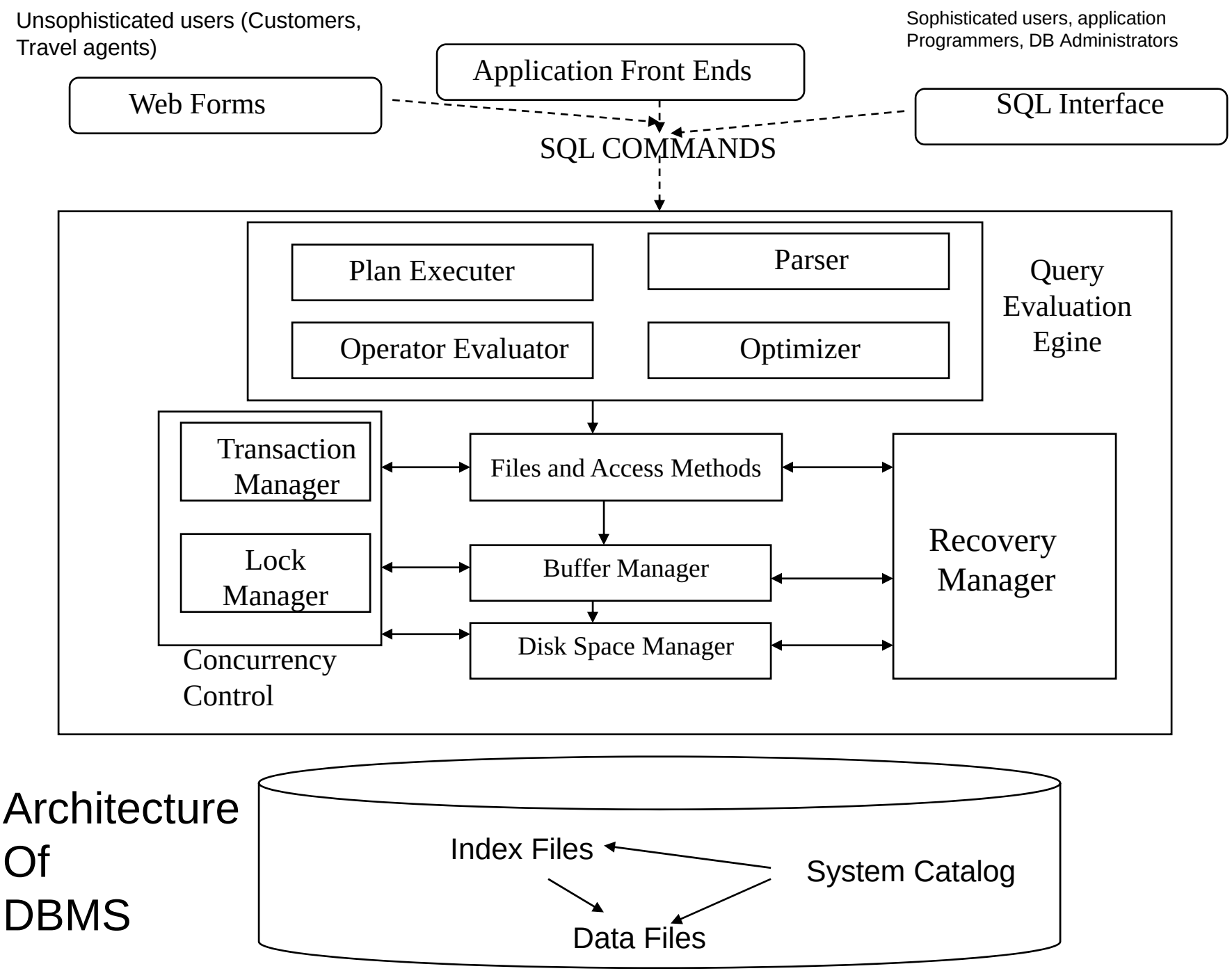
# Crash Recovery

## Chapter 20




# Architecture Of DBMS

# Review: The ACID properties

- Atomicity: All actions in the Xact happen, or none happen.
- 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)**

