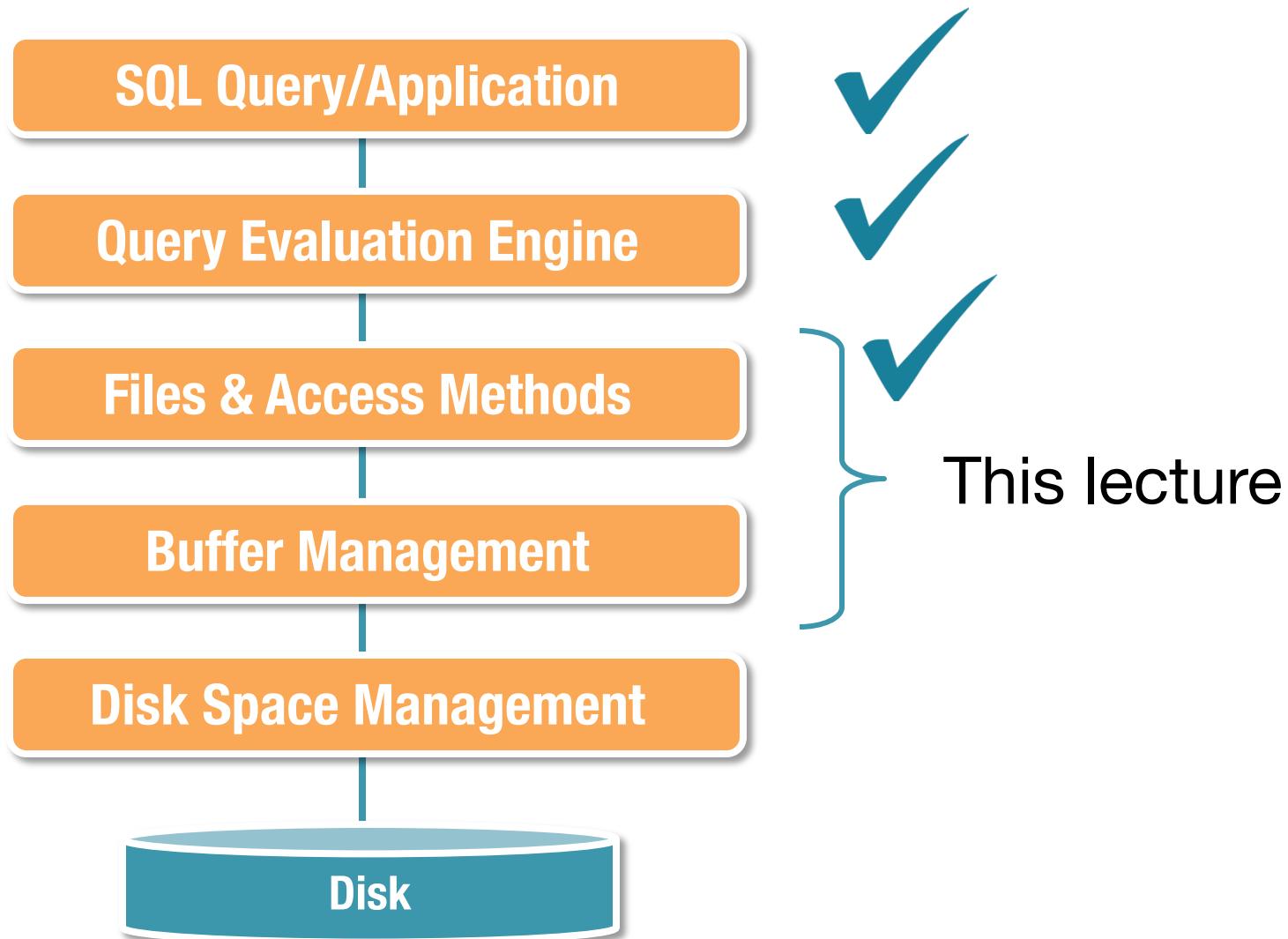


Database Recovery

Chapter 16.7 and 18
(except 18.6 and 18.8)

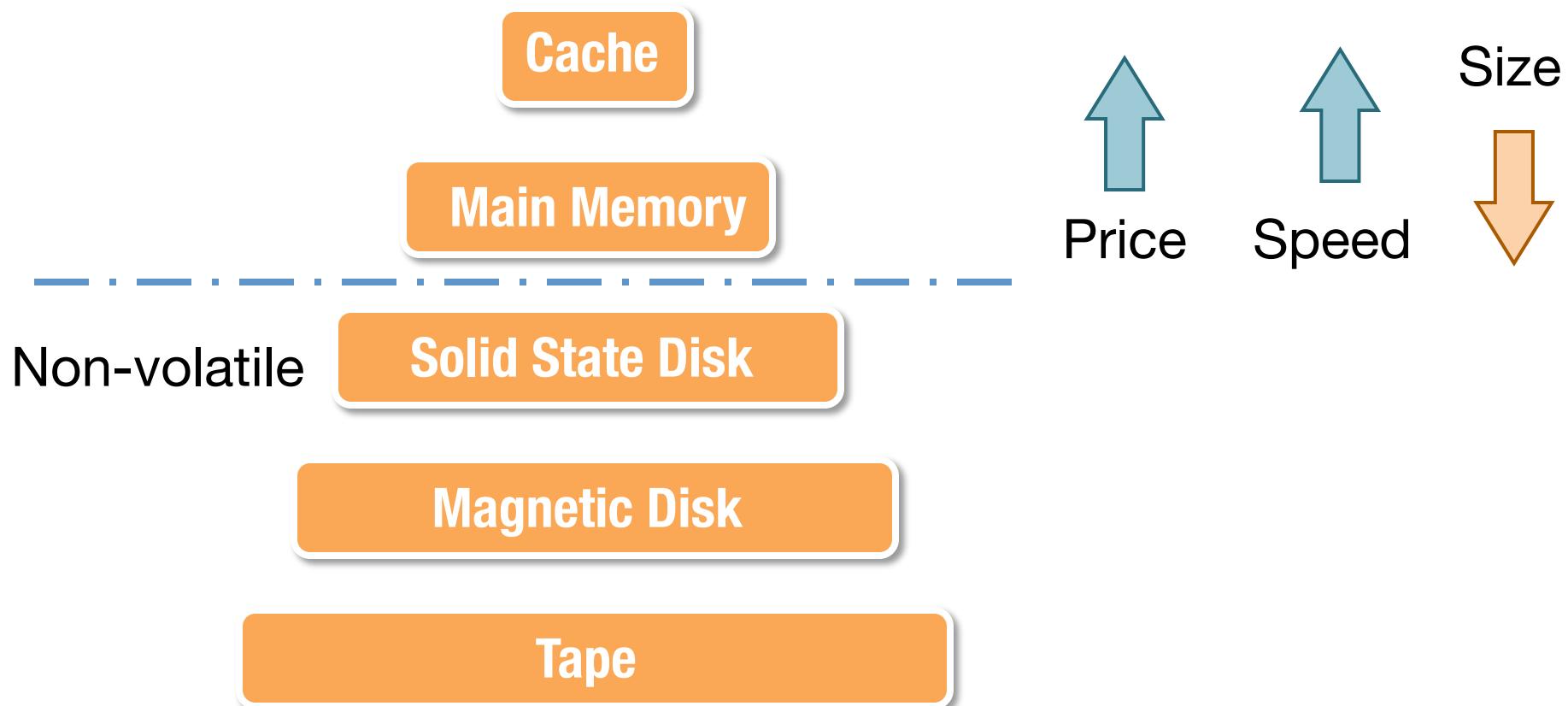
University of Michigan, Ann Arbor

DBMS Organization



The Memory Hierarchy

Performance of Microprocessors and Memory
improving faster than disks and tapes

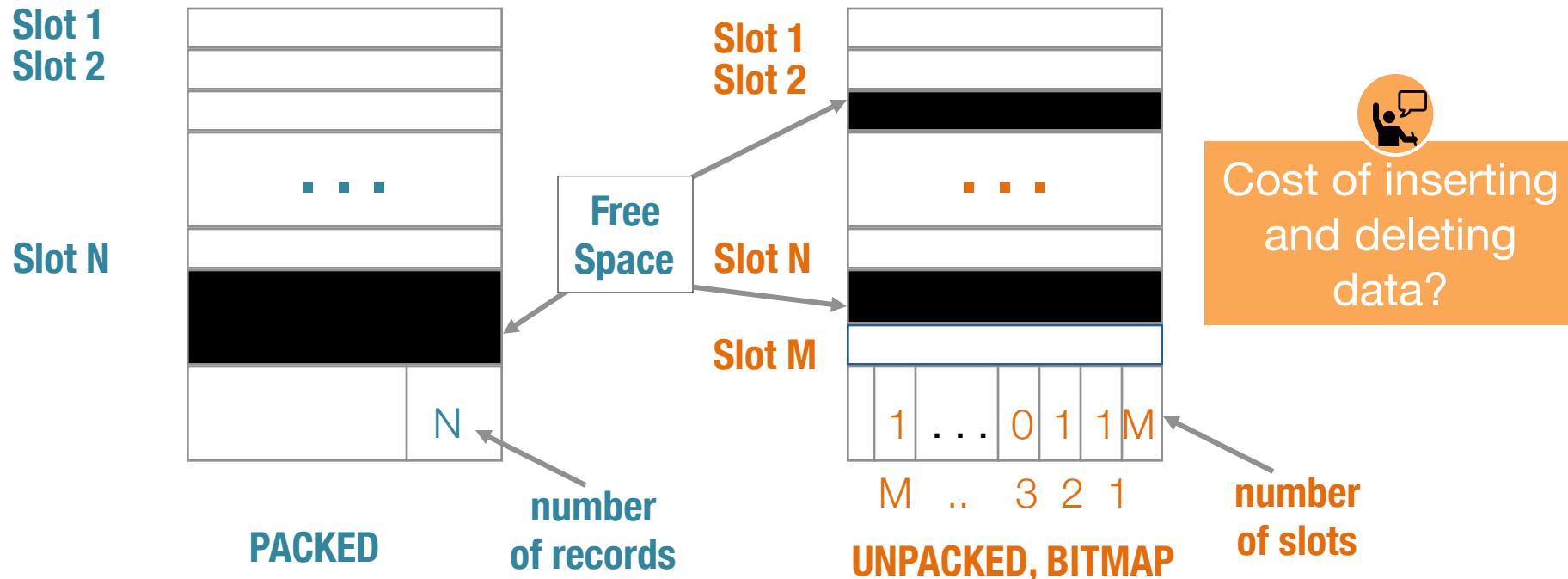




Unit of I/O

- Unit of in-memory I/O:
 - Words (32 or 64 bits)
- Unit of writing to disk:
 - Pages (512 bytes to 16K)
 - Pages contain records from a table
 - e.g. `INSERT INTO Sailors VALUES(3, 'dustin', 23, 8);`

