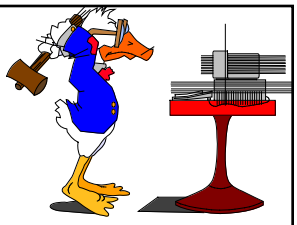
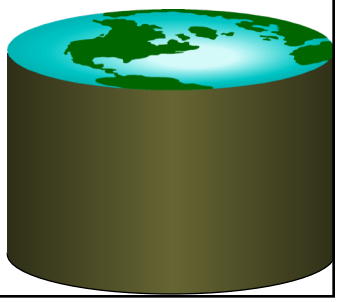
Crash Recovery



Chapter 18



1

Review: The ACID properties

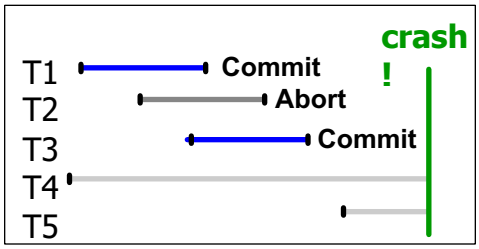
- **Atomicity:** All actions in the Xact happen, or none happens.
- **Consistency:** If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **Isolation:** Execution of one Xact is isolated from that of other Xacts.
- **Durability:** If a Xact commits, its effects persist.
- Question: which ones does the **Recovery Manager** help with?
  - Atomicity & Durability (and also used for Consistency-related rollbacks)

2

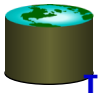
Motivation

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

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



3



Introduction to Crash Recovery

- **Recovery Manager**
  - When a DBMS is restarted after crashes, the recovery manager must bring the database to a consistent state
  - Ensures transaction atomicity and durability
  - Undos actions of transactions that do not commit
  - Redos actions of committed transactions during system failures and media failures (corrupted disk).
- **Recovery Manager maintains log information during normal execution of transactions for use during crash recovery**

4

The Log

- **Log consists of "records" that are written sequentially.**
  - Typically chained together by Xact id
  - Log is often *duplexed* and *archived* on stable storage.
- **Log stores modifications to the database**
  - *if Ti writes an object*, write a log record with:
  - If UNDO required need "before image"
  - IF REDO required need "after image".
  - *Ti commits/aborts*: a log record indicating this action.
- **Need for UNDO and/or REDO depend on Buffer Mgr.**
  - UNDO required if uncommitted data can overwrite stable version of committed data (STEAL buffer management).
  - REDO required if xact can commit before all its updates are on disk (NO FORCE buffer management).

5

Logging Continued

- **Write Ahead Logging (WAL) protocol**
  - Log record must go to disk *before* the changed page!
    - implemented via a handshake between log manager and the buffer manager.
  - All log records for a transaction (including it's commit record) must be written to disk before the transaction is considered "Committed".
- **All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.**

6

ARIES Recovery

- **There are 3 phases in ARIES recovery:**
  - *Analysis*: Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
  - *Redo*: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
  - *Undo*: The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, as found in the log), working backwards in the log.
- **At the end --- all committed updates and only those updates are reflected in the database.**
- **Some care must be taken to handle the case of a crash occurring during the recovery process!**

7

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?

8

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 uncommited 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…

9

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).

10

Buffer Management summary

	No Steal	Steal
No Force		Fastest
Force	Slowest	

Performance Implications

	No Steal	Steal
No Force	No UNDO REDO	UNDO REDO
Force	No UNDO No REDO	UNDO No REDO

Logging/Recovery Implications

11

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).

12

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*. (alt. 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**

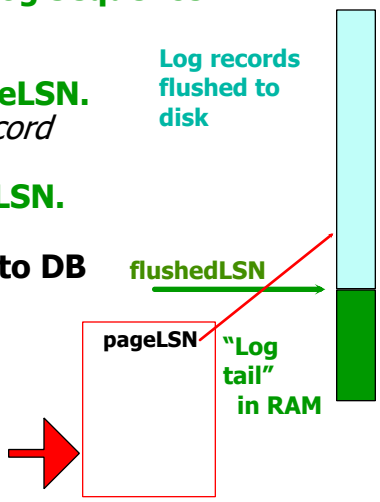
- **Exactly how is logging (and recovery!) done?**
- We'll look at the ARIES algorithms from IBM.

