CS54100: Database Systems

Failure & Recovery

9 April 2012

Prof. Chris Clifton

Recovery

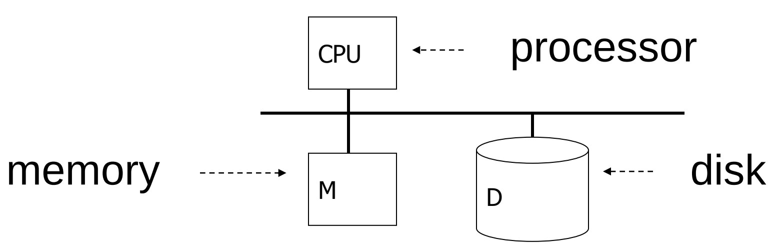
- First order of business:
  Failure Model

Events — Desired

Events — Undesired — Expected

Undesired — Unexpected

Our failure model



Desired events: see product manuals….

Undesired expected events:

System crash

- memory lost
- cpu halts, resets

that's it!!

Undesired Unexpected: Everything else!

Undesired Unexpected: Everything else!

Examples:

- Disk data is lost
- Memory lost without CPU halt
- CPU implodes wiping out universe….

Is this model reasonable?

Approach: Add low level checks + redundancy to increase probability model holds

E.g.,
- Replicate disk storage (stable store)
- Memory parity
- CPU checks

Review: The ACID properties

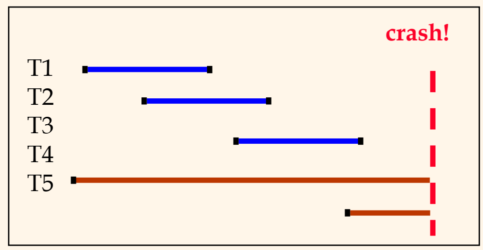
- **A** tomicity: All actions in the Xact happen, or none happen.
- **C** onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one Xact is isolated from that of other Xacts.
- **D** urability: If a Xact commits, its effects persist.

- The **Recovery Manager** guarantees Atomicity & Durability.

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 23

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).



Assumptions

- Concurrency control is in effect.
  - Strict 2PL, in particular.
- Updates are happening “in place”.
  - i.e. data is overwritten on (deleted from) the disk.
- A simple scheme to guarantee Atomicity & Durability?

Handling the Buffer Pool

- Force every write to disk?
  - Poor response time.
  - But provides durability.
- Steal buffer-pool frames from uncommited Xacts?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?

	No Steal	Steal
Force	Trivial	
No Force		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 Xact holds lock on P.
    - What if the Xact with the lock on P aborts?
    - 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 is written to disk?
  - Write as little as possible, in a convenient place, at commit time,to support REDOing modifications.

Operations:

- Input (x): block with x $\rightarrow$ memory
- Output (x): block with x $\rightarrow$ disk

  - Read (x,t): do input(x) if necessary
    t $\leftarrow$ value of x in block
  - Write (x,t): do input(x) if necessary
    value of x in block $\leftarrow$ t

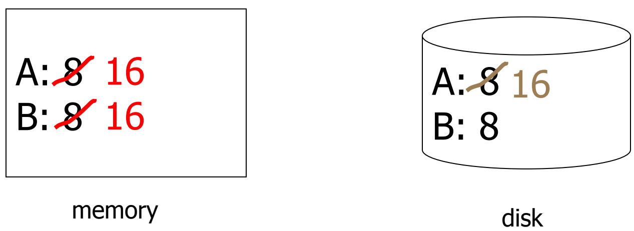
Key problem Unfinished transaction

Example Constraint: A=B

T_1: A $\leftarrow$ A $\times$ 2
B $\leftarrow$ B $\times$ 2

T1: Read (A,t); t ← t×2
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);

failure!



