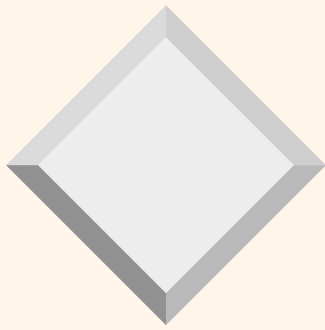


# *Reference to slides for Aries*

- <http://redbook.cs.berkeley.edu/redbook3/aries/aries.ppt>





# *Logging and Recovery*

These Slides are from Berkeley with  
modifications of Rao Kotagiri

“If you are going to be in the logging business, one of the things that you have to do is to learn about heavy equipment.”

Robert VanNatta,  
*Logging History of Columbia County*

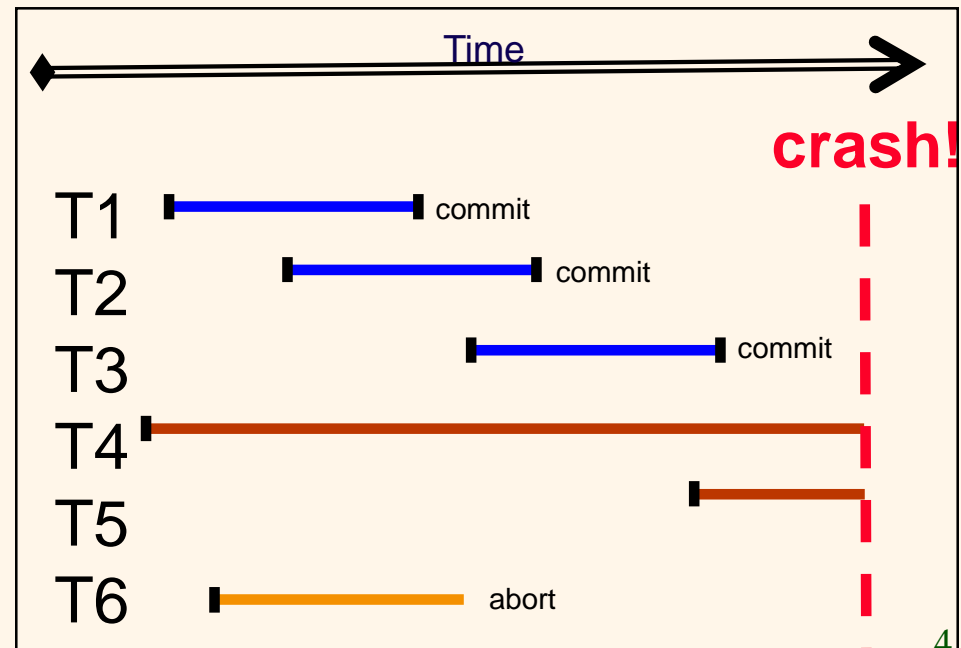



# *Review: The ACID properties*

- ❑ **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.

# Motivation


- Atomicity:
  - Transactions may abort (“Rollback”). E.g. T6
- Durability:
  - What if DBMS stops running? (Causes?)
- Desired Behavior after system restarts:
  - T1, T2 & T3 should be **durable** as they are committed before the crash.
  - T4, T5, T6 should be **aborted** (effects not seen).





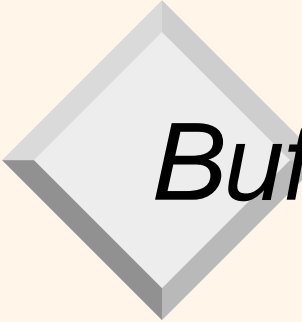
# *Assumptions*

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



# *Buffer Caches (pool)*

1. Data is stored on disks
2. Reading a data item requires reading the whole page of data (typically 4K or 8K bytes of data depending on the page size) from disk to memory containing the item.
3. Modifying a data item requires reading the whole page from disk to memory containing the item, modifying the item in memory and writing the whole page to disk.
4. Steps 2 & 3 can be very expensive and we can minimize the number of disk reads and writes by storing as many disk pages as possible in memory (buffer cache) – this means always check in buffer cache for the disk page of interest if not copy the associated page to buffer cache and perform the necessary operation.
5. When buffer cache is full we need to evict some pages from the buffer cache in order fetch the required pages from the disk .