13

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:**

$pageLSN_i <= flushedLSN$



14

Log Records

LogRecord fields:
- LSN
- prevLSN
- XID
- type
- **update records only:**
  - pageID
  - length
  - offset
  - before-image
  - after-image

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:

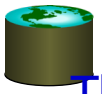
- **Update, Commit, Abort**
- **Checkpoint (for log maintainence)**
- **Compensation Log Records (CLRs)**
- for UNDO actions
- **End (end of commit or abort)**

15

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.

16

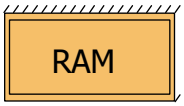


The Big Picture: What's Stored Where

LogRecords
LSN
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

17

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.**

18

Transaction Commit

- **Write commit record to log.**
- **All log records up to Xact's commit record are flushed to disk.**
- 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.**

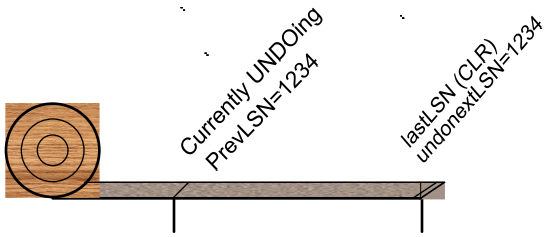
19

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.

20

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).
- 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.**

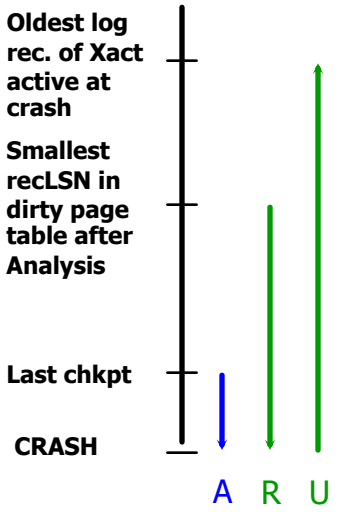
21

Checkpointing

- **Conceptually, keep log around for all time. Obviously this has performance/implemenation 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).

22

Crash Recovery: Big Picture



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

23

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

24

Phase 2: 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 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) >= LSN. (this last case requires I/O)
- **To REDO an action:**
- Reapply logged action.
- Set pageLSN to LSN. No additional logging, no forcing!

25

Phase 3: The UNDO Phase

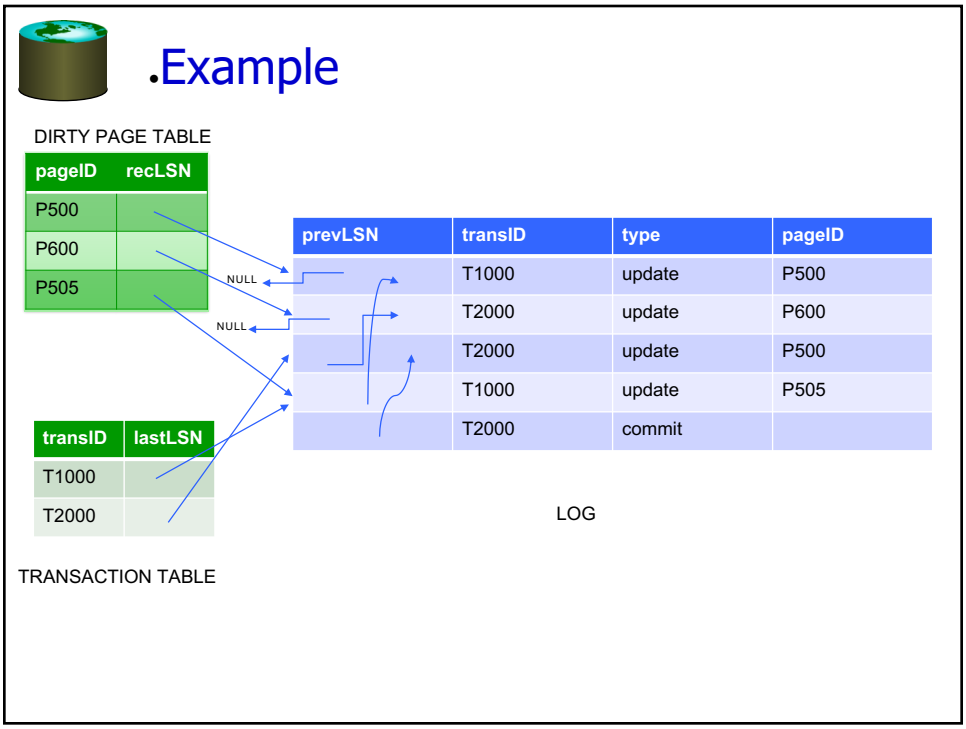
ToUndo={lastLSNs of all Xacts in the Trans Table}
a.k.a. "losers"