- Need <u>atomicity:</u> execute all actions of a transaction or none at all

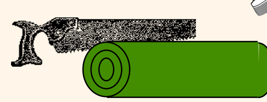
One solution: undo logging (immediate modification)

due to: Hansel and Gretel, 782 AD

- Improved in 784 AD to durable undo logging

(Okay, Ariadne deserves earlier credit)

Basic Idea: Logging



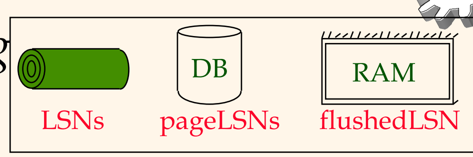
- Record REDO and UNDO information, for every update, in a *log*.
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.
- Log: An ordered list of REDO/UNDO actions
  - Log record contains:
    <XID, pageID, offset, length, old data, new data>
  - and additional control info (which we'll see soon).

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 33

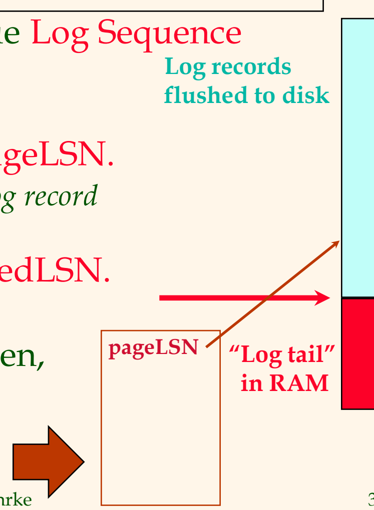
Write-Ahead Logging (WAL)

- 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 Xact *before commit*.
- #1 guarantees Atomicity.
- #2 guarantees Durability.
- Exactly how is logging (and recovery!) done?
  - We'll study the ARIES algorithms.

WAL & the Log

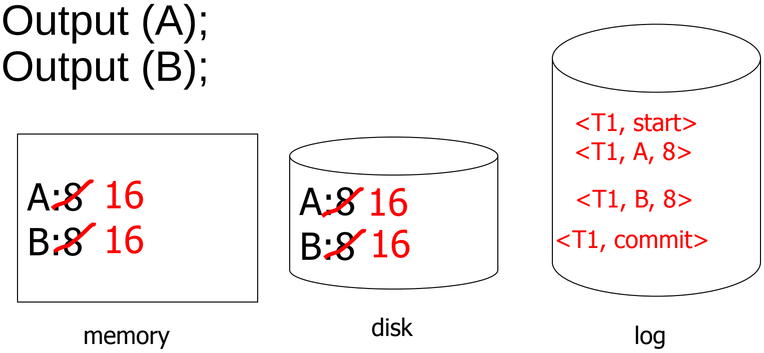


- Each log record has a unique Log Sequence Number (LSN).
  - LSNs always increasing.
- Each *data page* contains a pageLSN.
  - The LSN of the most recent *log record* for an update to that page.
- System keeps track of flushedLSN.
  - The max LSN flushed so far.
- WAL: *Before* a page is written,
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



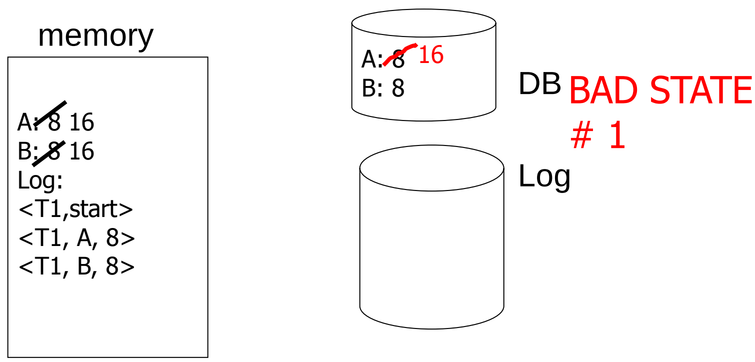
Undo logging (Immediate modification)

