PURDUE UNIVERSITY

CS54200: Distributed Database Systems

Recovery

30 January 2009

Prof. Chris Clifton

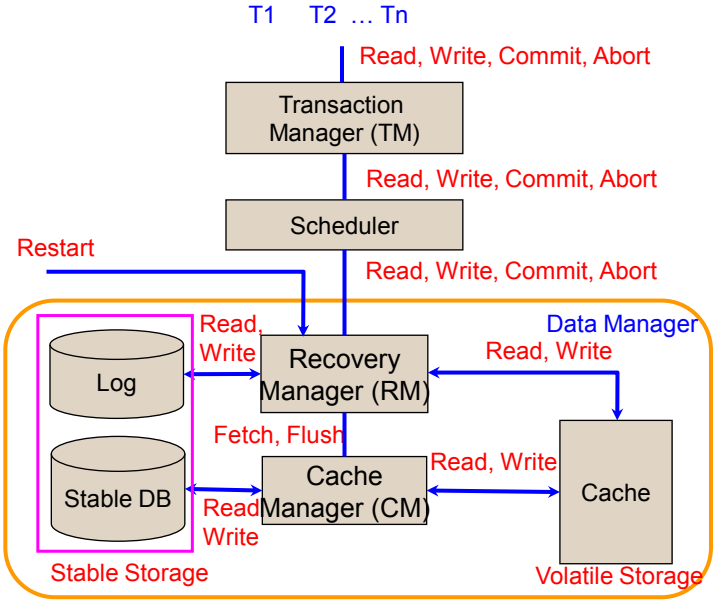
Indiana Center for Database Systems

Recovery

- The DBMS must be able to withstand various types of failures while ensuring Atomicity and Durability of transactions.
- The DBMS runs on a computer with volatile (RAM) and stable storage (disks).
- There are three types of failures in centralized systems:
  - *Transaction* Failures – txn aborts (semantic)
  - *System* Failures – loss of volatile data
  - *Media* Failures – loss of stable storage

2

DM Architecture



3

Recovery

- We will focus on system failures.
- Following the failure, the DBMS is restarted.
- At the start of recovery, the contents of *volatile storage are discarded.*
- The stable storage is potentially inconsistent
- A CONSISTENT database state corresponding to exactly the set of txns that had committed (as far as the DM is concerned) must be reconstructed, i.e. C(*H*).
- This reconstruction uses only data in stable storage – Stable DB and the LOG.

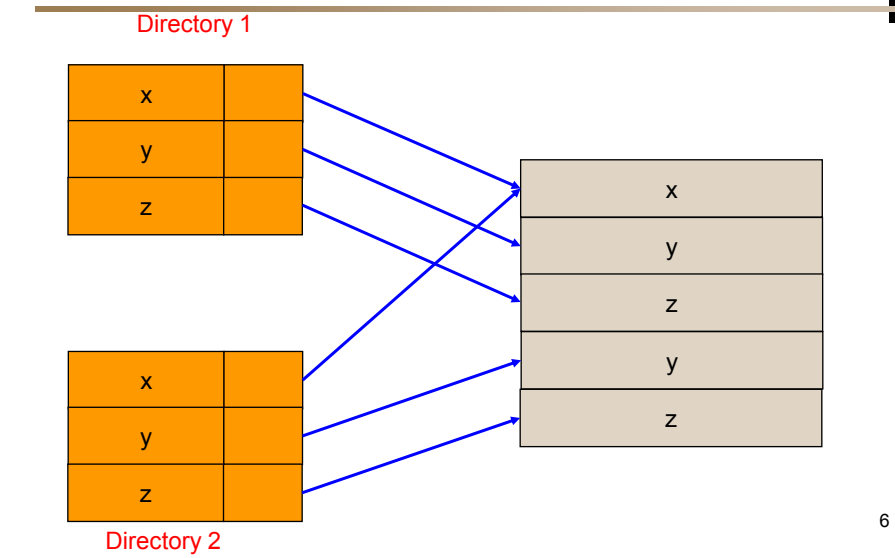
4

Assumptions

- Writes to stable storage by CM are *atomic*.
- Disk granularity matches data granularity.
- Scheduler ensures a strict serializable execution.
- In-place updating: there is only one copy of each data item on secondary storage.
- Shadowing: multiple copies are stored, along with a directory that points to the appropriate copy.
- The stable DB is the one pointed to by a special directory.