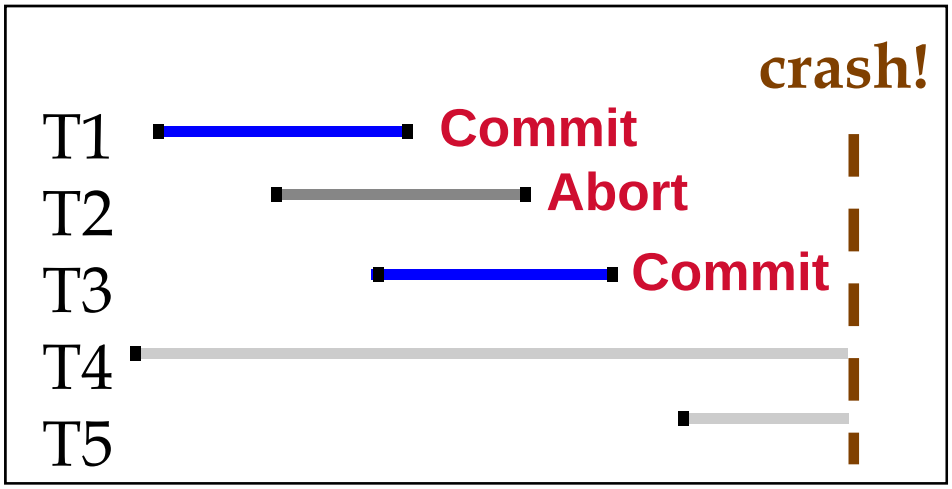
# Motivation

- **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).



# 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 happens when a transaction commits?
  - What happens when a transaction aborts?

# Steal and Forcing

- **With respect to writing the objects, two questions arise.**
  - Can the changes made to an object O in the buffer pool by a transaction T be written to disk before T commits ? Such writes are executed if some other transaction wants to use O.
    - If such writes are allowed it is called a STEAL approach
  - When a transaction commits, must we ensure that all the changes it has made to objects in the buffer pool are immediately forced to disk ? If yes FORCE approach is used.

# 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.

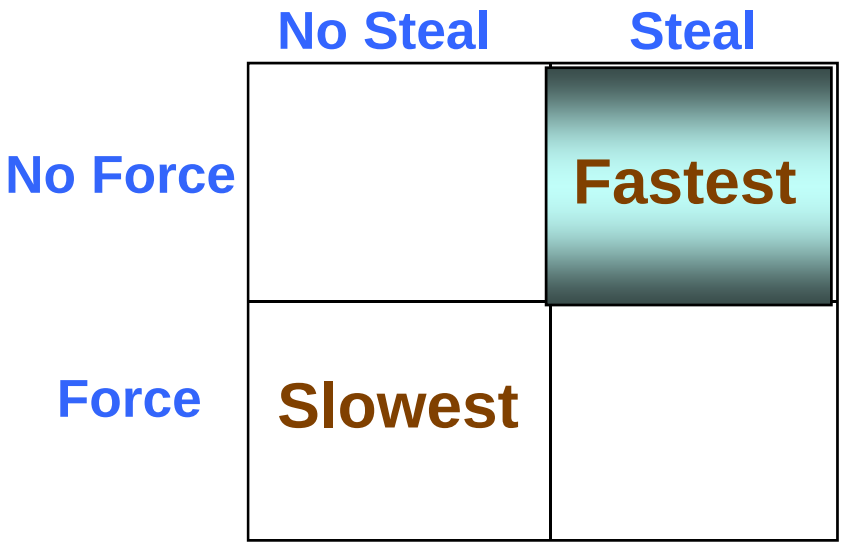
- **No Steal policy – don't allow buffer-pool frames with uncommited updates to overwrite committed data on disk.**
  - Useful for ensuring atomicity without UNDO logging.
  - But can cause poor performance.