T1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);



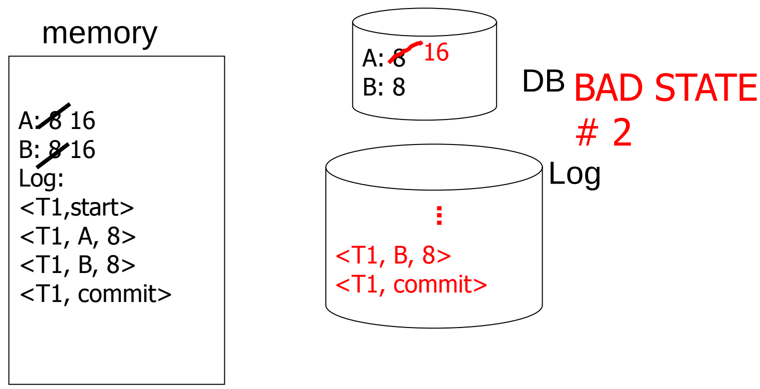
One "complication"

- Log is first written in memory
- Not written to disk on every action



One "complication"

- Log is first written in memory
- Not written to disk on every action



Undo logging rules

(1) For every action generate undo log record (containing old value)

(2) Before *x* is modified on disk, log records pertaining to *x* must be on disk (write ahead logging: WAL)

(3) Before commit is flushed to log, all writes of transaction must be reflected on disk

Recovery rules: Undo logging

- For every Ti with <Ti, start> in log:
  - If <Ti,commit> or <Ti,abort> in log, do nothing
  - Else
    - For all <Ti, *X*, *v*> in log:
      - write *(X, v)*
      - output (*X*)
    - Write <Ti, abort> to log

☒IS THIS CORRECT??

Recovery rules: Undo logging

(1) Let S = set of transactions with <Ti, start> in log, but no <Ti, commit> (or <Ti, abort>) record in log

(2) For each <Ti, X, v> in log, in reverse order (latest $\rightarrow$ earliest) do:
- if Ti $\in$ S then
  - write (X, v)
  - output (X)

(3) For each Ti $\in$ S do
- write <Ti, abort> to log

What if failure during recovery?

No problem! Undo idempotent

Log Records

LogRecord fields:

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

(pageID through 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

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 43

Other Log-Related State

- Transaction Table:
  - One entry per active Xact.
  - Contains XID, status (running/commited/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 an Xact

- 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.
- STEAL, NO-FORCE buffer management, with Write-Ahead Logging.

To discuss:

- Redo logging
- Undo/redo logging, why both?
- Real world actions
- Checkpoints
- Media failures

PURDUE UNIVERSITY

CS54100: Database Systems

Failure & Recovery

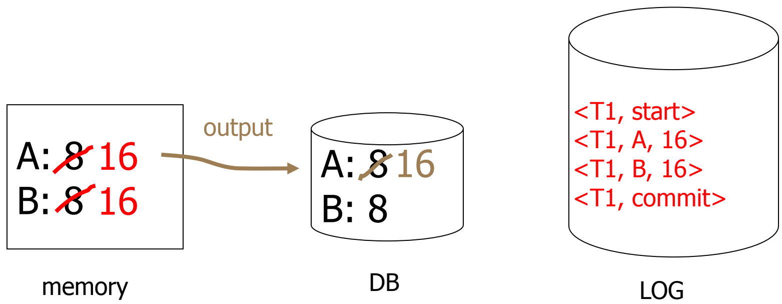
11 April 2012

Prof. Chris Clifton

Indiana Center for Database Systems™

Redo logging (deferred modification)

T1: Read(A,t); $t \leftarrow t \times 2$; write (A,t);
Read(B,t); $t \leftarrow t \times 2$; write (B,t);
Output(A); Output(B)



Redo logging rules

(1) For every action, generate redo log record (containing new value)

(2) Before X is modified on disk (DB), all log records for transaction that modified X (including commit) must be on disk

(3) Flush log at commit

Recovery rules: Redo logging

- For every Ti with <Ti, commit> in log:
  - For all <Ti, X, v> in log:

    { Write(X, v)
    { Output(X)

☒IS THIS CORRECT??