Page Formats: Fixed Length Records



- Record id = <page id, slot #>
- Two alternatives for maintaining database records shown above. First requires more in-memory copying on delete of a record



Naive Strategy

```
INSERT INTO Sailors VALUES(3, 'dustin', 23, 8);
```

- This requires:
 1. READ the page from the disk containing the Sailors records and an available slot
 2. Update the page in memory to add a new record
 3. WRITE the page back to disk (*persistence - crash*)
- The disk READ and WRITE are very slow, compared to in-memory update
- Can we do better?



Better Strategy: Use a Buffer Pool

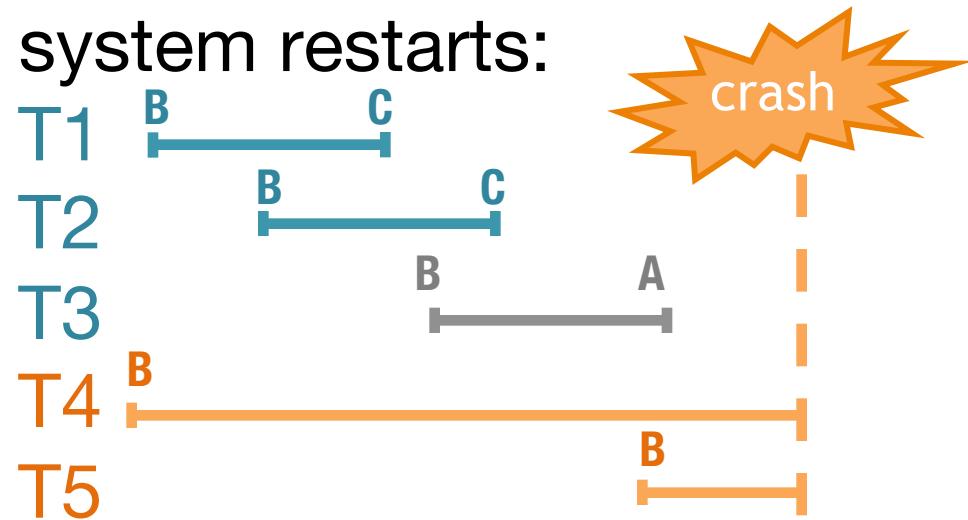
- **Reduce reads by caching the accessed pages in memory in a buffer pool**
 - But buffer pool is limited. Thus also need to remove pages from memory
- **Reduce writes by delaying writing pages back to disk until insufficient memory**



Challenge: What if there is a
CRASH?

Recovery

- Atomicity:
 - Transactions may abort (“Rollback”)
- Durability:
 - What if DBMS stops running? (Causes?)
- Desired Behavior after system restarts:
 - T1 & T2 should be durable
 - T4 & T5 should be aborted and any writes by them undone



What Can Go Wrong?

- Page writes of a **committed** transaction:
 - If **page writes only in memory** (not pushed to disk) then BAD! Crash could fail to recover committed writes
- Page writes of an **uncommitted** transaction:
 - If **uncommitted writes pushed to disk** (e.g., because of memory pressure), then BAD! Crash could make uncommitted writes durable!



One Solution

- Try to maintain an **invariant** that disk **only** has committed pages for **all** transactions. This requires Transaction Manager to:
 - **FORCE** pages of a transaction to disk upon a COMMIT.
 - **NOT STEAL** pages of uncommitted transactions when there is memory pressure. Stealing would require pushing uncommitted pages to disk
- Unfortunately, both would hurt performance!

Buffer Pool and ACID Properties

Fortunately, alternatives exist:

After a crash...



Force

Committed
pages

No Force

Uncommitted

No Steal *pages* Steal

	Trivial, but hurts performance	Steal is good, but undo of uncommitted transactions required
	No Force is good, but redo of committed transactions required	Best performance! But undo/redo both required

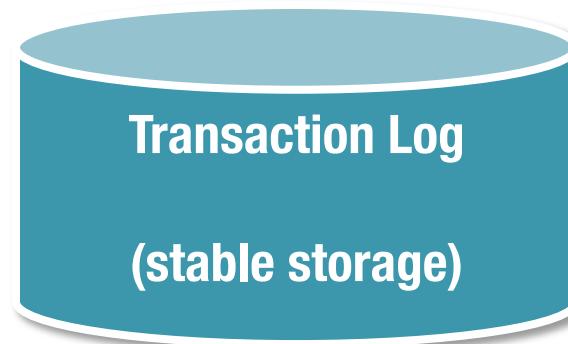
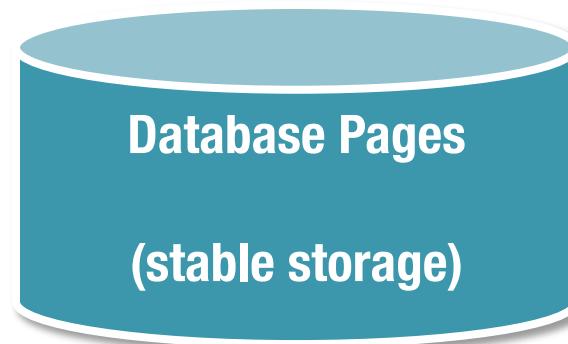
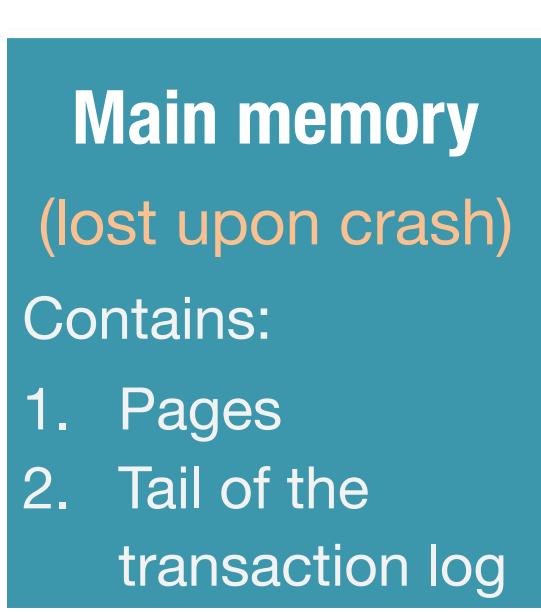
ARIES Protocol

- ARIES: Algorithm for Recovery and Isolation Exploiting Semantics
 - Developed by IBM researchers (1992)
- Key features:
 - Supports **steal** and **no force**
 - Every update written to two places:
 - To a database page (e.g., a record)
 - A transaction log
 - Transaction log used to help:
 - Undo stolen pages with uncommitted writes
 - Redo committed writes



C Mohan

Log to Support Undo or Redo



Log Allows Database Repair

LSN	LOG
• 10	Update: T1 writes P5 (oldval, newval)
• 20	Update: T2 writes P3 (oldval, newval)
• 30	T2 commit
• 40	T2 end
• 50	Update: T3 writes P1 (oldval, newval)
• 60	Update: T3 writes P3 (oldval, newval)

BOOM!!!! CRASSSHSH!!!!

Upon restart, what can you tell from the log?

- What txs were **active** at the time of the crash?
- What changes should potentially be **undone**?
- What changes should potentially be **redone**?

Log Entry Fields

- Record a **log** record for every **update**:
 - log sequence number (LSN)
 - XID
 - prevLSN (points to prev LSN in same transaction)
 - page#, offset, size, prev-image, new-image
- There are log records for commits/aborts too.