Repeat:

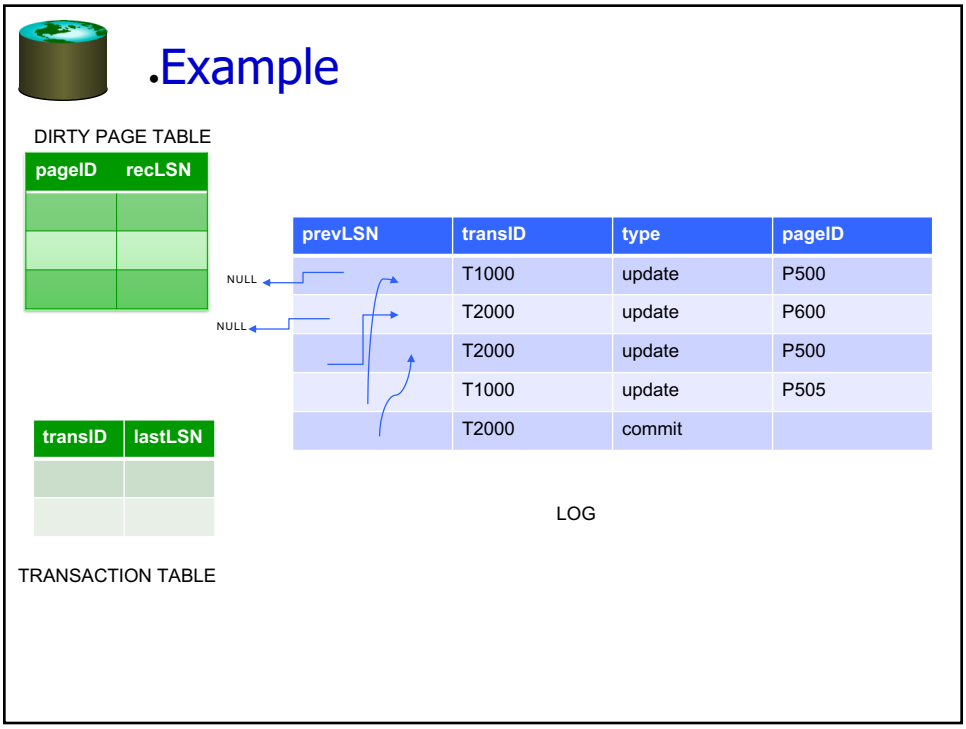
- Choose (and remove) largest LSN among ToUndo.
- If this LSN is a CLR and undonextLSN==NULL
- Write an End record for this Xact.
- If this LSN is a CLR, and undonextLSN != NULL
- Add undonextLSN to ToUndo
- Else this LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo.

Until ToUndo is empty.