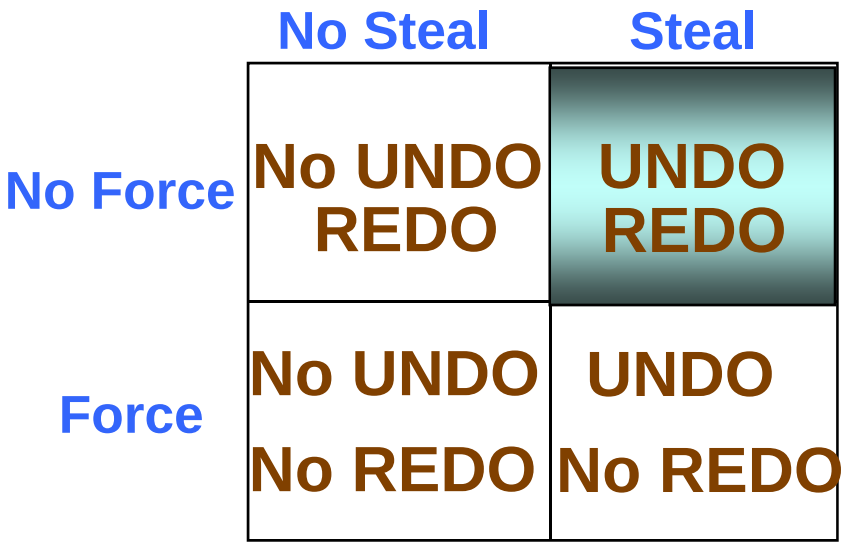
Of course, there are some details for even getting Force/NoSteal to work…

# 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 udpates aborts?
  - What if system crashes before Xact is finished?
  - Must remember the old value of P (to support UNDOing the write to page P).

# Buffer Management summary

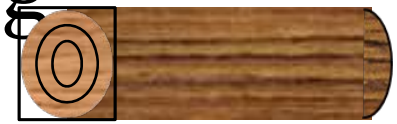



Performance Implications

Logging/Recovery Implications

# 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).

# 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 force all log records for a Xact *before commit*. (I.e. 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.

# ARIES (***Algorithms*** for Recovery and Isolation Exploiting Semantics)

- When the recovery manager is invoked after a crash, restart proceeds in the following phases
  - **Analysis:** Identifies dirty pages in the buffer pool (i.e., changes that have not been to disk) and active transactions at the time of crash
  - **Redo:** Repeats all actions, starting from appropriate point in the log, and restores the database state to what it was at the time of crash
  - **Undo:** Undoes the actions of all transactions that did not commit, so that the database reflects only the actions of committed transactions

# Example

| LSN | LOG |
|---|---|
| 10 | Update: T1 writes P5 |
| 20 | Update: T1 writes P3 |
| 30 | T2 commit |
| 40 | T2 end |
| 50 | update: T3 writes P1 |
| 60 | update: T3 write P3 |

CRASH, RESTART

The Analysis phase identifies T1 and T3 as transactions Active at the time of crash and to be undone; T2 is a committed Transaction, so its effects should be written to disk

# Principles of ARIES

- **Write-ahead logging:**
  - Any change to a database object is first recorded in the log; The log must be written into stable storage before change to the data object is written to the disk.
- **Repeating history During Redo:**
  - On re-start after the crash, ARIES retraces all actions of the DBMS before the crash and brings the system back to the exact state that it was at the time of the crash. It undoes the actions of all transactions still active at the time of crash (by aborting them).
- **Logging changes During Undo:**
  - Changes made to the database while undoing a transaction are logged to ensure such an action is not repeated in the event of repeated (failure causing) restarts**.**

# The Log

- History of actions executed by DBMS
- Log is a file of records stored in stable storage, which is assumed to survive crashes (maintaining two copies on two disks)
- Log tail: most recent portion of log, which is kept is main memory and periodically forced to stable storage.
- Every log record is given a unique Id called the **log sequence number** (LSN).
  - LSNs are assigned in an increasing order.
- **pageLSN:** LSN of the most recent log record that describes a change to this page.