5

Shadowing



6

Cache

Slot Number	Dirty Bit	Data Value
1	1	"xcvx"
2	0	98798
…	…	…

Cache

Data Item	Slot Number
x	2
y	1
…	…

Cache Directory

7

Cache Manager

- Has the following operations:
  - *Flush* cache slot I
  - *Fetch* item (page) x
  - *Pin* a slot
  - *Unpin* a slot
- Several cache replacement algorithms can be used
  - LRU
  - FIFO
  - Etc.

8

Recovery Manager

- The RM must execute *read, write, commit*, and *abort* operations atomically! I.e. equivalent to a serial execution of these operations.
- Strict, serializable execution implies that the last committed value of any data item is given by the last value written by a committed txn.
- Also, we use before-images and after-images:
  - The before image of x wrt T_i is the value of x just before T_i wrote into it;
  - The after image of x wrt T_i is the value of x written by T_i.

9

Log

- The log is a sequential record on stable memory of the execution maintained by the recovery manager to ensure that it can recover from system failures.
- Physical logging: record modified bits $[T_i,x,v]$
- Entries are in the same order as the execution of operations.
- Logical logging: may be cheaper to record just the logical operation – has other difficulties (more later).
- RM also maintains *commit, abort* and *active* lists.

10

PURDUE UNIVERSITY

CS54200: Distributed Database Systems

Recovery

2 February 2009

Prof. Chris Clifton

Indiana Center for Database Systems

Recovery Methods

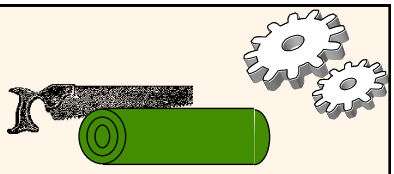
One solution: undo logging (immediate modification)

due to: Hansel and Gretel, 782 AD

- Improved in 784 AD to durable undo logging

(Be thorough in your literature search - Ariadne deserves earlier credit)

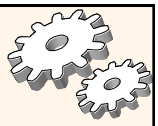
Basic Idea: Logging



- 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 13

Write-Ahead Logging (WAL)

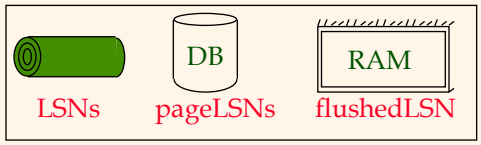


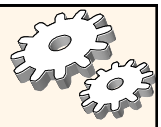
- The Write-Ahead Logging Protocol:
  1. Must force the log record for an update *before* the corresponding data page gets to disk.
  2. Must write all log records for a Xact *before commit*.
- #1 guarantees Atomicity.
- #2 guarantees Durability.

- Exactly how is logging (and recovery!) done?
  - We'll study the ARIES algorithms.

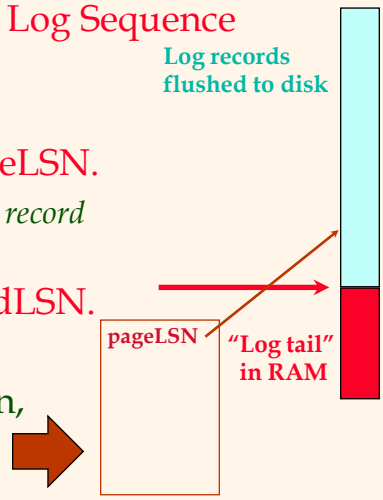
Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 14