Recovery rules: Redo logging

(1) Let S = set of transactions with
<Ti, commit> in log

(2) For each <Ti, X, v> in log, in forward
order (earliest $\rightarrow$ latest) do:
- if Ti $\in$ S then { Write(X, v)
{ Output(X) ← optional

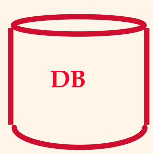
Checkpointing

- 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. (So it's a good idea to periodically flush dirty pages to disk!)
  - Store LSN of chkpt record in a safe place (*master* record).

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 53

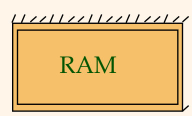
The Big Picture: What's Stored Where

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



Data pages
each with a pageLSN

master record



Xact Table
- lastLSN
- status

Dirty Page Table
- recLSN

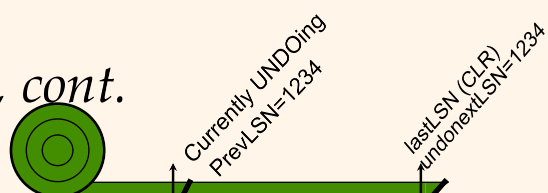
flushedLSN

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 54

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.



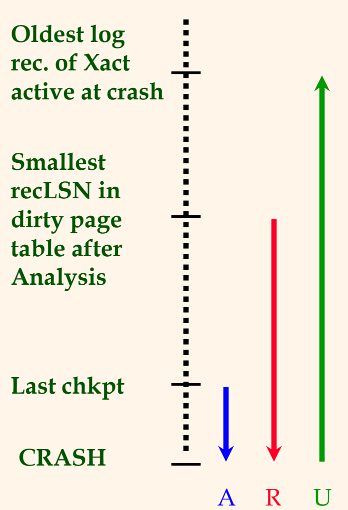
- To perform UNDO, must have a lock on data!
  - No problem!
- Before restoring old value of a page, write a CLR:
  - You continue logging while you UNDO!!
  - CLR has one extra field: undonextLSN
    - Points to the next LSN to undo (i.e. the prevLSN of the record we’re currently undoing).
  - CLRs *never* Undone (but they might be Redone when repeating history: guarantees Atomicity!)
- At end of UNDO, write an “end” log record.

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.

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 57

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.

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 58

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.

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 59

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!

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 60

Recovery: The UNDO Phase

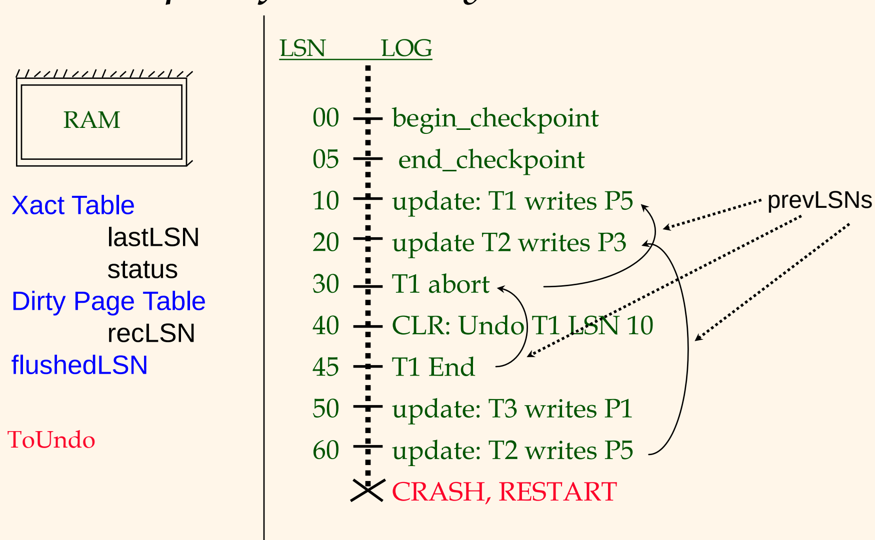
ToUndo={ *l* | *l* a lastLSN of a "loser" Xact}

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



Example: Crash During Restart!