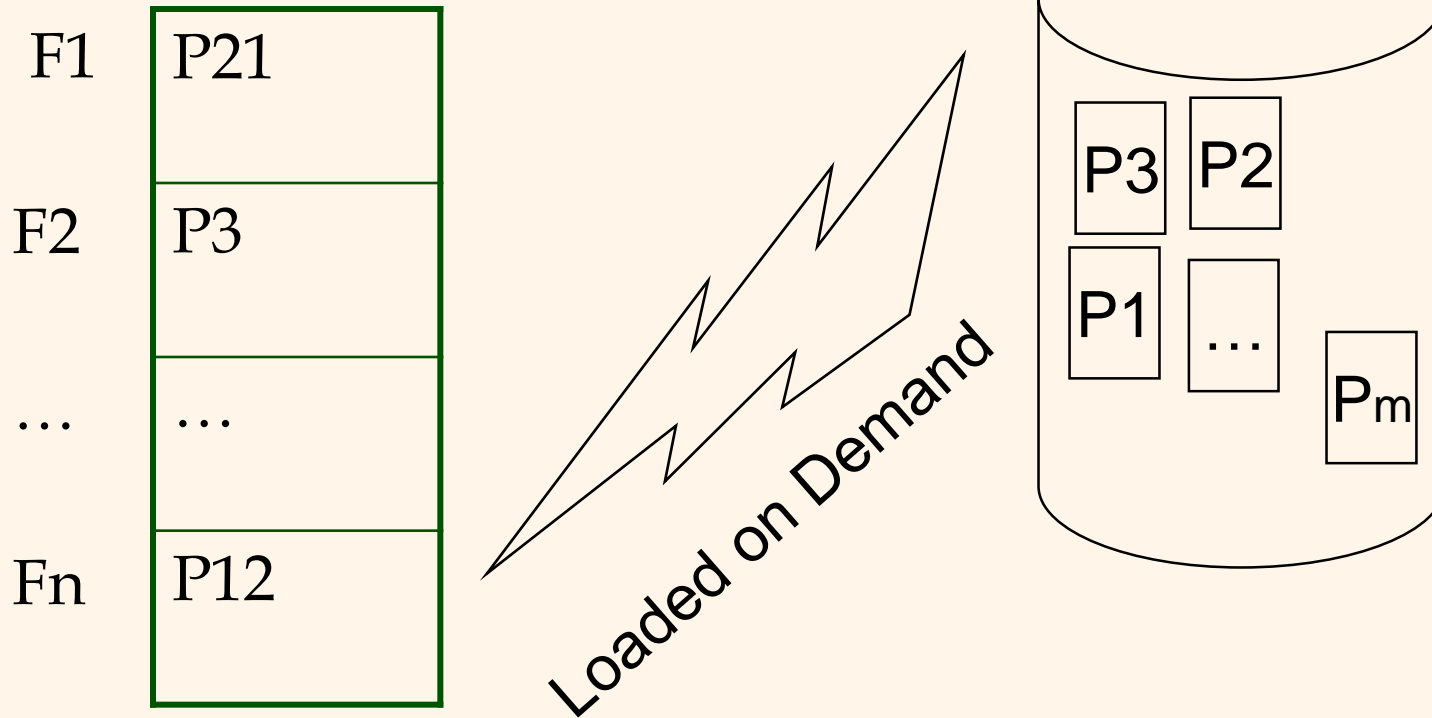


# *Buffer Caches (pool)*

6. Eviction needs to make sure that no one else is using the page and any modified pages should be copied to the disk.
  7. Since several transactions are executing concurrently this requires additional locking procedures using latches. These latches are used only for the duration of the operation (e.g. READ/WRITE) and can be released immediately unlike record locks which have to be kept locked until the end of the transaction.
- `fix(pageid)`
    - reads pages from disk into the buffer cache if it is not already in the buffer cache
    - fixed pages cannot be dropped from the buffer cache as transactions are accessing the contents
  - `unfix(pageid)`
    - The page is not in use by the transaction and can be evicted as far as the calling transaction is concerned. ( We need to check to see that no one else wants the page before it can be evicted)