WAL & the Log





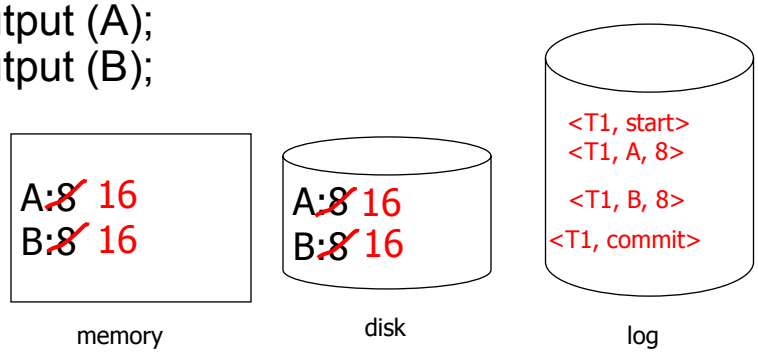
- Each log record has a unique Log Sequence Number (LSN).
  - LSNs always increasing.
- Each *data page* contains a pageLSN.
  - The LSN of the most recent *log record* for an update to that page.
- System keeps track of flushedLSN.
  - The max LSN flushed so far.
- WAL: *Before* a page is written,
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



Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke 15

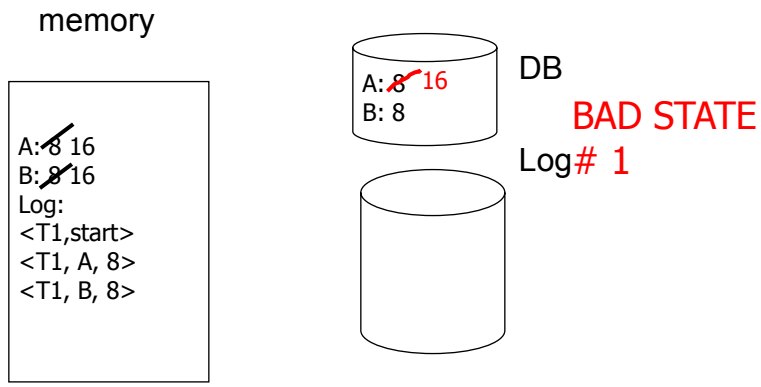
Undo logging (Immediate modification)

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);



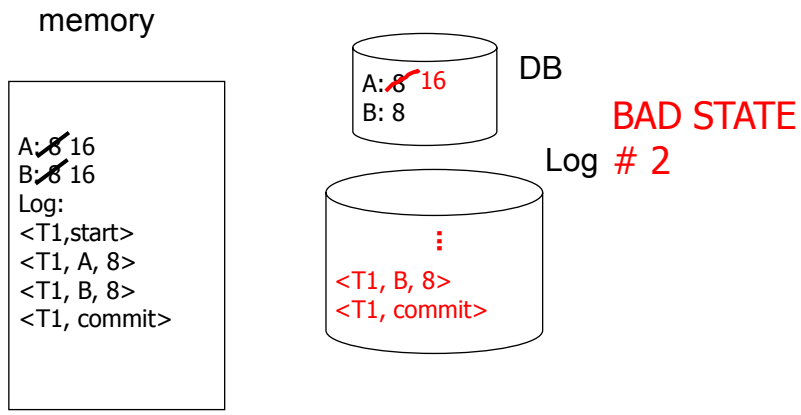
One "complication"

- 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

Fall 2007 Chris Clifton - CS541

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 → earliest) do:

- if Ti ∈ S then
  - write (X, v)
  - output (X)

(3) For each Ti ∈ S do

- write <Ti, abort> to log

What if failure during recovery?

No problem! ☞Undo idempotent

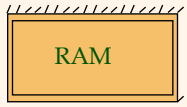
The Big Picture: What's Stored Where

LogRecords

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

Data pages
each
with a
pageLSN

master record



Xact Table
lastLSN
status

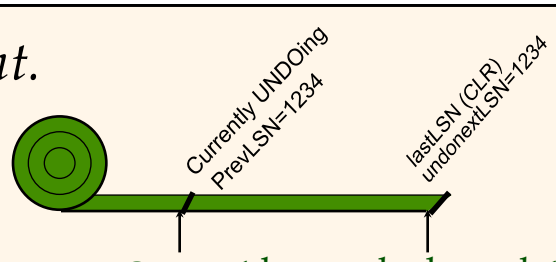
Dirty Page Table
recLSN

flushedLSN

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.