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	CLR: Undo T2 LSN 20, T2 end

undonextLSN

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 63

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.

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 64

Summary of Logging/Recovery

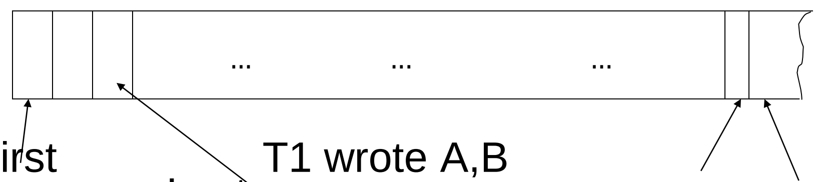
- Recovery Manager guarantees Atomicity & Durability.
- Use WAL to allow STEAL/NO-FORCE w/o sacrificing correctness.
- LSNs identify log records; linked into backwards chains per transaction (via prevLSN).
- pageLSN allows comparison of data page and log records.

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!

Recovery is very, very SLOW !

Redo log:



First Record (1 year ago)

Last Record

T1 wrote A,B

Committed a year ago

--> STILL, Need to redo after crash!!

Solution: Checkpoint (simple version)

Periodically:

(1) Do not accept new transactions

(2) Wait until all transactions finish

(3) Flush all log records to disk (log)

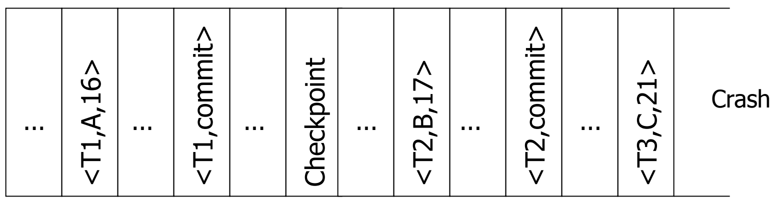
(4) Flush all buffers to disk (DB) (do not discard buffers)

(5) Write "checkpoint" record on disk (log)

(6) Resume transaction processing

Example: what to do at recovery?

- Redo log (disk):



Key drawbacks:

- *Undo logging:* cannot bring backup DB copies up to date
- *Redo logging:* need to keep all modified blocks in memory until commit

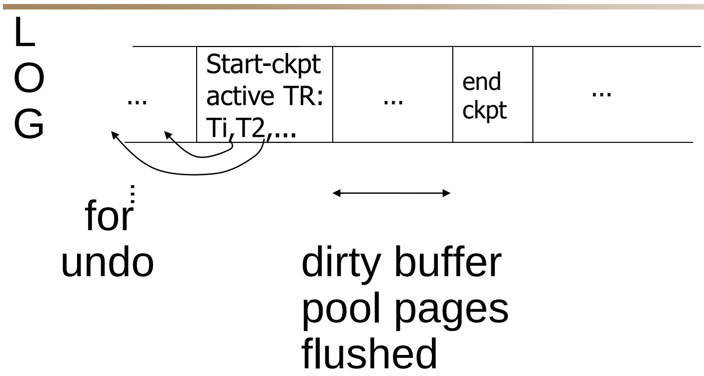
Solution: undo/redo logging!