# Main components of a Database System

## □ Buffer manger





# *Main components of a Database System*

## Lock manager

| Object Id  | Ref to lock details |
|------------|---------------------|
| Tuple A    |                     |
| ...        |                     |
| Relation X |                     |
| Page P7    |                     |

Set of database objects: e.g. tuples, pages, relations, indexes

Lock created on Demand

# Handling the Buffer Pool (cache)

- **Force** write to disk at commit?
  - Poor response time.
  - But provides durability.
- **NO Force** leave pages in memory as long as possible even after commit without modifying the data on the disk.
  - Improves response time and efficiency as many reads and updates can take place in main memory rather than on disk.
  - Durability becomes a problem as update may be lost if a crash occurs
- **Steal** buffer-pool frames from uncommitted Xacts?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?

|          | No Steal  | Steal   |
|----------|---|---------|
| Force    | Trivial<br>(that is<br>performing<br>only step2 &3) |         |
| No Force |   | Desired |

That is a page modified by a transaction is written to disk but the transaction decides to abort!



# *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 **UNDO**ing 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 **REDO**ing modifications.

# *Basic Idea: Logging*



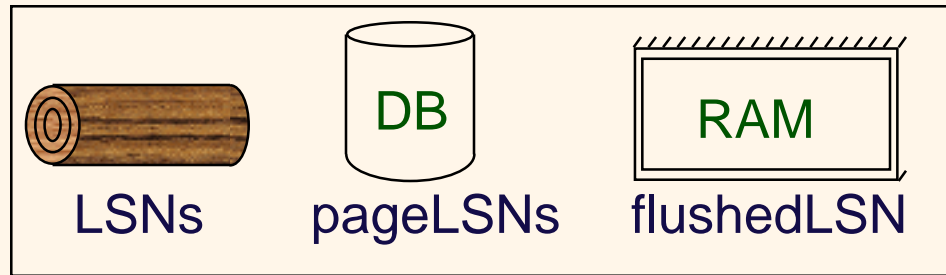
- Record REDO (new value) and UNDO (old value) information, for every update, in a *log*.
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.
- Log: An ordered list of REDO/UNDO actions
  - Log record contains: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



# Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  1. Must **force** the **log record** which has both old and new values for an update before the corresponding **data page** gets to disk (**stolen**).
  2. Must **write all log records to disk (force)** for a Xact before commit.
- 1. guarantees Atomicity because we can undo updates performed by aborted transactions and redo those updates of committed transactions.
- 2. guarantees Durability.
- Exactly how is logging (and recovery!) done?
  - We 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.

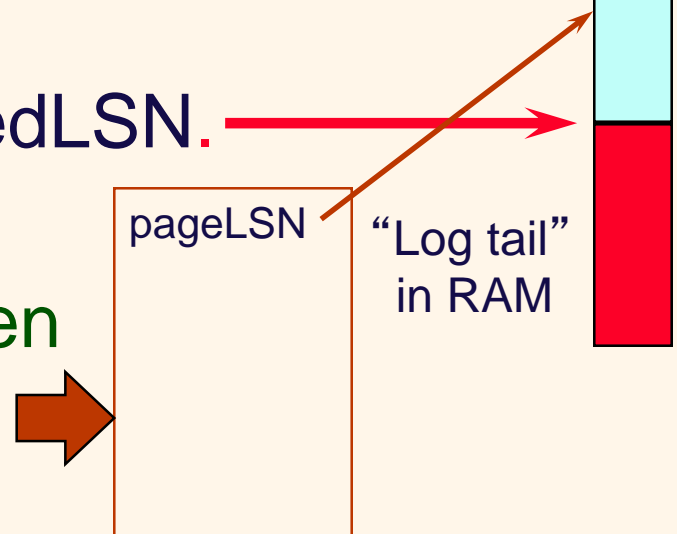
- The LSN of the most recent *log record* for an update to that page.