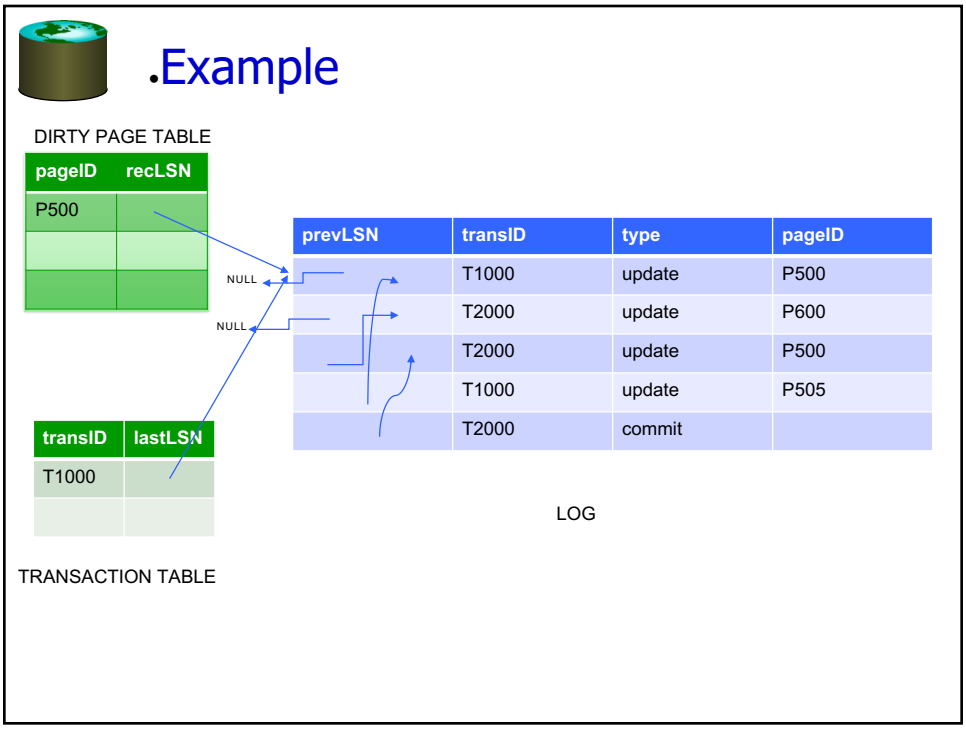
26


.Example
DIRTY PAGE TABLE
pageID
recLSN
P500
P600
P505
NULL
NULL
prevLSN
transID
type
pageID
T1000
update
P500
T2000
update
P600
T2000
update
P500
T1000
update
P505
T2000
commit
transID
lastLSN
T1000
T2000
LOG
TRANSACTION TABLE

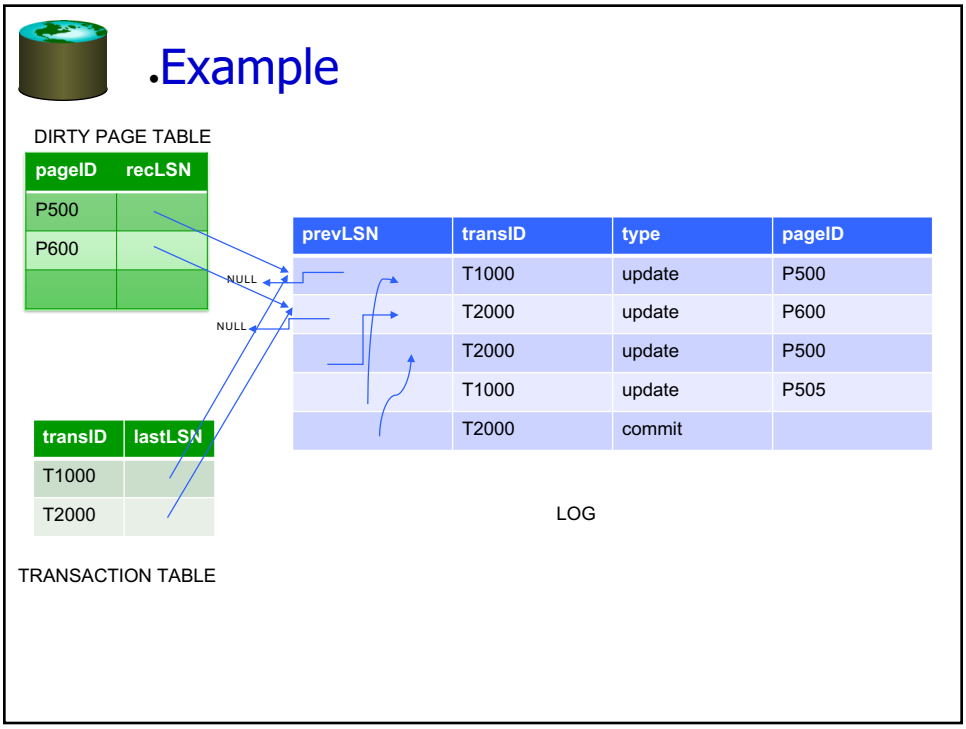
27


.Example
DIRTY PAGE TABLE
pageID
recLSN
NULL
NULL
prevLSN
transID
type
pageID
T1000
update
P500
T2000
update
P600
T2000
update
P500
T1000
update
P505
T2000
commit
transID
lastLSN
LOG
TRANSACTION TABLE