Log Record Types

LogRecord fields:

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

update records only {

Possible log record types:

- **Update**
- **Commit**
- **Abort**
- **End** (end of commit or abort)
- Compensation Log Rec. (CLRs)
 - For UNDO actions
 - (More later)

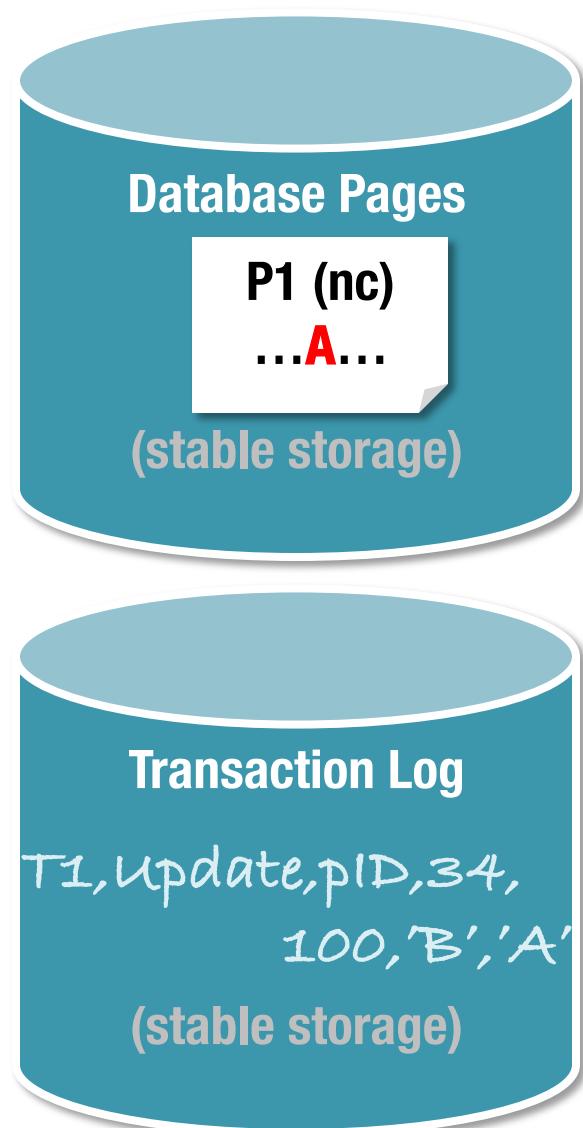
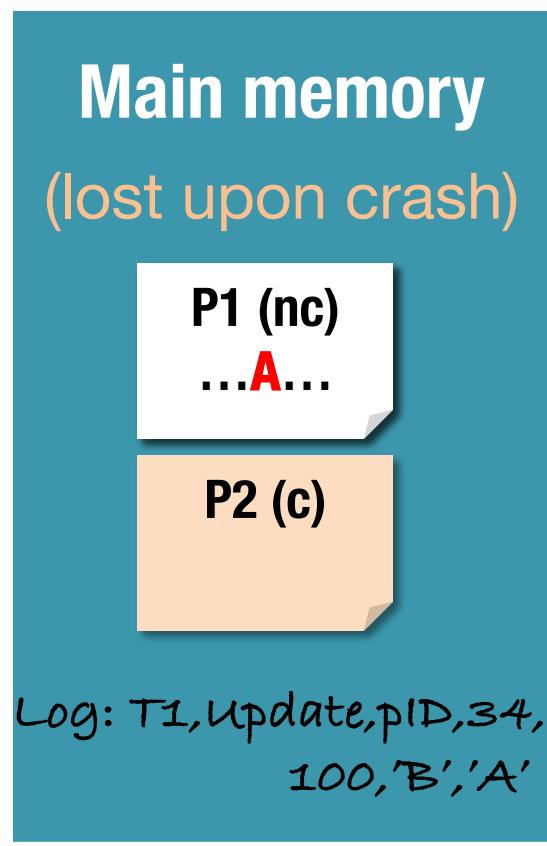
Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
 1. 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
 - Allows undo of stolen pages
- #2 guarantees Durability
 - Allows redo of unforced but committed pages

Logging: Grade Update B→A

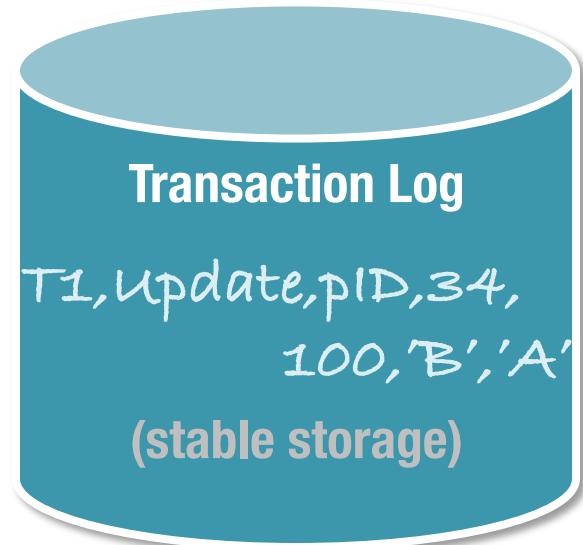
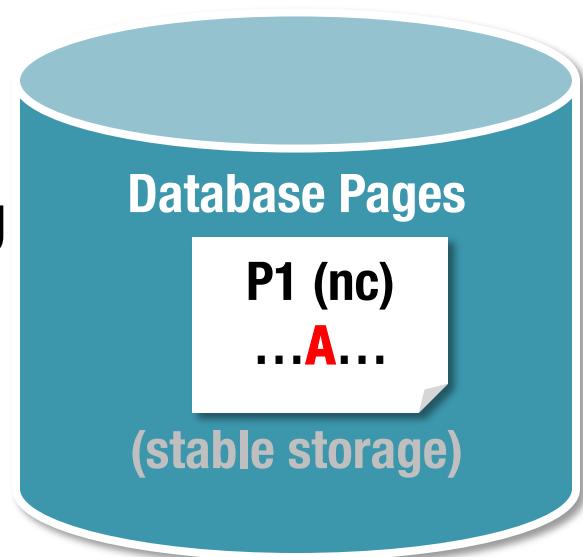
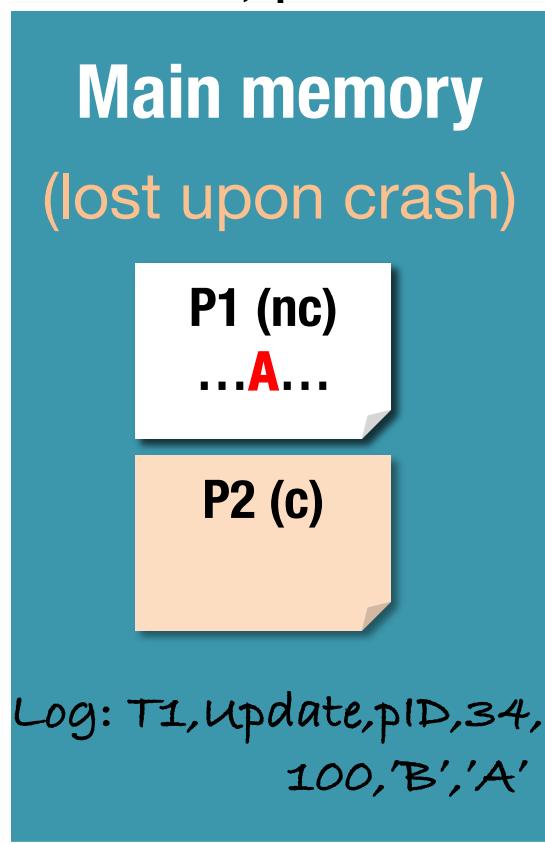
Assume we need to write P1 to the disk
because of limited memory

An uncommitted
transaction
changes P1 by
updating B->A



Logging: Grade Update B→A

1. Write in-memory log
2. Write P1 in memory
3. Before pushing P1 to disk, push the log
4. Push P1 to disk



Logging: Grade Update B→A

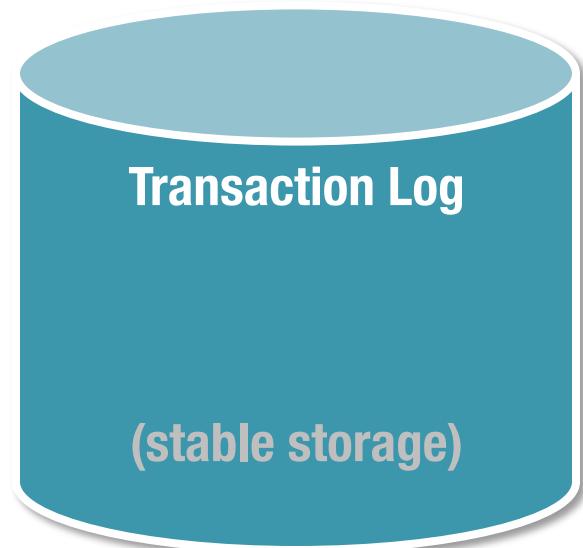
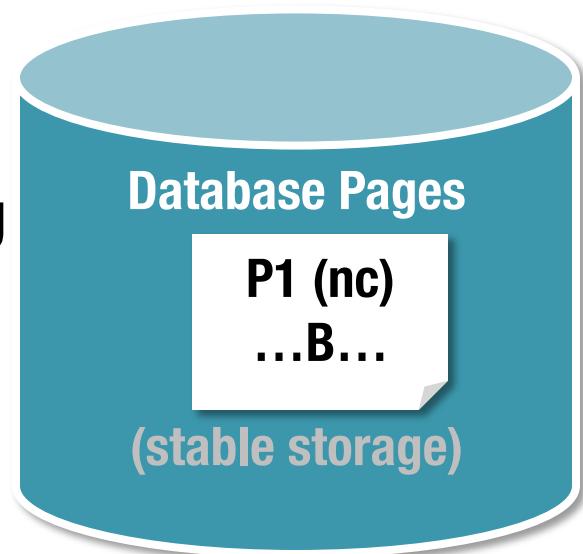
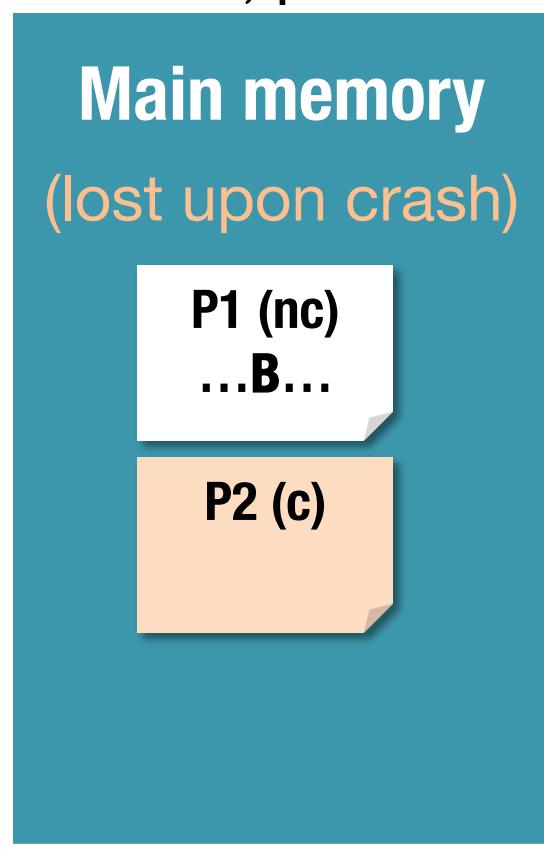
crash

1. Write in-memory log
2. Write P1 in memory
3. Before pushing P1 to disk, push the log
4. Push P1 to disk



What if there is
a crash before
any of these
steps?