- System keeps track of flushedLSN.

- The max LSN flushed so far.

- WAL: Before a page is written to disc make sure  $\text{pageLSN} \leq \text{flushedLSN}$

Log records  
flushed to disk



# Log Records

## LogRecord fields:

prevLSN  
XID  
type  
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



# *Other Log-Related State*

- Transaction Table:
  - One entry per active Xact.
  - Contains XID, status (running/committed/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 since loaded into the buffer cache from the disk.



# *Instance of Log, Transaction and Page tables*

Dirty Page table

| Page # | Oldest LSN (Recent LSN) |
|--------|-------------------------|
|        |                         |
|        |                         |
|        |                         |

X-table

| Xid | Status | Last LSN |
|-----|--------|----------|
|     |        |          |
|     |        |          |

Log

| Prev LSN | Tid | Type | Page | Length | Offset | Old Value | New Value |
|----------|-----|------|------|--------|--------|-----------|-----------|
|          |     |      |      |        |        |           |           |



# Instance of Log, Transaction and Page tables

Dirty Page table

| Page # | Oldest LSN (Recent LSN) |
|--------|-------------------------|
| P5     |                         |
|        |                         |
|        |                         |

X-table



| Xid | Status | Last LSN |
|-----|--------|----------|
| T1  | R      |          |
|     |        |          |

Log



| Prev LSN | Tid | Type | Page | Length | Offset | Old Value | New Value |
|----------|-----|------|------|--------|--------|-----------|-----------|
|          | T1  | U    | p5   | 4      | 20     | abcd      | ABCD      |

# Instance of Log, Transaction and Page tables

Dirty Page table

| Page # | Oldest LSN<br>(Recent LSN)  |
|--------|---|
| P5     |  |
| P6     |  |
|        |   |

X-table



| Xid | Status | Last LSN  |
|-----|--------|---|
| T1  | R      |  |
| T2  | R      |  |

Log



| Prev LSN | Tid | Type | Page | Length | Offset | Old Value | New Value |
|----------|-----|------|------|--------|--------|-----------|-----------|
|          | T1  | U    | p5   | 4      | 20     | abcd      | ABCD      |
|          | T2  | U    | p6   | 4      | 40     | efgh      | EFGH      |

# Instance of Log, Transaction and Page tables




Dirty Page table

| Page # | Oldest LSN<br>(Recent LSN)  |
|--------|---|
| P5     |  |
| P6     |  |
|        |   |

X-table

| Xid | Status | Last LSN  |
|-----|--------|---|
| T1  | R      |  |
| T2  | R      |  |

Log

| Prev LSN  | Tid | Type | Page | Length | Offset | Old Value | New Value |
|---|-----|------|------|--------|--------|-----------|-----------|
|    | T1  | U    | p5   | 4      | 20     | abcd      | ABCD      |
|    | T2  | U    | p6   | 4      | 40     | efgh      | EFGH      |
|  | T2  | U    | p5   | 4      | 80     | jklm      | JKLM      |

# Instance of Log, Transaction and Page tables

Dirty Page table

| Page # | Oldest LSN (Recent LSN) |
|--------|-------------------------|
| P5     |                         |
| P6     |                         |
| P7     |                         |

Log

| Prev LSN | Tid | Type | Page | Length | Offset | Old Value | New Value |
|----------|-----|------|------|--------|--------|-----------|-----------|
|          | T1  | U    | p5   | 4      | 20     | abcd      | ABCD      |
|          | T2  | U    | p6   | 4      | 40     | efgh      | EFGH      |
|          | T2  | U    | p5   | 4      | 80     | jklm      | JKLM      |
|          | T1  | U    | p7   | 4      | 20     | nopq      | NOPQ      |

X-table

| Xid | Status | Last LSN |
|-----|--------|----------|
| T1  | R      |          |
| T2  | R      |          |



# *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. We discussed how we do this earlier.
- Strict 2PL.
- STEAL, NO-FORCE buffer management, with Write-Ahead Logging.