# The Log Record is written..

- **Updating a page:**
  - After modifying the page, an update type record is appended to log tail. The pageLSN of the page is then set to the LSN of the update log record.
- **Commit:**
  - When a transaction decides to commit, it force-writes a commit type log record containing the transaction Id.
    - The log record is appended
    - Log tail is written to stable storage up to and including the commit record. (Once the commit record is written, the transaction is considered as committed).
- **Abort:**
  - When a transaction is aborted, an abort type log record containing transaction id is appended in the log. Undo is initiated for this transactions.
- **End:**
  - When a transaction is committed or aborted, some additional actions must be taken beyond writing commit/abort recond in the log. After all these steps are completed, the end type log record containing transaction ID is appended to the log.
- **Undoing an update**:
  - When a transaction is rolled-back, its updates were undone. When the action described by update log record is undone, a compensating log record (CLR) is written.

# Contents of Log Records

| prevLSN | transID | Type | PageID | Lengh | Offset | Before-Image | After-image |
|---|---|---|---|---|---|---|---|

- **Field of log record:**
  - prevLSN
  - transID
  - Type
  - PageID: Page ID of modified page
  - Length: Length in bytes
  - Offset: offset
  - Before-image: The value of changed-bytes before the change
  - After-image: The value of changed bytes after change
- **Update log record** contains information for both redoing and undoing.
- **A redo-only update log record** contains only after-image.
- **An undo-only update record** contains just the before-image.

# Compensation Log Records (CLRs)

- A CLR is written just before the change recorded in an update log record U is undone.
  - It can happen during normal system execution or recovery from crash.
- A compensation log record C describes the action taken to undo the actions recorded in the corresponding update log record and is appended to the log tail.
- The compensating log record also contains a field called **undoNextLSN**, which is the LSN of the next log record that is to be undone.
- Unlike an update log record, a CLR describes an action that will be never undone. We will never undo an undo action.
  - So, the number of CLRs that can be written during Undo is no more than the number of active transactions at the time of crash.
  - A CLR may be written to stable storage but the undo action it describes may not yet been written to disk when the system crashes again. So, the undo action is reapplied during the Redo phase.

# Other Recovery-Related Structures

- **Transaction Table:**
  - Contains one entry for each active transaction
  - It contains transaction ID and lastLSN which the LSN of the recent log record for the transaction.
  - The status of the transaction is that it is in progress, committed or aborted.
- **Dirty Page table**
  - It contains one entry for each dirty page in the buffer pool
  - Page that changes not yet reflected on disk.
  - The entry contains recLSN, which is the LSN of the first record that caused the page dirty.
- During normal operation, these tables are maintained by transaction manager and buffer manager.
- During restart operation, these tables are reconstructed in the analysis phase of restart.

# Example

| pageID | recLSN |
|---|---|
| P500 | |
| P600 | |
| P505 | |

Dirty Page Table

| transID | lastLSN |
|---|---|
| T1000 | |
| T2000 | |

Transaction Table

| prevLSN | transID | type | pageID | Length | offset | Before Image | After Image |
|---|---|---|---|---|---|---|---|
| | T1000 | Update | P500 | 3 | 21 | ABC | DEF |
| | T2000 | Update | P600 | 3 | 41 | HIJ | KLM |
| | T2000 | Update | P500 | 3 | 20 | GDE | QRS |
| | T1000 | Update | P505 | 3 | 21 | TUV | WXY |

- T1000 changes the bytes 21-23 from ABC to DEF
- T2000 changes HIJ to KLM on page P600
- T2000 changes from GDE t QRS on page 500
- T1000 changes TUV to WXY on P505
- At this instant, the log, dirty page table and transaction table is shown