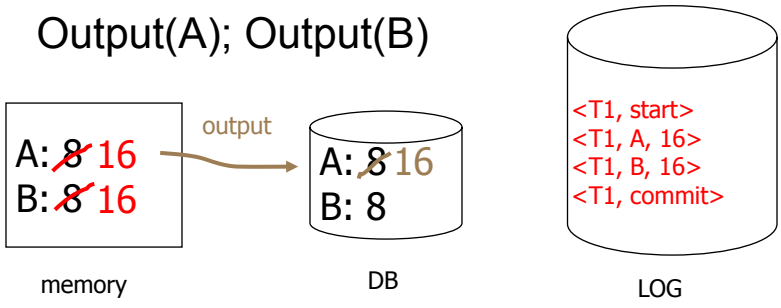
- To perform UNDO, must have a lock on data!
  - No problem!
- Before restoring old value of a page, write a CLR:
  - You continue logging while you UNDO!!
  - CLR has one extra field: undonextLSN
    - Points to the next LSN to undo (i.e. the prevLSN of the record we're currently undoing).
  - CLRs *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.

Redo logging (deferred modification)

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

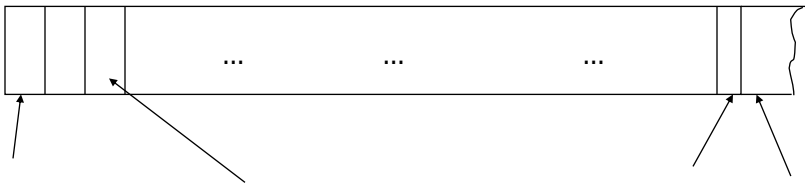
(1) Let S = set of transactions with <Ti, commit> in log

(2) For each <Ti, X, v> in log, in forward order (earliest → latest) do:

- if Ti ∈ S then Write(X, v)
  Output(X) ← optional

Recovery is very, very SLOW !

Redo log:



First Record (1 year ago) | T1 wrote A,B Committed a year ago | Last Record

→ 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):

...	<T1,A,16>	...	<T1,commit>	...	Checkpoint	...	<T2,B,17>	...	<T2,commit>	...	<T3,C,21>	Crash

Key drawbacks:

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

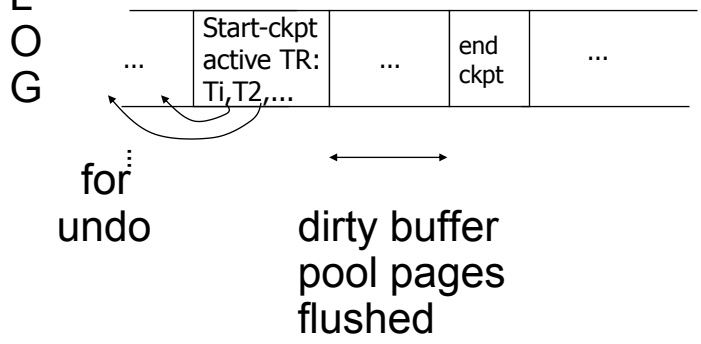
Solution: undo/redo logging!

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)

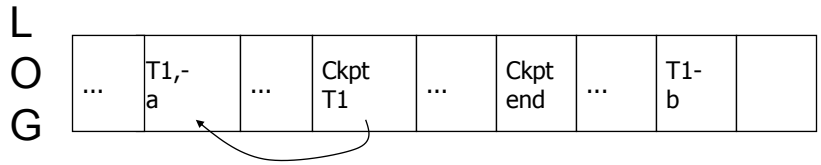
Non-quiesce checkpoint



Fall 2007 Chris Clifton - CS541

Examples what to do at recovery time?

no T1 commit

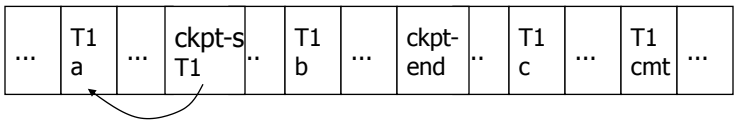


☒ Undo T_1 (undo a,b)

Fall 2007 Chris Clifton - CS541

Example

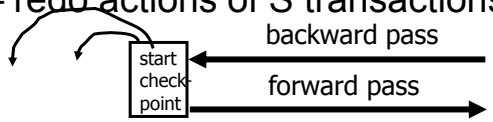
LOG



☒ 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



PURDUE UNIVERSITY

CS54200: Distributed Database Systems

Recovery

4 February 2009

Prof. Chris Clifton

Indiana Center for Database Systems

Fuzzy Checkpointing

- Cache consistent checkpoint can be further improved by reducing the amount of data flushed.
- Flush only those slots that have not been flushed since the previous (penultimate) checkpoint.
- Expect that normal cache replacement will have more time to flush dirty cache slots ➔ quicker checkpoint.
- Of course, now recovery has to go back a little further!