# 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 is limited by the 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*).

# *The Big Picture: What's Stored Where*



## **LogRecords**

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

**master record**



## **Data pages**

each  
with a  
pageLSN



## **Xact Table**

lastLSN  
status

## **Dirty Page Table**

recLSN

**flushedLSN**

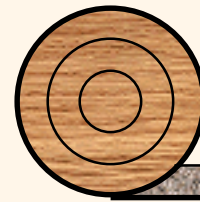




# *Simple Transaction Abort*

- 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.



Currently UNDOing  
PrevLSN=1234

lastLSN (CLR)  
undonextLSN=1234

- To perform UNDO, must have a lock on data!
  - No problem!
- Before restoring old value of a page, write a CLR (Compensation Log Record):
  - 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 *never* Undone (but they might be Redone when repeating history: guarantees Atomicity!)
- At end of UNDO, write an “end” log record.

# Example of Transaction Abort

RAM

Xact Table

lastLSN

status

Dirty Page Table


recLSN

flushedLSN

ToUndo

| LSN | LOG                  |
|-----|----------------------|
| 400 | begin_checkpoint     |
| 405 | end_checkpoint       |
| 410 | update: T1 writes P5 |
| 420 | update T2 writes P3  |
| 430 | T1 abort             |
| 440 | CLR: Undo T1 LSN 410 |
| 445 | T1 End               |
| 450 | update: T3 writes P1 |
| 460 | update: T2 writes P5 |

prevLSNs



# Transaction Commit

- Write **commit** record to log.
- All log records up to Xact's **lastLSN** are flushed.
  - Guarantees that **flushedLSN**  $\geq$  **lastLSN**.
  - Note that log flushes are sequential, synchronous writes to disk – (very fast writes to disk).
  - Many log records per log page – (very efficient due to multiple writes).
- Commit() returns.
- Write **end** record to log.

# Instance of Log, Transaction and Page tables

Dirty Page table

Log

| Page # | Oldest LSN (Recent LSN) |
|--------|-------------------------|
| P5     | ●                       |
| P6     | ●                       |
| P7     | ●                       |