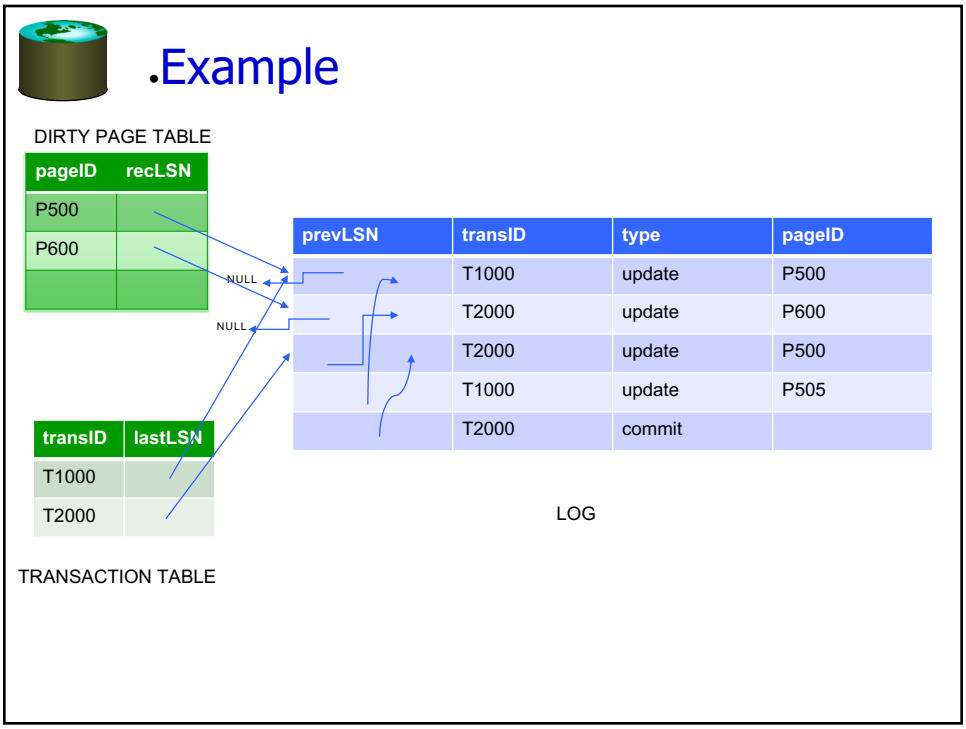
28


.Example
DIRTY PAGE TABLE
pageID
recLSN
P500
NULL
NULL
prevLSN
transID
type
pageID
T1000
update
P500
T2000
update
P600
T2000
update
P500
T1000
update
P505
T2000
commit
transID
lastLSN
T1000
LOG
TRANSACTION TABLE

