WAL with undos, LSNs written with every page, and redo records restricted to a single page.

Additional Crash Issues (idempotency)

- What happens if system crashes

  During Analysis?
  - nothing has changed; do analysis again!

  During REDO?
  - things might have changed on disk; But we are going to REDO and restore it to the correct consistent state!!

  During UNDO
  - Whatever undo has been done they are NOT repeated again. Instead, REDO takes care of that. A successful undo is done only once!!!
- 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?
  - Cannot, except avoiding long-running Xacts.

Summary of Logging/Recovery

- Recovery Manager guarantees Atomicity & Durability.
- Use WAL to allow STEAL/NO-FORCE without sacrificing correctness.
- LSNs identify log records; linked backwards for each Tx (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.
- While performing Undo, write CLRs.
- Redo "repeats history": Simplifies the logic!

Thank You !