59

Recovery

- Restart redoes just those updates of txns in commit list that come after the penultimate checkpoint
- Undoes the updates of those txns that are either in the active (but not commit) list, or are in the abort list and follow the penultimate checkpoint marker in the abort list.

60

Idempotence of Restart

- Recovery must be resilient to failure too.
- I.e. if there is a failure during recovery, when we start recovery again, the recovered state should be no different than if the second failure had not taken place.
- Therefore, the restart procedure should not modify the stable DB or log in a manner that will affect subsequent recovery – even temporarily.

66

Optimizations

- To reduce the amount of work restart has to do, we can avoid certain undo/redo operations.
- During backward scan for *[Ti,x,v]* in *AL,* need not undo this operation if:
  - A1: *Ti*'s abort lies between CKPT and CKPT', but *x* is not among the dirty items at CKPT.
  - A2: *Ti*'s abort record lies between CKPT and CKPT', and *x* was in a dirty cache slot at CKPT, but its stable-*LSN* (saved in CKPT) is greater than the *LSN* of *Ti*'s abort record.

94

Optimizations

- In the forward scan, *[T_i,x,v]*, where T_i is in *CL,* need not be redone if:
  - C1: T_i's update record lies between CKPT and CKPT', but *x* is not in the list of dirty cache slots at CKPT; or
  - C2: T_i's update record lies between CKPT and CKPT', *x* is in the list of data items in dirty cache slots at CKPT, but its stable-*LSN* is greater than the *LSN* of the update record at hand.
- RM can improve performance in case of multiple failures by appending two CKPT records at the end of recovery. No work needs to be done upon successive recovery unless new operations have been processed.

95

Logical Logging

- Logical logging can significantly reduce the amount of storage needed for the log, e.g. *add entry 5 to B-tree, adding a record to a file*.
- We must be able to log, undo, and redo each logical operation.
- However, multiple undo/redo of logical operations are not equivalent – we must be careful!!
- Could be solved by implementing undo/redo such that they are idempotent – not always possible.

96

Alternative 2

- Another alternative is to save a copy of the stable DB at the last checkpoint.
- Restart essentially replays operations from the checkpoint:
  - It begins with the checkpoint DB state, and
    - Undoes all updates that precede the CKPT, but were by txns that were active at CKPT and did not commit;
    - Redoes update records that follow CKPT by committed txns.
  - From strictness this is exactly what we want.
- IBM system R used this with shadowing.

97

Alternative 3

- Another solution is to save *LSN*s in the stable DB.
- Each data item on stable DB saves the *LSN* of the last update applied to it by an active or committed txn.
- For performance, we chain back the updates for each data item in the log (as well as for each txn).
- New algo: LSN-based logical logging.
- Assume logical logging with fuzzy checkpointing; strict executions;

98

LSN-based Logical Logging

- RM-Write:
  - Create an update record, *U*
  - *s*ave current *LSN(x)* in *U*
  - Update *x*, set *LSN(x)* ← *LSN(U)*.
- RM-Abort:
  - Upon undo for record *U* restore *LSN(x)* ← *prev LSN(x)* saved in *U*.
- Restart scans back from the end of the log for undo, and fwd from CKPT' to redo, as before.

99

Backward Scan

- In backward scan of restart, when dealing with an update record *U* by T_i (aborted) for *x:*
  - Fetch *x* and examine *LSN(x)*
  - If *LSN(x)=LSN(U)* (*x* is in the same state after U was applied), then undo U, set *LSN(x)* to the value stored in *U*.
  - If *LSN(x) < LSN(U)* (*x* does not contain *U*'s update) – do not undo *U*.
  - If *LSN(x) > LSN(U): x* contains a later update (*V*) – which was not undone, so *V* follows *U* and must have committed (o/w *LSN(x)* must have been set to *LSN(U)* as above). By strictness, *U* must have been undone before *V* was applied, thus no need to undo *U*.