Ans: all good.
nothing changed
and nothing
written to disk



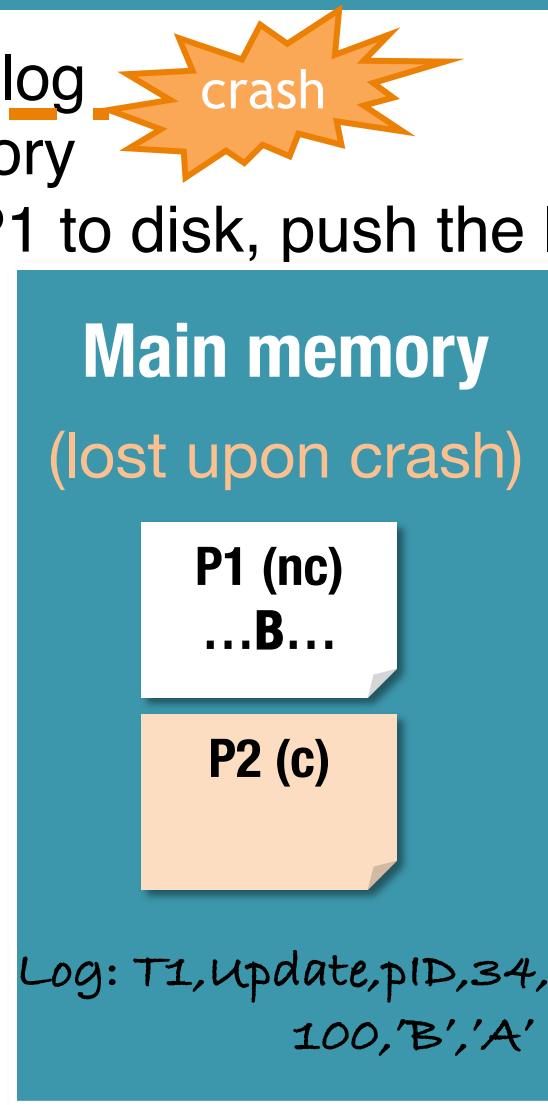
Logging: Grade Update B→A

1. Write in-memory log
2. Write P1 in memory
3. Before pushing P1 to disk, push the log
4. Push P1 to disk



What if there is
a crash before
any of these
steps?

Ans: all good.
nothing written to
disk for this
uncommitted trax



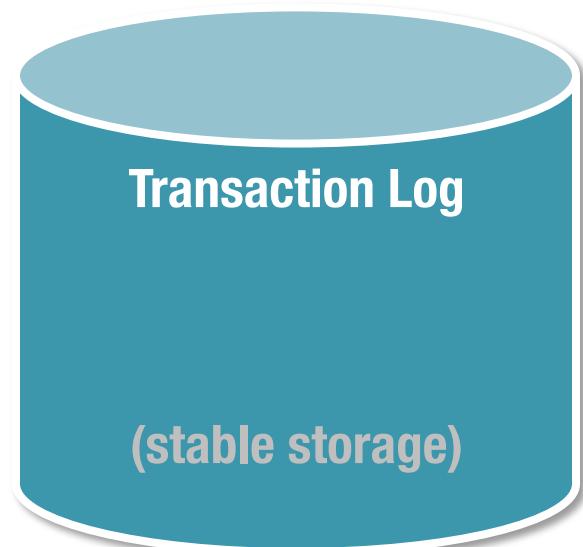
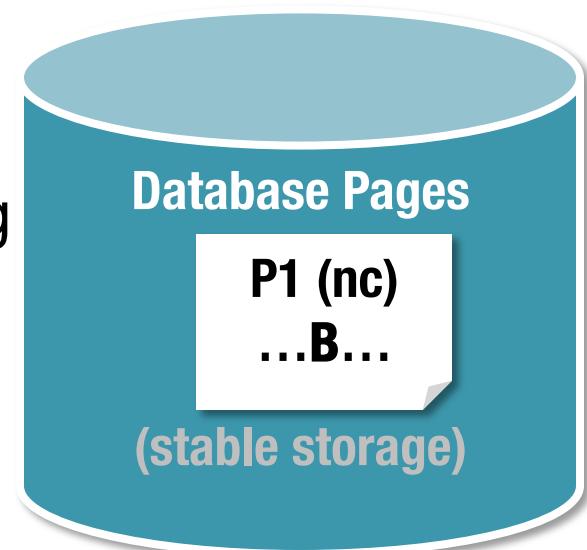
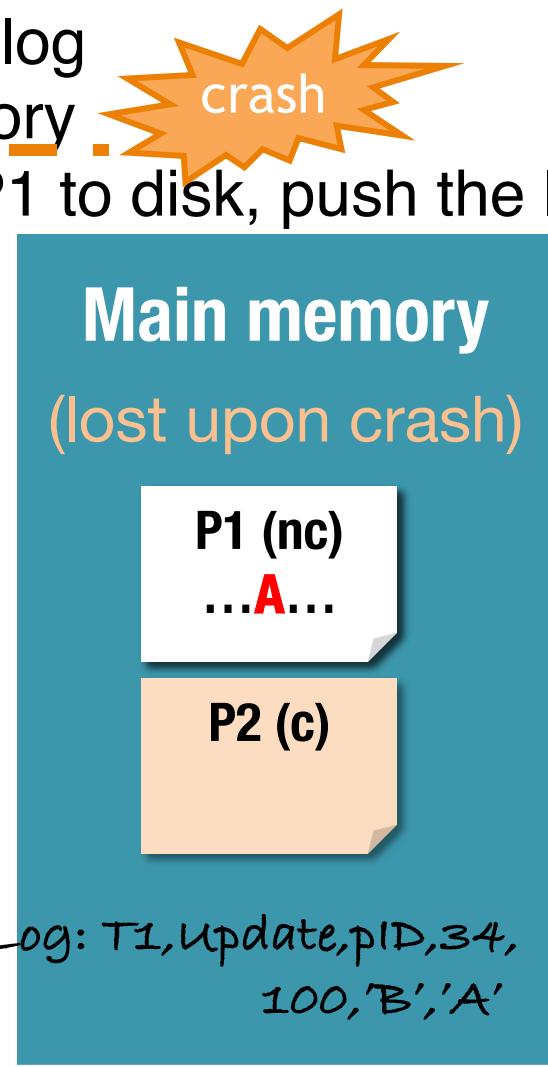
Logging: Grade Update B→A

1. Write in-memory log
2. Write P1 in memory
3. Before pushing P1 to disk, push the log
4. Push P1 to disk



What if there is a crash before any of these steps?

Ans: all good.
nothing written to disk for uncommitted trax. so no inconsistencies