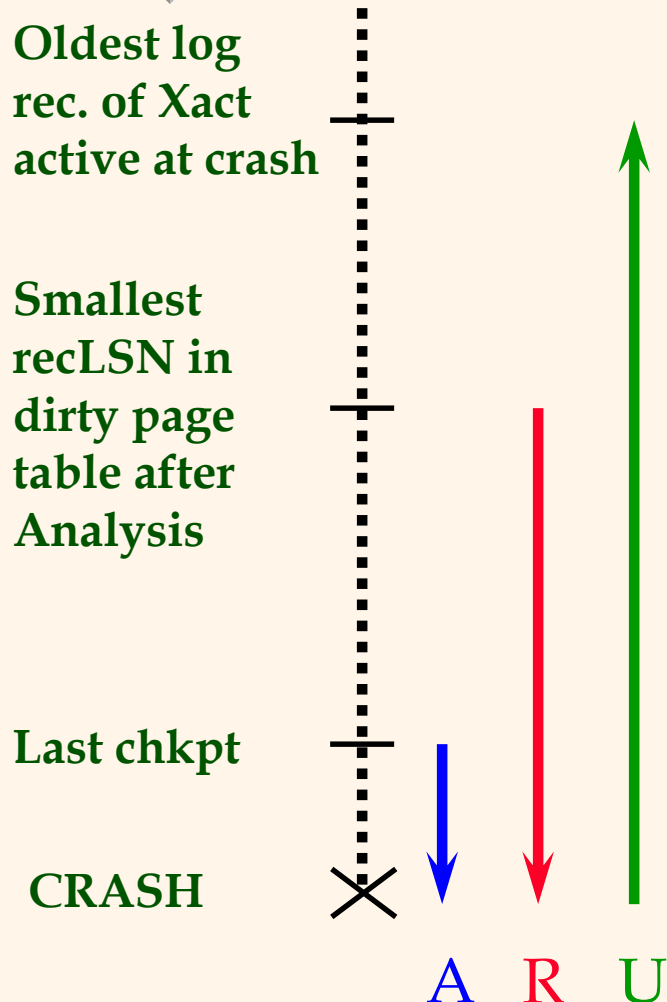
X-table

| Xid | Status | Last LSN |
|-----|--------|----------|
| T1  | R      | ●        |
| T2  | R      | ●        |

| PrevL SN | Tid | Type   | Page | Length | Offset | Old Value | New Value |
|----------|-----|--------|------|--------|--------|-----------|-----------|
|          | T1  | U      | p5   | 4      | 20     | abcd      | ABCD      |
|          | T2  | U      | p6   | 4      | 40     | efgh      | EFGH      |
|          | T2  | U      | p5   | 4      | 80     | jklm      | JKLM      |
|          | T1  | U      | p7   | 4      | 20     | nopq      | NOPQ      |
|          | T2  | Commit |      |        |        |           |           |

We have to flush the log to this point because of commit of T2 and making T2 durable.

# 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.



# 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**.

X-table

| Xid | Status | Last LSN |
|-----|--------|----------|
| T1  | R      |          |
| T2  | R      |          |

Dirty Page table

| Page # | Oldest LSN (Recent LSN) |
|--------|-------------------------|
| P5     |                         |
| P6     |                         |
| P7     |                         |



# Recovery: The REDO Phase

- We *repeat History* to reconstruct state at crash:
  - Reapply *all* updates (even of aborted Xacts!), redo CLR's.
- 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*. Why?
- To *REDO* an action:
  - Reapply logged action. (Note: This happens in the buffer pool.)
  - Set *pageLSN* to *LSN*. No additional logging! Why?



# Instance of Log, Transaction and Page tables

Dirty Page table

| Page # | Oldest LSN (Recent LSN) |
|--------|-------------------------|
|        |                         |
| P6     | •                       |
| P7     | •                       |

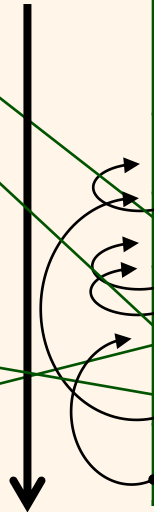
Log


| PrevL SN | Tid | Type   | Page | Length | Offset | Old Value | New Value |
|----------|-----|--------|------|--------|--------|-----------|-----------|
|          | T1  | U      | p5   | 4      | 20     | abcd      | ABCD      |
|          | T2  | U      | p6   | 4      | 40     | efgh      | EFGH      |
|          | T2  | U      | p5   | 4      | 80     | jklm      | JKLM      |
|          | T1  | U      | p7   | 4      | 20     | nopq      | NOPQ      |
|          | T2  | Commit |      |        |        |           |           |

Redo from here

X-table

| Xid | Status | Last LSN |
|-----|--------|----------|
| T1  | R      | •        |
| T2  | R      | •        |





# *Recovery: The UNDO Phase*

$\text{ToUndo} = \{ l \mid l \text{ is a lastLSN of a "loser" Xact} \}$

We can form this list from Xtable.

## **Repeat:**

- Choose the 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
  - (Q: what happens to other CLR's?)
- Else this LSN is an **update**. Undo the update, write a CLR, add **prevLSN** to ToUndo.