Update ⇒ <Ti, Xid, New X val, Old X val>
page X

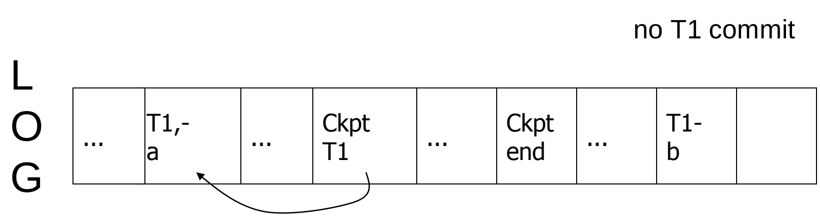
Rules

- Page X can be flushed before or
  after Ti commit
- Log record flushed before corresponding updated page (WAL)
- Flush at commit (log only)

Non-quiesce checkpoint

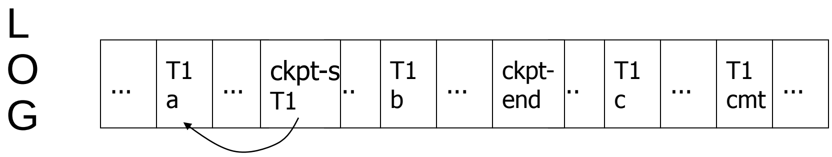


Examples what to do at recovery time?



☒ Undo T_1 (undo a,b)

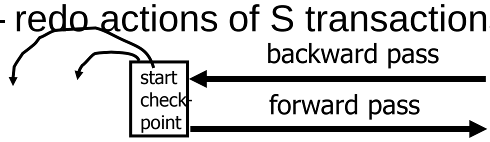
Example



- ☒ Redo T1: (redo b,c)

Recovery process:

- Backwards pass (end of log ➲ latest checkpoint start)
  - construct set S of committed transactions
  - undo actions of transactions not in S
- Undo pending transactions
  - follow undo chains for transactions in (checkpoint active list) - S
- Forward pass (latest checkpoint start ➲ end of log)
  - redo actions of S transactions



Real world actions

E.g., dispense cash at ATM

$T_i = a_1 \; a_2 \ldots\ldots a_j \ldots\ldots a_n$

↓

$

Solution

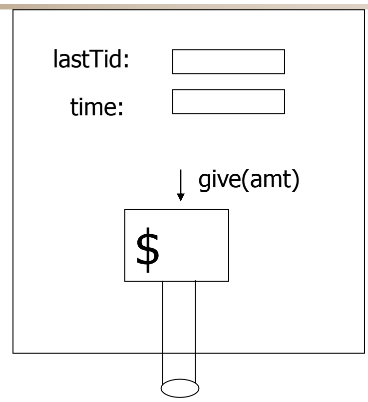
(1) execute real-world actions after commit

(2) try to make idempotent

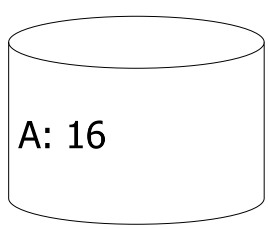
ATM

Give$$

(amt, Tid, time) →



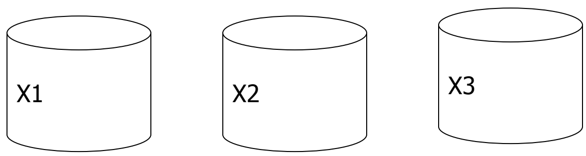
Media failure (loss of non-volatile storage)



Solution: Make copies of data!