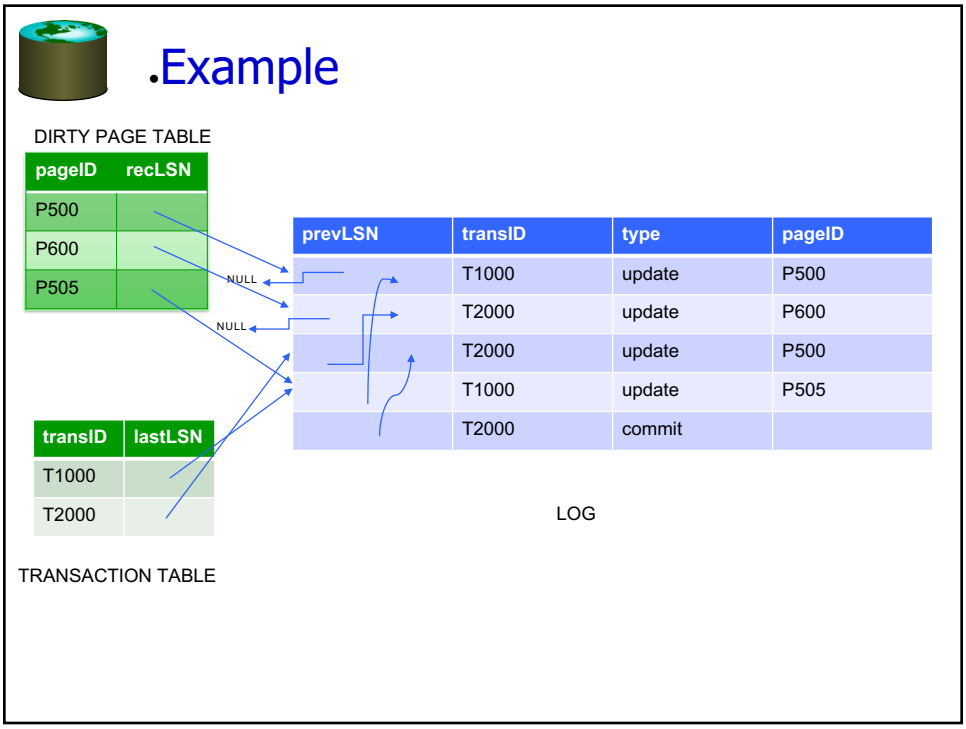
29


.Example
DIRTY PAGE TABLE
pageID
recLSN
P500
P600
NULL
NULL
prevLSN
transID
type
pageID
T1000
update
P500
T2000
update
P600
T2000
update
P500
T1000
update
P505
T2000
commit
transID
lastLSN
T1000
T2000
LOG
TRANSACTION TABLE

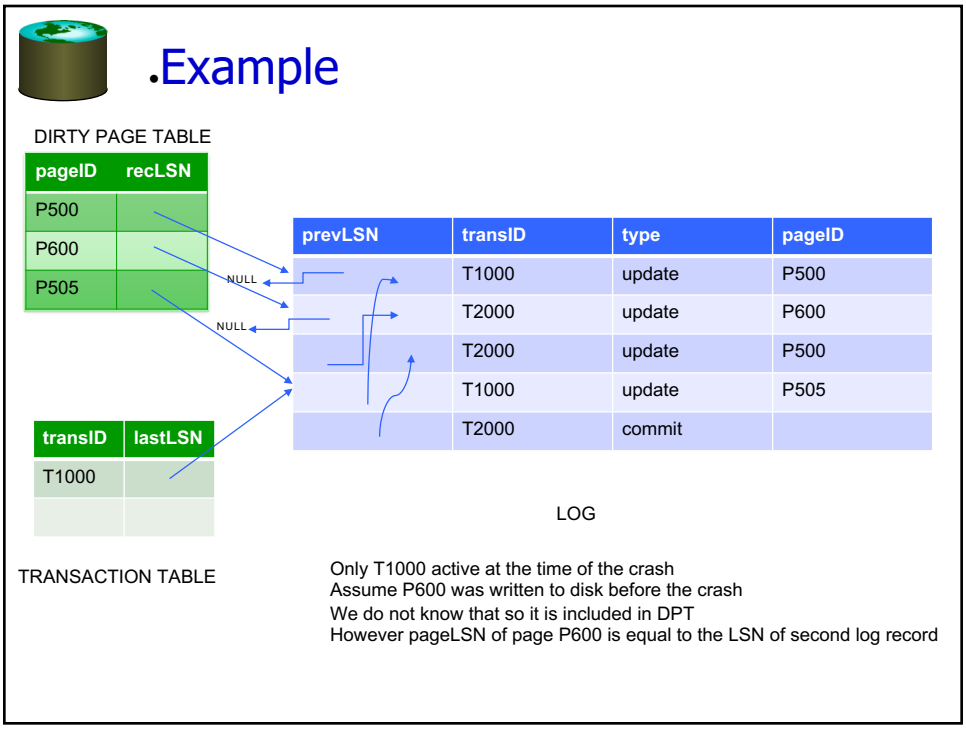
30


.Example
DIRTY PAGE TABLE
pageID
recLSN
P500
P600
NULL
NULL
prevLSN
transID
type
pageID
T1000
update
P500
T2000
update
P600
T2000
update
P500
T1000
update
P505
T2000
commit
transID
lastLSN
T1000
T2000
LOG
TRANSACTION TABLE