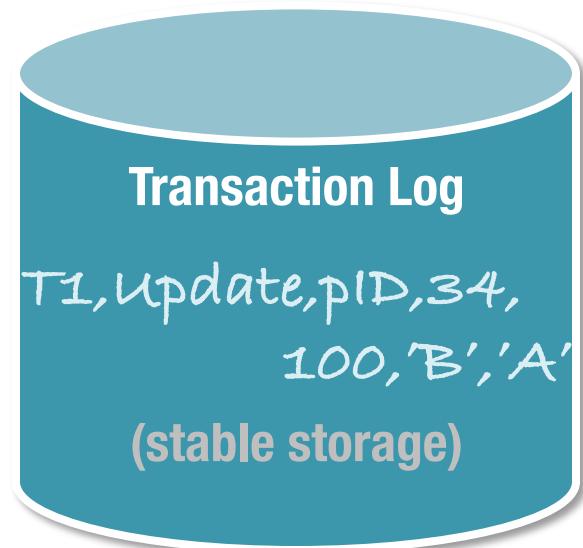
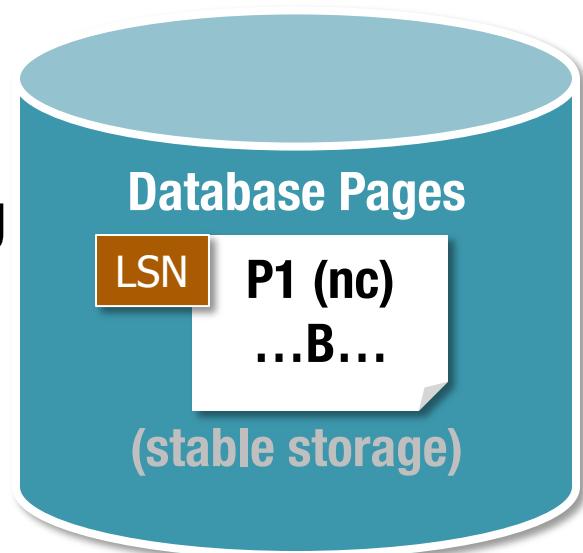
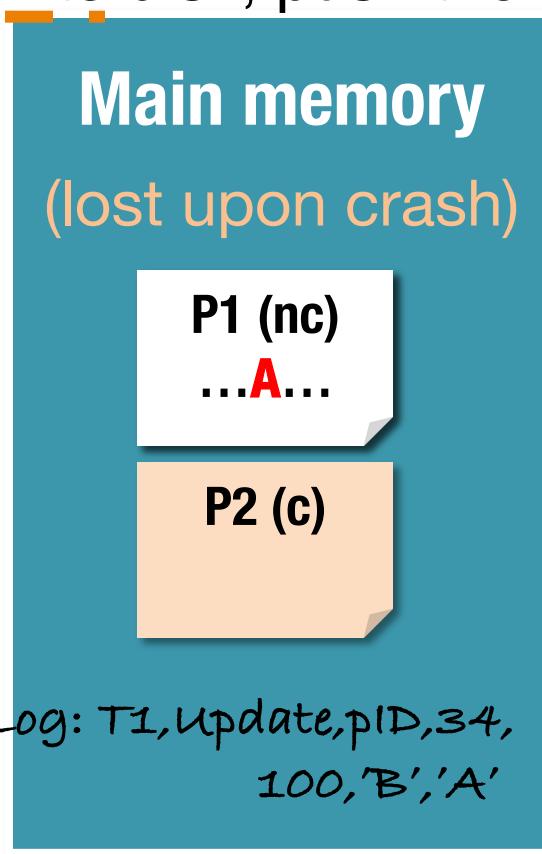
Logging: Grade Update B→A

1. Write in-memory log
2. Write P1 in memory
3. Before pushing P1 to disk, push the log
4. Push P1 to disk



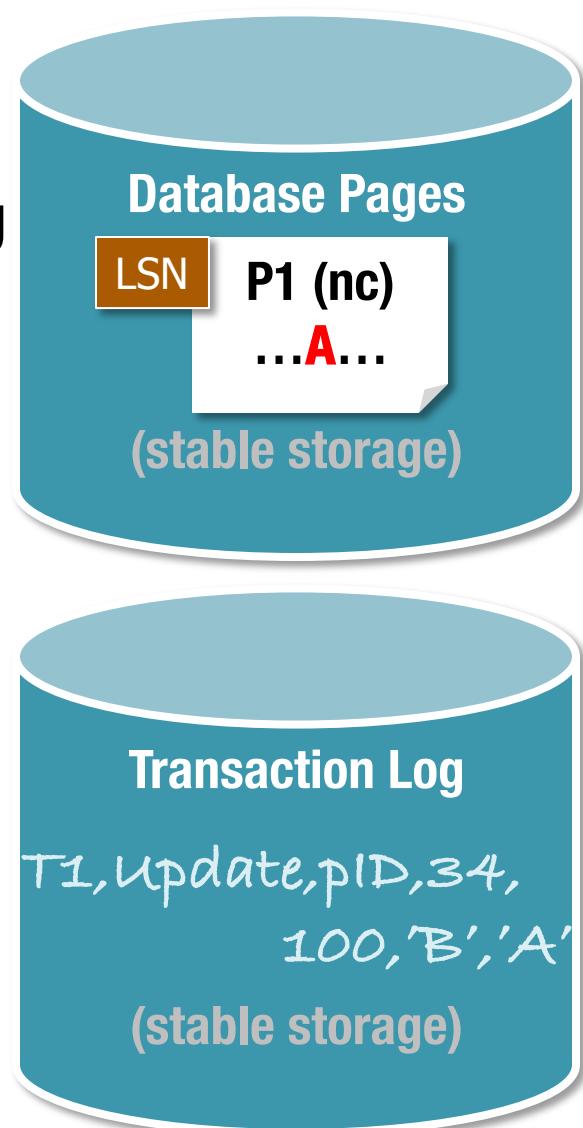
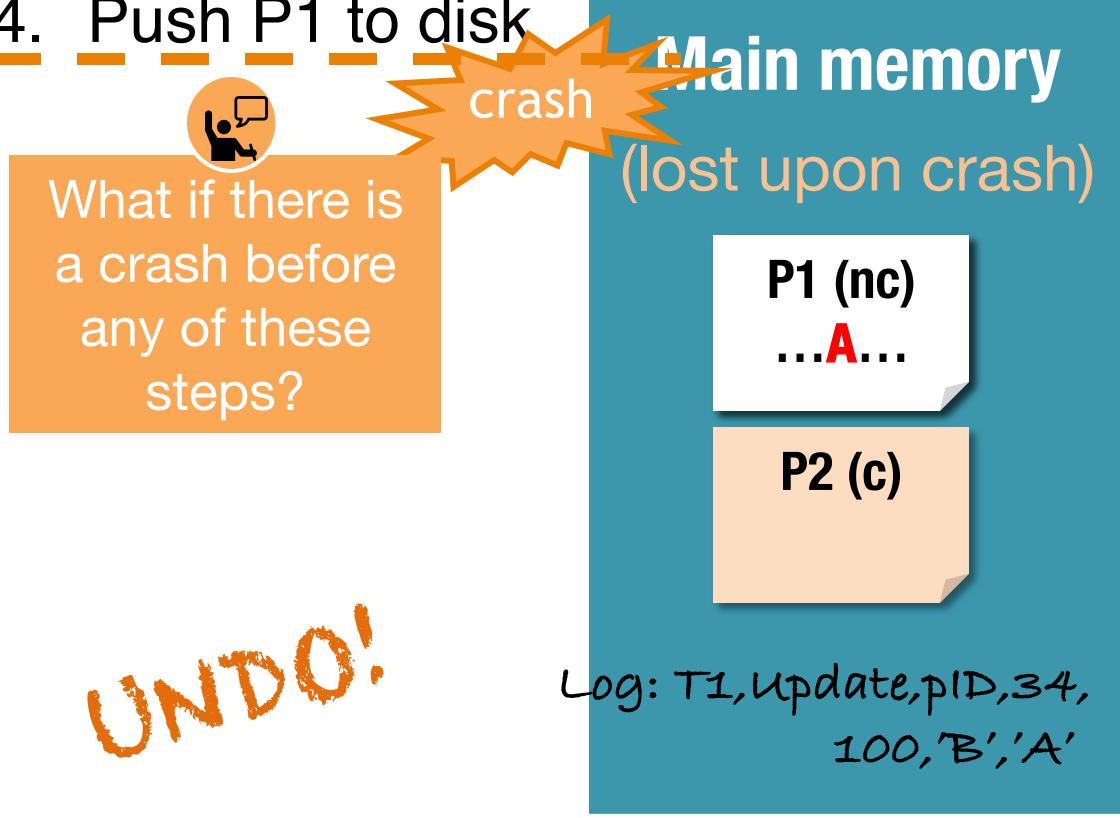
What if there is
a crash before
any of these
steps?

Ans: log says A,
but disk copy
says B. easy to
detect: compare
LSN values



Logging: Grade Update B→A

1. Write in-memory log
2. Write P1 in memory
3. Before pushing P1 to disk, push the log
4. Push P1 to disk



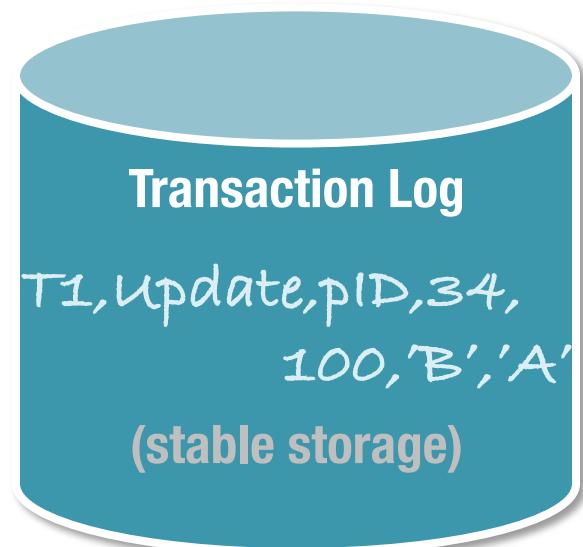
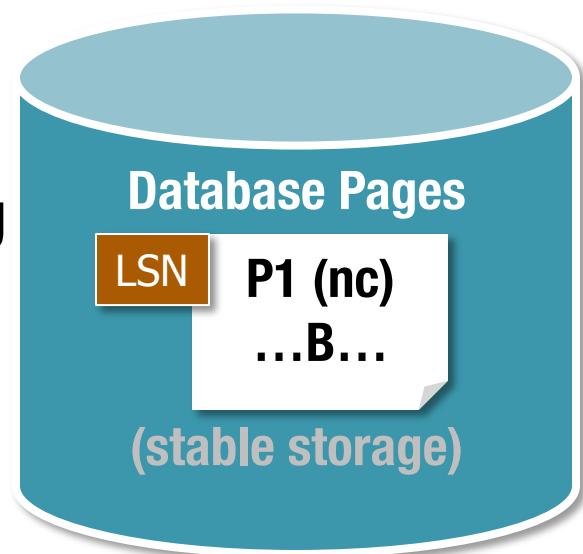
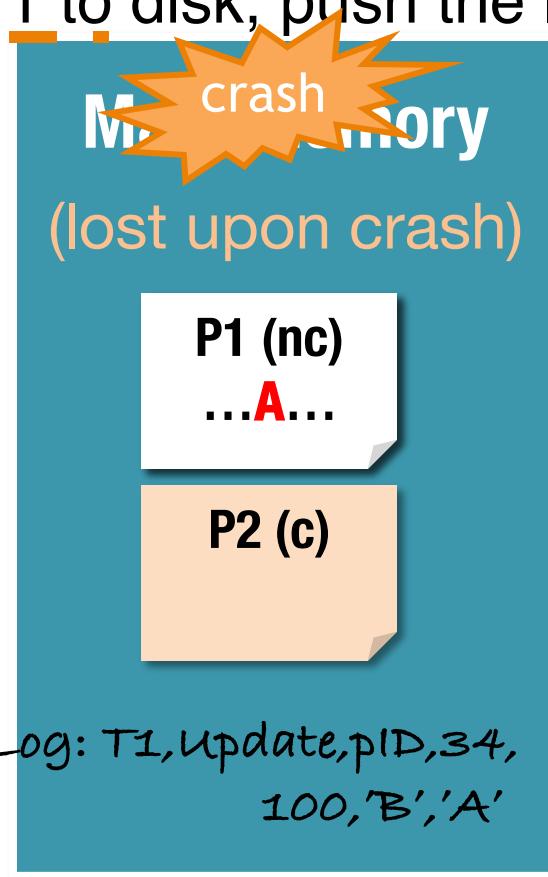
Logging: Grade Update B→A