Example 1 Triple modular redundancy

- Keep 3 copies on separate disks
- Output(X) --> three outputs
- Input(X) --> three inputs + vote

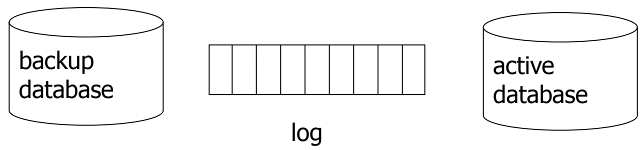


Example #2 Redundant writes, Single reads

- Keep N copies on separate disks
- Output(X) --> N outputs
- Input(X) --> Input one copy
  - if ok, done
  - else try another one

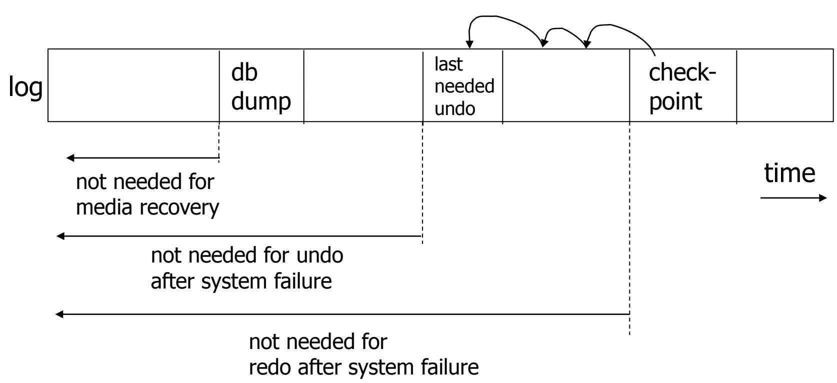
⇔Assumes bad data can be detected

Example #3: DB Dump + Log



- If active database is lost,
  - restore active database from backup
  - bring up-to-date using redo entries in log

When can log be discarded?



More on transaction processing

Topics:

- Cascading rollback, recoverable schedule
- Deadlocks
  - Prevention
  - Detection
- View serializability
- Distributed transactions
- Long transactions (nested, compensation)

Concurrency control & recovery

Example:

T_j	T_i
⋮	⋮
$W_j(A)$	⋮
	$r_i(A)$
⋮	Commit T_i
⋮	
	⋮
Abort T_j	

☛ Cascading rollback (Bad!)

- Schedule is conflict serializable
- $T_j \longrightarrow T_i$
- But not recoverable

- Need to make "final' decision for each transaction:
  - **commit decision** - system guarantees transaction will or has completed, no matter what
  - **abort decision** - system guarantees transaction will or has been rolled back (has no effect)

To model this, two new actions:

- C_i - transaction T_i commits
- A_i - transaction T_i aborts

Back to example:

T_j	T_i
⋮	⋮
$W_j(A)$	
	$r_i(A)$
⋮	
	⋮
	C_i ← can we commit here?

Definition

T_i reads from T_j in S ($T_j \Rightarrow_S T_i$) if

(1) $w_j(A) <_S r_i(A)$

(2) $a_j \not<_S r_i(A)$ ($\not<$: does not precede)

(3) If $w_j(A) <_S w_k(A) <_S r_i(A)$ then
$a_k <_S r_i(A)$

Definition

Schedule S is recoverable if
whenever $T_j \Rightarrow_S T_i$ and $j \neq i$ and $C_i \in S$
then $C_j <_S C_i$

Note: in transactions, reads and writes precede commit or abort

⬄ If $C_i \in T_i$, then $r_i(A) < C_i$

$w_i(A) < C_i$

⬄ If $A_i \in T_i$, then $r_i(A) < A_i$

$w_i(A) < A_i$

- Also, one of C_i, A_i per transaction

How to achieve recoverable schedules?

⇔ With 2PL, hold write locks to commit (strict 2PL)

T_j	T_i
⋮	⋮
$W_j(A)$	⋮
⋮	⋮
C_j	⋮
$u_j(A)$	
	$r_i(A)$
⋮	

Fall 2007 Chris Clifton - CS541 98

⇔ With validation, no change!

Fall 2007 Chris Clifton - CS541 99

- S is <u>recoverable</u> if each transaction *commits* only after all transactions from which it read have committed.
- S <u>avoids cascading rollback</u> if each transaction may *read* only those values written by committed transactions.