**Until ToUndo is empty.**

# Example of Recovery



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 |
|----|---------------------|

|    |        |
|----|--------|
| 45 | T1 End |
|----|--------|

|    |                      |
|----|----------------------|
| 50 | update: T3 writes P1 |
|----|----------------------|

|    |                      |
|----|----------------------|
| 60 | update: T2 writes P5 |
|----|----------------------|

✗ CRASH, RESTART

prevLSNs

# Example: Crash During Restart!



Xact Table

lastLSN

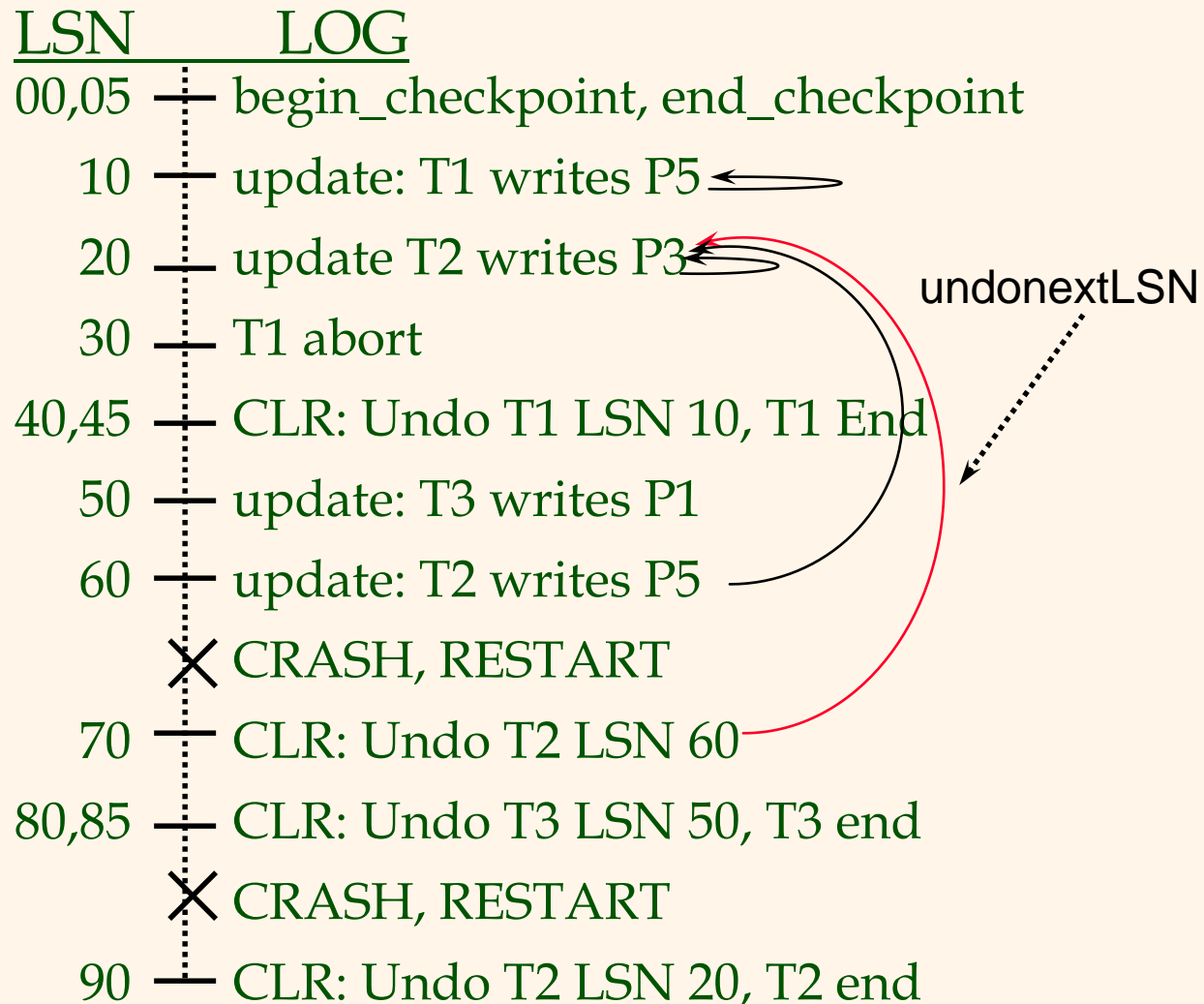
status

Dirty Page Table

recLSN

flushedLSN


ToUndo





# *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.



# *Summary of Logging/Recovery*

- **Recovery Manager** guarantees Atomicity & Durability.
- Use WAL to allow STEAL/NO-FORCE without sacrificing correctness.
- LSNs identify log records; linked into backwards chains per transaction (via prevLSN).
- pageLSN allows comparison of data page and log records.

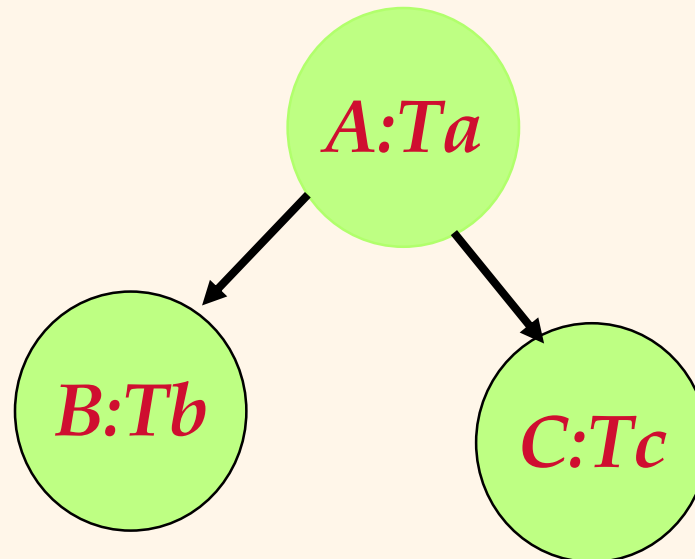


## *Summary, Cont.*

- **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 CLR.s.
- Redo “repeats history”: Simplifies the logic!

# *Distributed Recovery*

- A transaction *T* containing three sub-transactions, *T<sub>a</sub>*, *T<sub>b</sub>* and *T<sub>c</sub>* are run on three nodes *A*, *B* and *C* respectively. The transaction initiating node, in the example Node *A*, is called the master node (or the coordinator of the transaction).







# *Distributed Recovery*

- New kinds of failure such as failure of communication links and failure of remote sites where parts of the transaction is run.
- We need to ensure that all sub-transactions of distributed transaction must commit or none must commit.