1. Write in-memory log
2. Write P1 in memory
3. Before pushing P1 to disk, push the log
4. Push P1 to disk

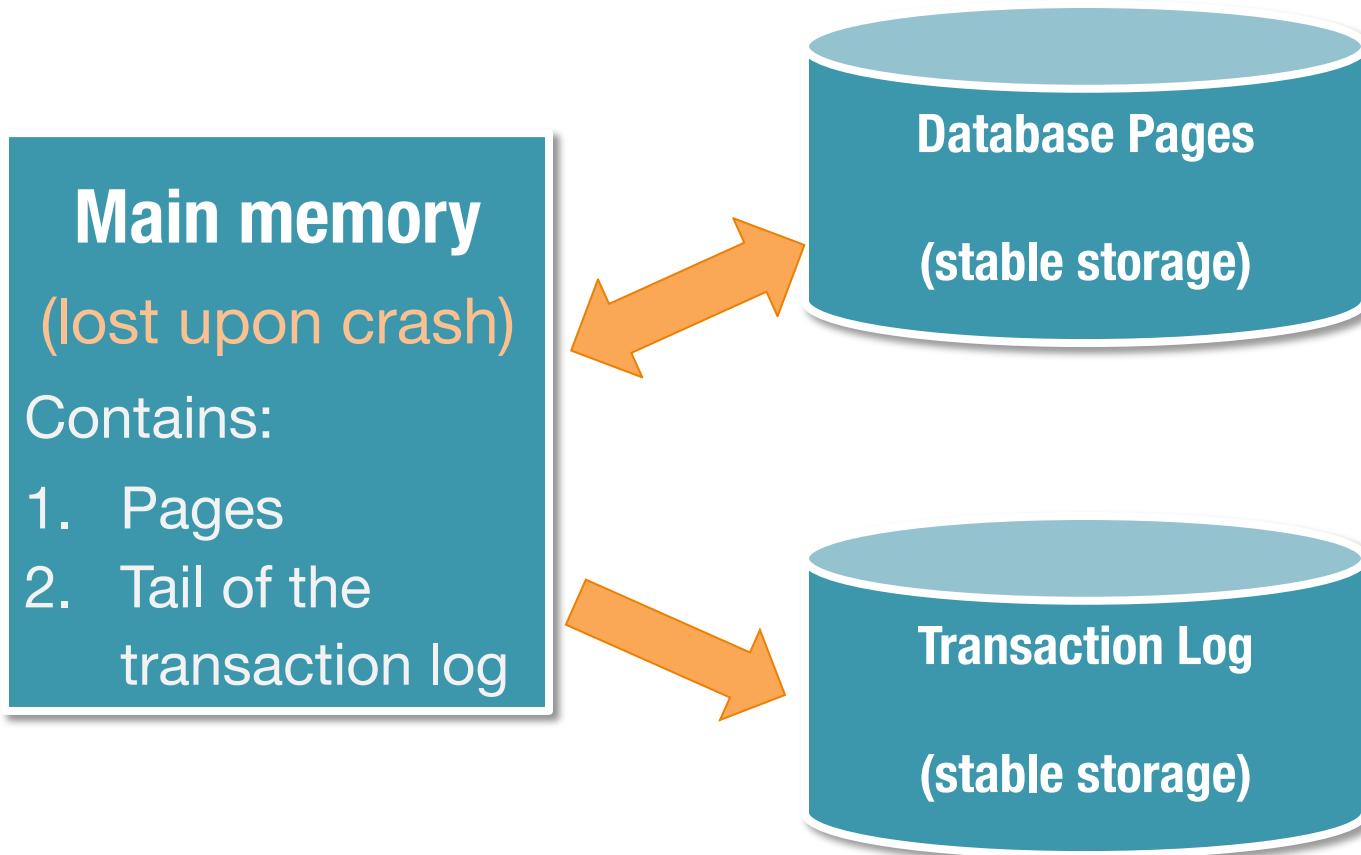


What if there is
a crash for a
committed
transaction T1?

REDO!



Log to Support Undo or Redo



The log is always AHEAD.

Write-Ahead Log (WAL)

- Why is WAL faster than FORCE?
 - Both need to FORCE data to disk as transactions are committed
- Two advantages:
 - WAL records typically **smaller** than database data pages
 - WAL is **sequential**. Traditional disks are great for sequential writes

Example of Update Records

LSN	Txid	Type	Page	Len	Offset	Before	After	PrevLSN
10	T1000	update	P500	3	21	Abc	Def	---
20	T2000	update	P600	3	41	Hij	Klm	---
30	T2000	update	P500	3	20	Gde	Qrs	20
40	T1000	update	P505	3	21	Tuv	Wxy	10

Compensating Log Records

- Describes updates about to be **undone**
- Add a CLR entry to the log for every write that is about to be undone due to abort
 - For UNDO actions.
 - Contains **undoNextLSN**
 - Reverse chain of update logs
 - Contains before-image only (the value being restored).
 - No after-image needed: CLRs never undone, since undone actions are due to aborts (no undo of undo)



Example:

A Log File with Different Types of Record Type

- 00: begin_checkpoint
- 05: end_checkpoint
- 10: update: T1 writes P5 (offset, size, old-image, new-image)
- 20: update: T2 writes P3 (...)
- 30: T1 aborts
- 40: CLR: Undo T1 LSN 10 (offset, size, old-image)
- 45: T1 end
- 50: update: T3 writes P1 (...)
- 60: update: T2 writes P5 (...)
- 70: CLR: Undo T2 LSN 60 (...) -  crash
- 80: CLR: Undo T3 LSN 50
- 85: T3 end (...)
- 90: CLR: Undo T2 LSN 20 (...)
- 95:T2 end

CLRs

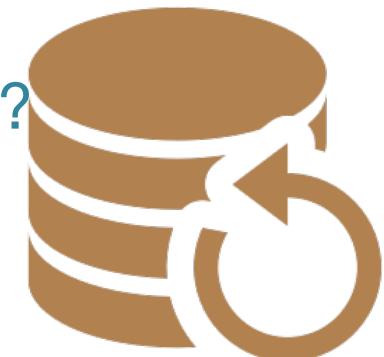


How many CLRs can be written
to log during crash recovery?

Bounded by # of update log entries
for active txs at time of crash

Checkpointing

- Checkpoint: Snapshot of the database
 - Minimize recovery time by limiting log we need to examine
 - After crash, system locates most recent (begin_)checkpoint
- How to Keep it Consistent?
 - Treat it like a transaction
 - This transaction will touch “everything”
 - Bring the system to a halt while checkpointing.
- What if you cannot afford to stop the system?
- Fuzzy Checkpoint
 - Taken while system continues to run
 - Checkpoint may not be a consistent snapshot
 - Keep track of transactions running at the time, and use log



More Checkpointing

- Write to log:
 - `begin_checkpoint` record: Indicates when chkpt began
 - `end_checkpoint` record:
 - Record **Tx table** and **dirty page** table *at time of begin_checkpoint*
 - *No attempt to force dirty pages to disk*
 - `end_checkpoint` can be big
 - This is a `fuzzy checkpoint`
 - `Master record` stores LSN of `begin_checkpoint` record in a safe place so we can jump there on restart

Even More Checkpointing

- **Transaction table** contains (txid, status, lastLSN) for each live tx
 - Points to most recent LSN for each live tx
 - Tells us “*latest* possible *undo* point” for tx
- **Dirty page table** contains (recLSN) for each dirty page in buffer pool.
 - Points to first log record that dirtied the page
 - Tells us “*earliest* possible *redo* point” for page
- These two tables are recovered during restart

Example: Reconstruct the DP + Tx Tables

Page	recLSN
P500	
P600	
P505	

Dirty
Page
Table

LSN	Txid	Type	Page	Len	Offset	Before	After	PrevLSN
1	T1000	update	P500	3	21	Abc	Def	--
2	T2000	update	P600	3	41	Hij	Klm	--
3	T2000	update	P500	3	20	Gde	Qrs	--
4	T1000	update	P505	3	21	Tuv	Wxy	--