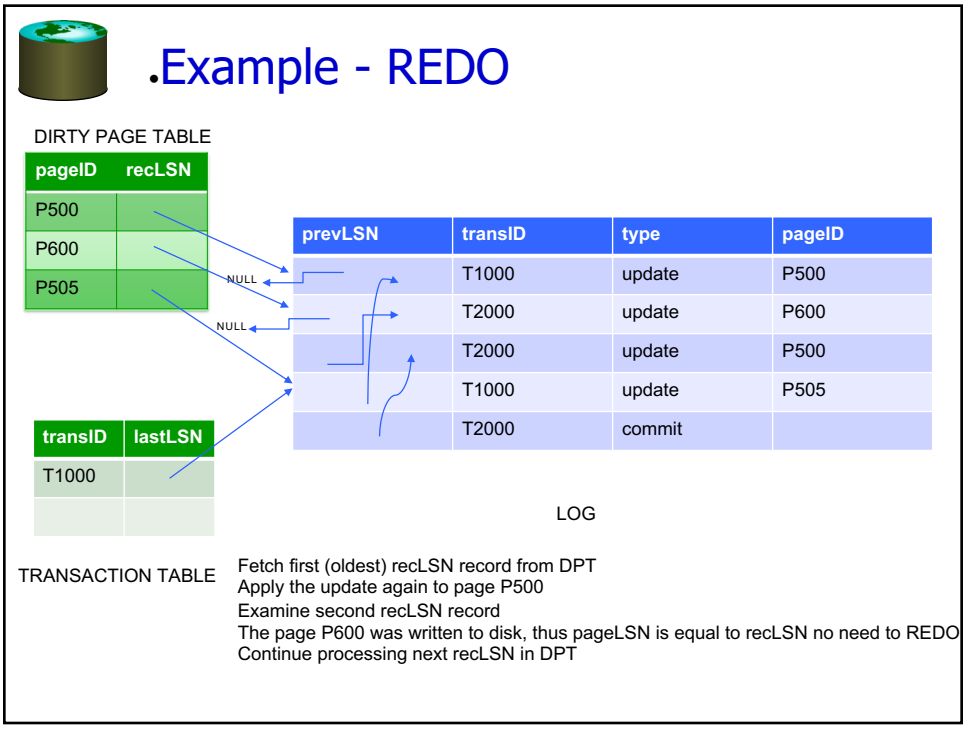
31


.Example
DIRTY PAGE TABLE
pageID
recLSN
P500
P600
P505
NULL
NULL
prevLSN
transID
type
pageID
T1000
update
P500
T2000
update
P600
T2000
update
P500
T1000
update
P505
T2000
commit
transID
lastLSN
T1000
T2000
LOG
TRANSACTION TABLE

32


.Example
DIRTY PAGE TABLE
pageID
recLSN
P500
P600
P505
NULL
NULL
prevLSN
transID
type
pageID
T1000
update
P500
T2000
update
P600
T2000
update
P500
T1000
update
P505
T2000
commit
transID
lastLSN
T1000
LOG
TRANSACTION TABLE
Only T1000 active at the time of the crash
Assume P600 was written to disk before the crash
We do not know that so it is included in DPT
However pageLSN of page P600 is equal to the LSN of second log record

33


.Example - REDO
DIRTY PAGE TABLE
pageID
recLSN
P500
P600
P505
NULL
NULL
prevLSN
transID
type
pageID
T1000
update
P500
T2000
update
P600
T2000
update
P500
T1000
update
P505
T2000
commit
transID
lastLSN
T1000
LOG
TRANSACTION TABLE
Fetch first (oldest) recLSN record from DPT
Apply the update again to page P500
Examine second recLSN record
The page P600 was written to disk, thus pageLSN is equal to recLSN no need to REDO
Continue processing next recLSN in DPT

34

.Example - UNDO

DIRTY PAGE TABLE

pageID	recLSN
P500	
P600	
P505	

prevLSN	transID	type	pageID
NULL	T1000	update	P500
NULL	T2000	update	P600
	T2000	update	P500
	T1000	update	P505
	T2000	commit	

LOG

transID	lastLSN
T1000	

TRANSACTION TABLE

Only active transaction is T1000 from the Transaction Table
LSN of most recend record of T1000 is fourth log record
Start UNDOing following prevLSN, issuing a CLR with undoNextLSN = to first log record
Then UNDO the first log record, issue a CLR and an END log record for T1000

35

.Example - UNDO

DIRTY PAGE TABLE

pageID	recLSN
P500	
P600	
P505	

prevLSN	transID	type	pageID
NULL	T1000	update	P500
NULL	T2000	update	P600
	T2000	update	P500
	T1000	update	P505
	T2000	commit	

LOG

transID	lastLSN
T1000	

TRANSACTION TABLE

ISSUE: UNDOing first log record, causes the changes of the third log record
From a COMMITTED transaction to be lost! This is because T2000 overwrote
A data item written by T1000 while T1000 was still active. With Strict PL T2000
Would never have been allowed to overwrite.

36

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.

39

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.**

40

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!**

41