100

Forward Scan

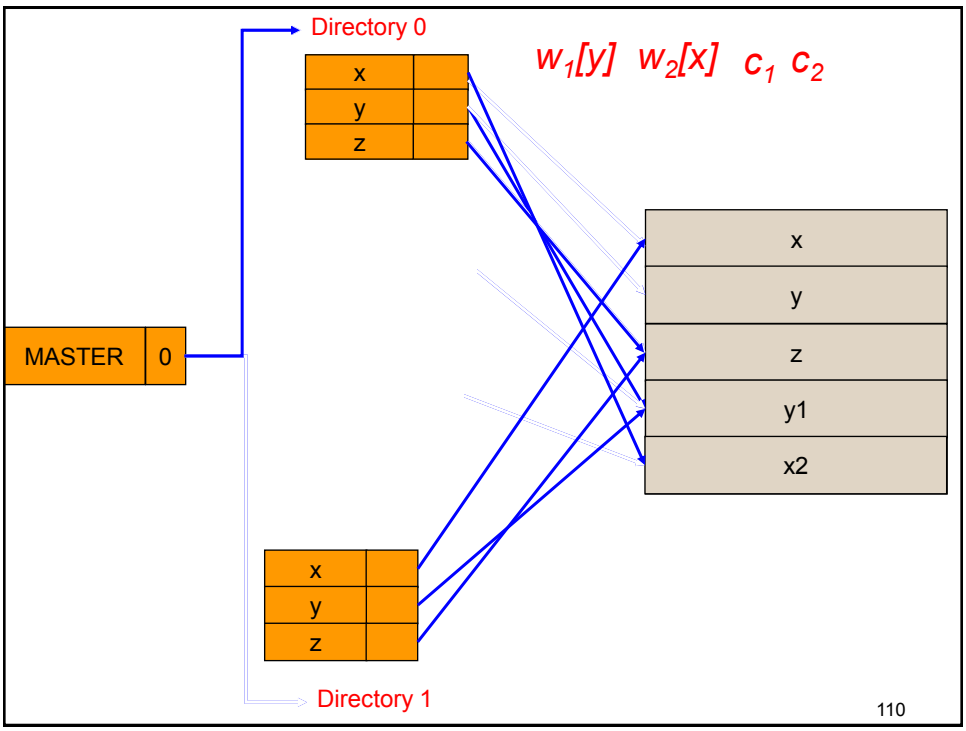
- Backward scan ends as with physical logging.
- Forward scan begins at CKPT', and processes each update record *U* of committed txns.
  - If *LSN(x) < LSN(U),* then *U* hasn't been applied ➔ redo *U*
  - If *LSN(x) = LSN(U),* then *U* has been applied, need not redo.
  - If *LSN(x) > LSN(U),* then a later committed (cannot be an aborted or active txn – why?) update has been applied, so no need to redo.
- *LSN*s in stable DB can help with physical logging too.

101

No-Undo/No-Redo RM!!

- We want to ensure that:
  - Updates of committed txns are stable before commit
  - Updates of uncommitted txns are not!
- I.e. we want the commit operation to atomically insert all updates of a txn to the stable DB *and* mark the txn as commited.
- How can this be done?
- Solution: SHADOWING

109



110

No-Undo/No-Redo Algorithm

- Let D^b represent the current stable DB, D^0, D^1 are the two possibilities.
- For each txn *Ti*, there is a directory *Di*.
- *Di[x]* refers to the pointer for *x* in *Di*.
- Master pointer, directories must be stable.
- RM-Write*(Ti,x,v):*
  - Write *v* into new location on stable storage, save address in *Di[x].*
  - Ack

111

No-Undo/No-Redo Algorithm

- RM-Read(*Ti,x*):
  - If *Ti* has written *x*, return from *Di[x].*
  - Otherwise, return from $D^b[x]$.
- RM-Commit(*Ti*):
  - For each *x* updated by *Ti*: $D^{-b}[x] \leftarrow Di[x]$, where *b* is the current value of *Master*.
  - *Master* $\leftarrow -b$
  - For each *x* updated by *Ti*: $D^{-b}[x] \leftarrow Di[x]$, where *b* is the (new) value of *Master.*
  - Discard *Di*.
  - Acknowledge Commit.