Log

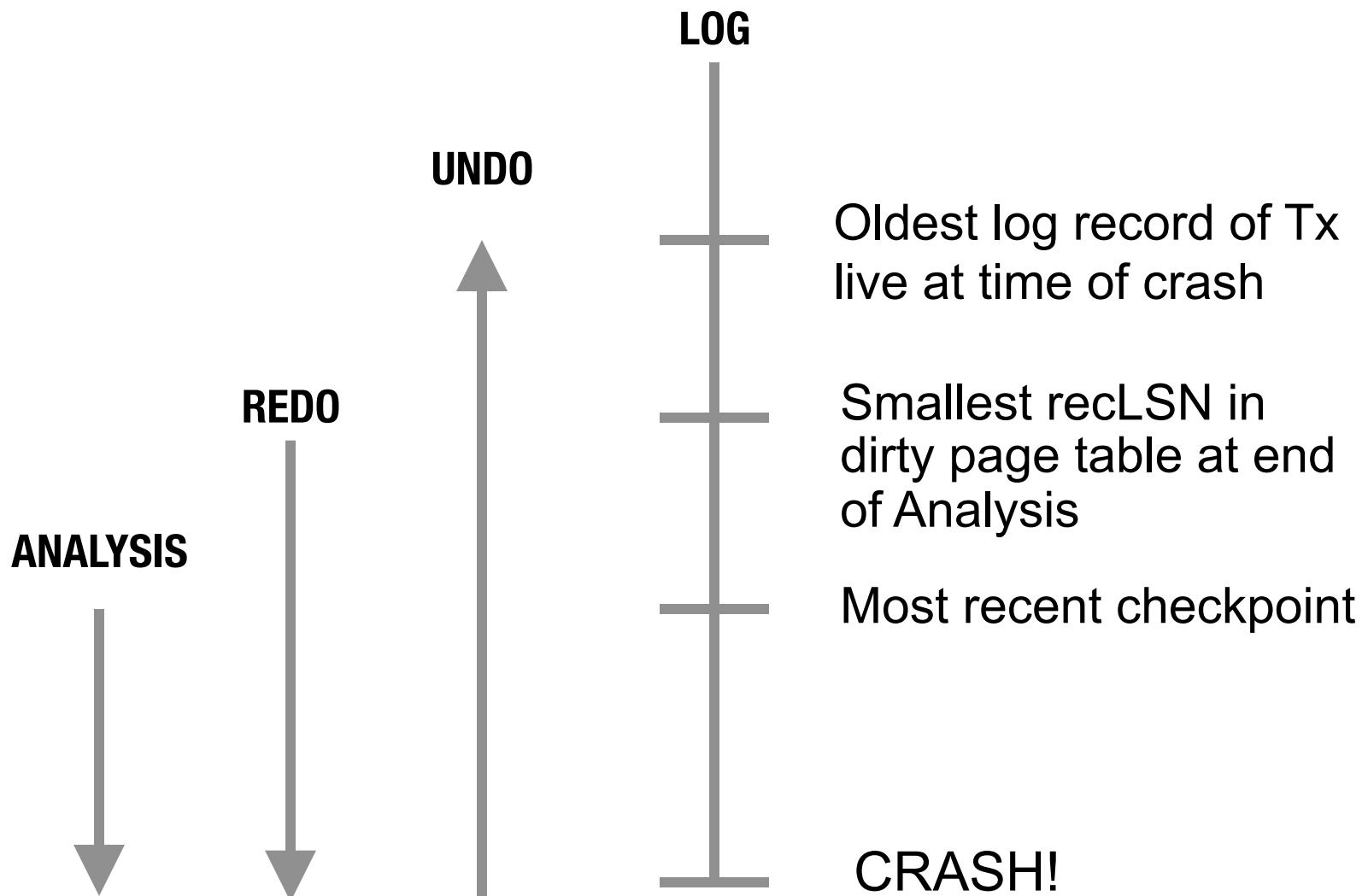
Txid	lastLSN	Status
T1000		U
T2000		U

Tx
Table

How Much Log Is Enough?

- Upon restart, how much log do you have to process?
 - What pages might be dirty and unwritten? (REDO)
 - What pages might be written but uncommitted? (UNDO)
- Undo: latest lastLSN of txs active at crash
- Redo: earliest recLSN in dirty page table at crash

Recovery Process



Recovery: 1. Analysis

- **1. Analysis:** reconstructs tables at time of crash
 - Jump to most-recent checkpoint, (copy the Txtable and DPT) scan forward in log
 - If we find end-tx in log, remove it from tx table (not active)
 - If we find log entry for tx not in table, add it to table
 - If we find log entry that impacts page P, and P is not in dirty page table, add it to that table
 - If we find a commit/abort log, update the tx table for that tx
- At end of Analysis:
 - Tx table is correct
 - Dirty page table is superset of correct (for performance reasons, we don't log each page flush)

Example

LSN	Log Record
10	T1: UPDATE P1 (OLD: YYY NEW: ZZZ)
15	T2: UPDATE P3 (OLD: UUU NEW: VVV)
20	BEGIN CHECKPOINT
25	END CHECKPOINT (XACT TABLE=[[T1,10],[T2,15]]; DPT=[[P1,10],[P3,15]])
30	T1: UPDATE P2 (OLD: WWW NEW: XXX)
35	T1: COMMIT
40	T2: UPDATE P1 (OLD: ZZZ NEW: TTT)
45	T2: ABORT
50	T2: CLR P1(ZZZ), undonextLSN=15
	BOOM!!!#CRASH!!!!

Analysis phase:

- Scan forward through the log starting at LSN 20 (master record)
- LSN 25: Initialize XACT table with T1 (LastLSN 10) and T2 (LastLSN 15). Initialize DPT to P1 (RecLSN 10) and P3 (RecLSN 15)
- LSN 30: Set LastLSN=30 for T1 in XACT table. Add (P2, LSN 30) to DPT
- LSN 35: Change T1 status to "Commit" and its LastLSN to 35 in XACT table
- LSN 40: Set LastLSN=40 for T2 in XACT table -- LSN 10 < 40
- LSN 45: Change T2 status to "Abort" (Uncommitted in Project 4) and its LastLSN=45 in XACT table
- LSN 50: Set LastLSN=50 for T2 in XACT table

Recovery: The Winners

- 2. **Redo:** applies all updates in log (**goes forward**)
 - Start with log record of smallest recLSN of any page in dirty page table; scan forward
 - For each update/CLR encountered, check whether the update has to be applied:
 1. Is this page in the dirty page table?
 2. If yes, is the dirty page entry's recLSN \leq current log LSN?
 3. If yes, read the actual page from disk. Is the, LSN recorded on page (**called PageLSN**) smaller current log LSN?
 - If yes, apply the update/CLR log to this page and set its **PageLSN** to the current log's LSN
 - If the answer to any of the questions above is no, move on
 - Write end records for all committed TX in Txtable and erase them from the Txtable

Example

LSN	Log Record
10	T1: UPDATE P1 (OLD: YYY NEW: ZZZ)
15	T2: UPDATE P3 (OLD: UUU NEW: VVV)
20	BEGIN CHECKPOINT
25	END CHECKPOINT (XACT TABLE=[[T1,10],[T2,15]]; DPT=[[P1,10], [P3,15]])
30	T1: UPDATE P2 (OLD: WWW NEW: XXX)
35	T1: COMMIT
40	T2: UPDATE P1 (OLD: ZZZ NEW: TTT)
45	T2: ABORT
50	T2: CLR P1(ZZZ), undonextLSN=15
	BOOM!!!#CRASH!!!!

Redo Phase:

- Scan forward through the log starting at LSN 10
- LSN 10: Read page P1, check PageLSN stored in the page. If PageLSN<10, redo LSN 10 (set value to ZZZ) and set the page's PageLSN=10
- LSN 15: Read page P3, check PageLSN stored in the page. If PageLSN<15, redo LSN 15 (set value to VVV) and set the page's PageLSN=15
- LSN 30: Read page P2, check PageLSN stored in the page. If PageLSN<30, redo LSN 30 (set value to XXX) and set the page's PageLSN=30
- LSN 40: Read page P1 if it has been flushed, check PageLSN stored in the page. It will be 10. Redo LSN 40 (set value to TTT) and set the page's PageLSN=40
- LSN 50: Read page P1 if it has been flushed, check PageLSN stored in the page. It will be 40. Redo LSN 50 (set value to ZZZ) and set the page's PageLSN=50
- End record is written for T1; Remove T1 from Xact

Recovery: The Losers

- 3. Undo: Scan log **backwards**
 - Identify all live txs at time of crash
 - $ToUndo = \{\text{the LastLSN of all uncommitted live txs}\}$
 - While $ToUndo$ is not empty
 - $L \leftarrow$ Pick the max LSN in $ToUndo$
 - If L is an update record
 - Undo the action of L , write an CLR record to the log with $undonextLSN=L.\text{PrevLSN}$
 - If $L.\text{prevLSN} \neq \text{null}$
 - $ToUndo \leftarrow ToUndo \cup \{L.\text{prevLSN}\}$
 - If $L.\text{prevLSN} = \text{null}$, write an end log for L 's transaction
 - If L is a CLR record
 - If $L.\text{undonextLSN} \neq \text{null}$
 - $ToUndo \leftarrow ToUndo \cup \{L.\text{undonextLSN}\}$
 - If $L.\text{undonextLSN} = \text{null}$, write an end log for L 's transaction
 - If L is abort
 - If $L.\text{prevLSN} \neq \text{null}$ then $ToUndo \leftarrow ToUndo \cup \{L.\text{prevLSN}\}$ otherwise write an end log for L 's transaction
 - Remove L from $ToUndo$

Example

LSN	Log Record
10	T1: UPDATE P1 (OLD: YYY NEW: ZZZ)
15	T2: UPDATE P3 (OLD: UUU NEW: VVV)
20	BEGIN CHECKPOINT
25	END CHECKPOINT (XACT TABLE=[[T1,10],[T2,15]]; DPT=[[P1,10], [P3,15]])
30	T1: UPDATE P2 (OLD: WWW NEW: XXX)
35	T1: COMMIT
40	T2: UPDATE P1 (OLD: ZZZ NEW: TTT)
45	T2: ABORT
50	T2: CLR P1(ZZZ), undonextLSN=15
	BOOM!!!#CRASH!!!!