# *Distributed Recovery*

## Two phase commit

(Note: not to be confused with 2-phase locking!)

- When a user decides to commit a transaction the following messages are sent:
  - Phase 1
    - The coordinator sends a prepare message to each subordinate
    - On receiving a prepare message a subordinate decides to either commit or abort its sub-transaction. It force-writes an abort or prepare log record and sends yes or no message to the coordinator.



# *Distributed Recovery ...*

## □ Phase2:

- If the coordinator receives all “yes” messages from all subordinates it force-writes a commit log record and then sends commit messages to all subordinates. If it receives even one no message or does not receive a message within some specified time it force-writes an abort log and sends abort message to all subordinates.
- When a subordinate receives an abort message it force-writes an abort log record and sends an acknowledgement to the coordinator. It aborts the sub-transaction.



# *Restart after a Failure*

On recovery, the recovery node does the following:

- If we have a commit or abort log of T, the status is clear. We redo or undo T as in centralized database systems. The coordinator needs sending messages to its subordinates abort or commit messages until it receives acknowledgements from them.
- If we have only prepare log for T but no commit or abort log record and the site is a subordinate we must repeatedly contact the the coordinator to respond for commit or abort instruction. After receiving a message the rest of actions are 2PC.



## *Restart after a Failure*

- If there are no prepare, commit or abort log records the subordinate aborts the transaction unilaterally. If the site is the coordinator it then has to inform all subordinates to abort their sub-transactions.