112

No-Undo/No-Redo Algorithm

- RM-Abort(*Ti*):
  - Add *Ti* to the abort list.
  - Acknowledge
- Restart
  - Copy D^b into D^{-b}
  - Free storage for active txn directories and their copies
  - Acknowledge end of restart

113

Media Failure

- So far, we have relied on the stable storage to bail us out.
- What happens in there is a disk crash – I.e. a *Media Failure?*
- A Media failure results in the loss of all or part of the stable storage (stable DB and stable LOG).
- Nothing is peculiar to Databases for such failures:
  - We can solve (avoid) by using redundancy (RAID, mirroring, etc.).
  - Archiving: idea similar to checkpointing.

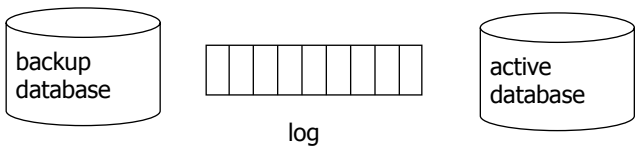
114

Archiving

- Similar to checkpoint: archive vs. stable CKPT.
- Stable (regular) CKPT:
  - Update the log to indicate what is in stable DB
  - Update the stable DB to includes updates that are only in cache.
- Archive CKPT:
  - Update the log to indicate what is in the archive DB
  - Update the achive DB to include updates that are only in stable DB and cache.

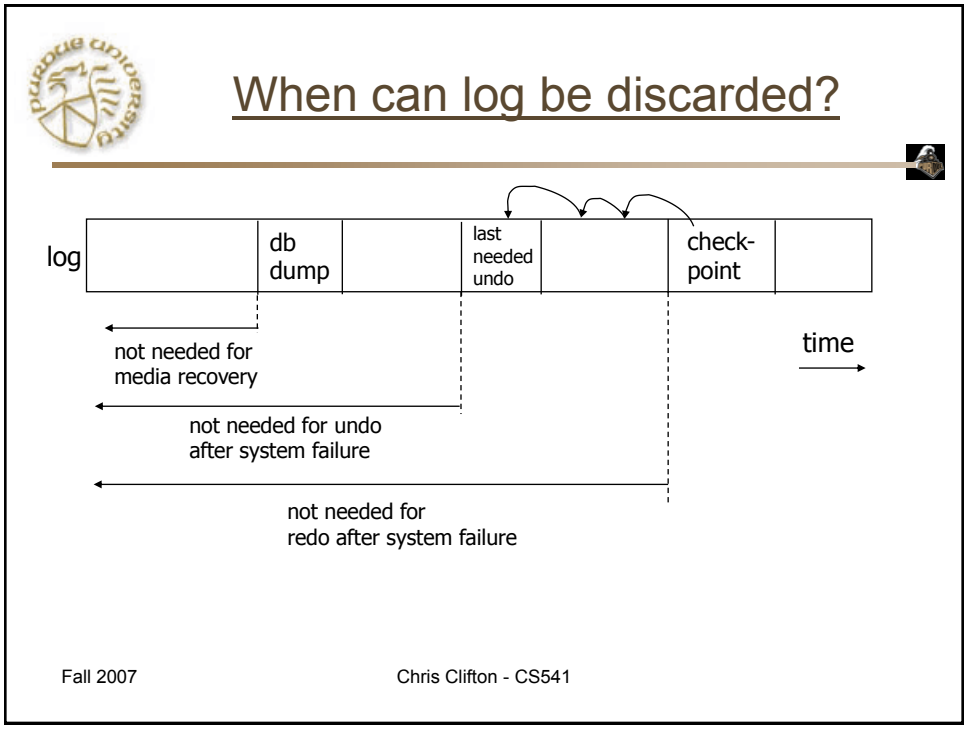
118

DB Dump + Log



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

Fall 2007 Chris Clifton - CS541


When can log be discarded?
log
db dump
last needed undo
check-point
time
not needed for media recovery
not needed for undo after system failure
not needed for redo after system failure
Fall 2007
Chris Clifton - CS541