Undo Phase:

- T2 must be undone. ToUndo = {50}
- LSN 50: Put LSN 15 in ToUndo
ToUndo = {15}
- LSN 15: Undo LSN 15 - write a
CLR for P3 with "set P3=UUU" and
undonextLSN=NULL. Write UUU
into P3
- Write an end record for T2;
Remove T2 from Xact table
- ToUndo = {}

Media Recovery

- Used for disaster recovery
 - Disk crashes, fires, alien attack, etc.
- Periodically make a copy of the database
 - Similar to a “fuzzy checkpoint”
- When an object is corrupted:
 - Get potentially-outdated copy
 - Apply logs to bring it up-to-date



Summary

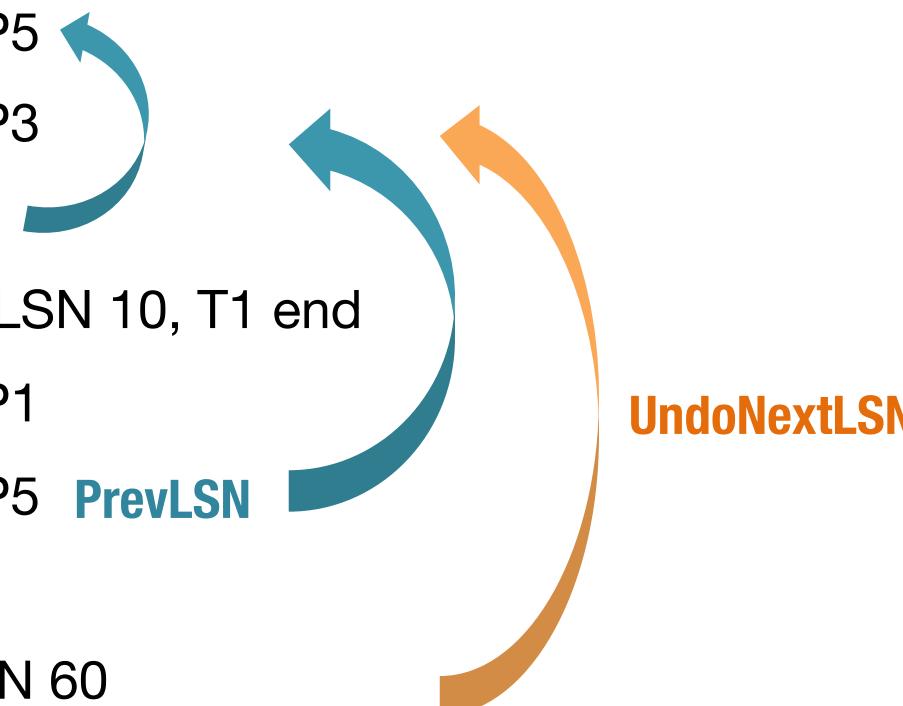
- Atomicity & Durability
- WAL to allow STEAL/NO-FORCE
- Checkpointing: A quick way to limit the amount of log to scan on recovery
- Recovery works in 3 phases:
 - Analysis: Forward from checkpoint
 - Redo: Forward from oldest recLSN
 - Undo: Backward from end to first LSN of oldest Xact alive at crash
- Upon Undo, write CLRs
- Redo “repeats history”
- Interested in the history of ARIES:
http://www.almaden.ibm.com/u/mohan/ARIES_Impact.html

Optional Exercises

- From textbook: 16.1, 16.3, 17.5
- Additional Review Exercise: Recall the three kinds of conflicts from last class (RW, WR, WW). For each, think of an example where the conflict occurs. How does 2PL prevent these conflicts?

Example 3 (Fig 18.5 from textbook)

- 00, 05: begin_checkpoint, end_checkpoint
- 10: update: T1 writes P5
- 20: update: T2 writes P3
- 30: T1 aborts **PrevLSN**
- 40, 45: CLR: Undo T1 LSN 10, T1 end
- 50: update: T3 writes P1
- 60: update: T2 writes P5 **PrevLSN**
- CRASH, Restart
- 70: CLRL: Undo T2 LSN 60
- 80, 85: CLR: Undo T3 LSN 50, T3 end
- CRASH, Restart
- 90, 95: CLR: Undo T2 LSN 20, T2 end



Example 4

LSN	Log Record
0	BEGIN CHECKPOINT
5	END CHECKPOINT (EMPTY XACT TABLE AND DPT)
10	T1: UPDATE P1 (OLD: YYY NEW: ZZZ)
15	T1: UPDATE P2 (OLD: WWW NEW: XXX)
20	T1: COMMIT
	BOOM!!!#CRASH!!!!

Analysis phase:

- Scan forward through the log starting at LSN 0
- LSN 5: Initialize XACT table and DPT to empty
- LSN 10: Add (T1, LSN 10) to XACT table. Add (P1, LSN 10) to DPT
- LSN 15: Set LastLSN=15 for T1 in XACT table. Add (P2, LSN 15) to DPT
- LSN 20: Change T1 status to "Commit" in XACT table

Redo phase:

- Scan forward through the log starting at LSN 10
- LSN 10: Read page P1, check PageLSN stored in the page. If PageLSN<10, redo LSN 10 (set value to ZZZ) and set the page's PageLSN=10
- LSN 15: Read page P2, check PageLSN stored in the page. If PageLSN<15, redo LSN 15 (set value to XXX) and set the page's PageLSN=15
- Create End log record for T1 (LSN 25) and erase T1 from the TxTable

Undo phase:

- Do nothing; no transactions to undo

Example 5

Analysis phase:

LSN	Log Record
0	BEGIN CHECKPOINT
5	END CHECKPOINT (EMPTY XACT TABLE AND DPT)
10	T1: UPDATE P1 (OLD: YYY NEW: ZZZ)
15	T1: UPDATE P2 (OLD: WWW NEW: XXX)
20	T2: UPDATE P3 (OLD: UUU NEW: VVV)
25	T1: COMMIT
30	T2: UPDATE P1 (OLD: ZZZ NEW: TTT)
	BOOM!!!#CRASH!!!!

- Scan forward through the log starting at LSN 0
- LSN 5: Initialize XACT table and DPT to empty
- LSN 10: Add (T1, LSN 10) to XACT table. Add (P1, LSN 10) to DPT
- LSN 15: Set LastLSN=15 for T1 in XACT table. Add (P2, LSN 15) to DPT
- LSN 20: Add (T2, LSN 20) to XACT table. Add (P3, LSN 20) to DPT
- LSN 25: Change T1 status to "Commit" and its LastLSN to 25 in XACT table
- LSN 30: Set LastLSN=30 for T2 in XACT table

Example 5

Redo phase:

LSN	Log Record
0	BEGIN CHECKPOINT
5	END CHECKPOINT (EMPTY XACT TABLE AND DPT)
10	T1: UPDATE P1 (OLD: YYY NEW: ZZZ)
15	T1: UPDATE P2 (OLD: WWW NEW: XXX)
20	T2: UPDATE P3 (OLD: UUU NEW: VVV)
25	T1: COMMIT
30	T2: UPDATE P1 (OLD: ZZZ NEW: TTT)
	BOOM!!!#CRASH!!!!

- Scan forward through the log starting at LSN 10 (smallest LSN on DPT)
- LSN 10: Read page P1, check PageLSN stored in the page. If PageLSN<10, redo LSN 10 (set value to ZZZ) and set the page's PageLSN=10
- LSN 15: Read page P2, check PageLSN stored in the page. If PageLSN<15, redo LSN 15 (set value to XXX) and set the page's PageLSN=15
- LSN 20: Read page P3, check PageLSN stored in the page. If PageLSN<20, redo LSN 20 (set value to VVV) and set the page's PageLSN=20
- LSN 30: Read page P1 if it has been flushed, check PageLSN stored in the page. It will be 10
- Redo LSN 30 (set value to TTT) and set the page's PageLSN=30
- Write end record for T1; Remove T1 from Xact table

Example 5

LSN	Log Record
0	BEGIN CHECKPOINT
5	END CHECKPOINT (EMPTY XACT TABLE AND DPT)
10	T1: UPDATE P1 (OLD: YYY NEW: ZZZ)
15	T1: UPDATE P2 (OLD: WWW NEW: XXX)
20	T2: UPDATE P3 (OLD: UUU NEW: VVV)
25	T1: COMMIT
30	T2: UPDATE P1 (OLD: ZZZ NEW: TTT)
	BOOM!!!#CRASH!!!!

Undo phase:

- Since TxTable only has T2, start from LastLSN of T2 obtained from TxTable (LSN 30)
- T2 must be undone. Put LSN 30 in ToUndo
- LSN 30: Undo LSN 30 - write a CLR for P1 with "set P1=ZZZ" and undonextLSN=20. Write ZZZ into P1. Put LSN 20 in ToUndo
- LSN 20: Undo LSN 20 - write a CLR for P3 with "set P3=UUU" and undonextLSN=NULL. Write UUU into P3
- Write 'End' record for T2; remove T2 from XACT table