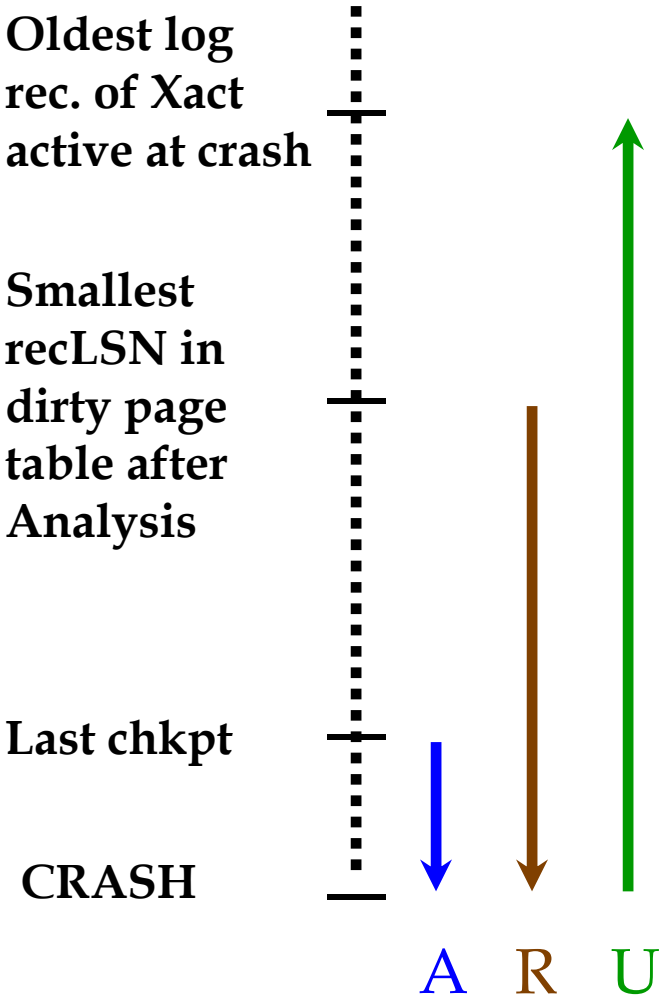
# The Write-ahead log Protocol

- Before writing a page to disk, every update log record that describes a change to this page must be forced to stable storage.
  - Force all log records up to and including the one with LSN equal to the pageLSN to stable storage before writing a page to disk.
- **Committed transaction:**
  - **A transaction all of whose log records, including a commit record, have been written to stable storage.**
- When a transaction is committed, the log tail is forced to stable storage.
- Difference with the force approach:
  - **Under force approach, all the pages modified by the transaction are forced.**
- In ARIES we follow no force approach and write only log tail.
- It can be noted that the cost of forcing the log tail is much smaller than the cost of writing all changed pages to disk.

# Checkpointing

- Conceptually, keep log around for all time. Obviously this has performance/implementation 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:
- A checkpoint is a snapshot of the system
- A checkpoint in ARIES has three steps.
  - 1. begin_checkpoint record: Indicates when chkpt began.
  - 2. end_checkpoint record with current status of current *Xact table* and *dirty page table*.
  - 3. Master record: A special master record containing theLSN of the begin-checkpoint log record is written to a known place on stable storage.
  - 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.

# 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.

# 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,
    - If the log record is a commit record, the status of transaction is set to C, other it is set to U (to be undone)
  - 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

# 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.**
- **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
    - It means all changes have been written to disk
  - Affected page is in D.P.T., but has recLSN > LSN, or
    - It means that the update has propagated to disk.
  - 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!

# Phase 3: The UNDO Phase

- **A Naïve solution:**
  - The xacts in the Xact Table are losers.
  - For each loser, perform simple transaction abort.

# Phase 3: The UNDO Phase

**ToUndo={lastLSNs of all Xacts in the Xact Table}**

Repeat:

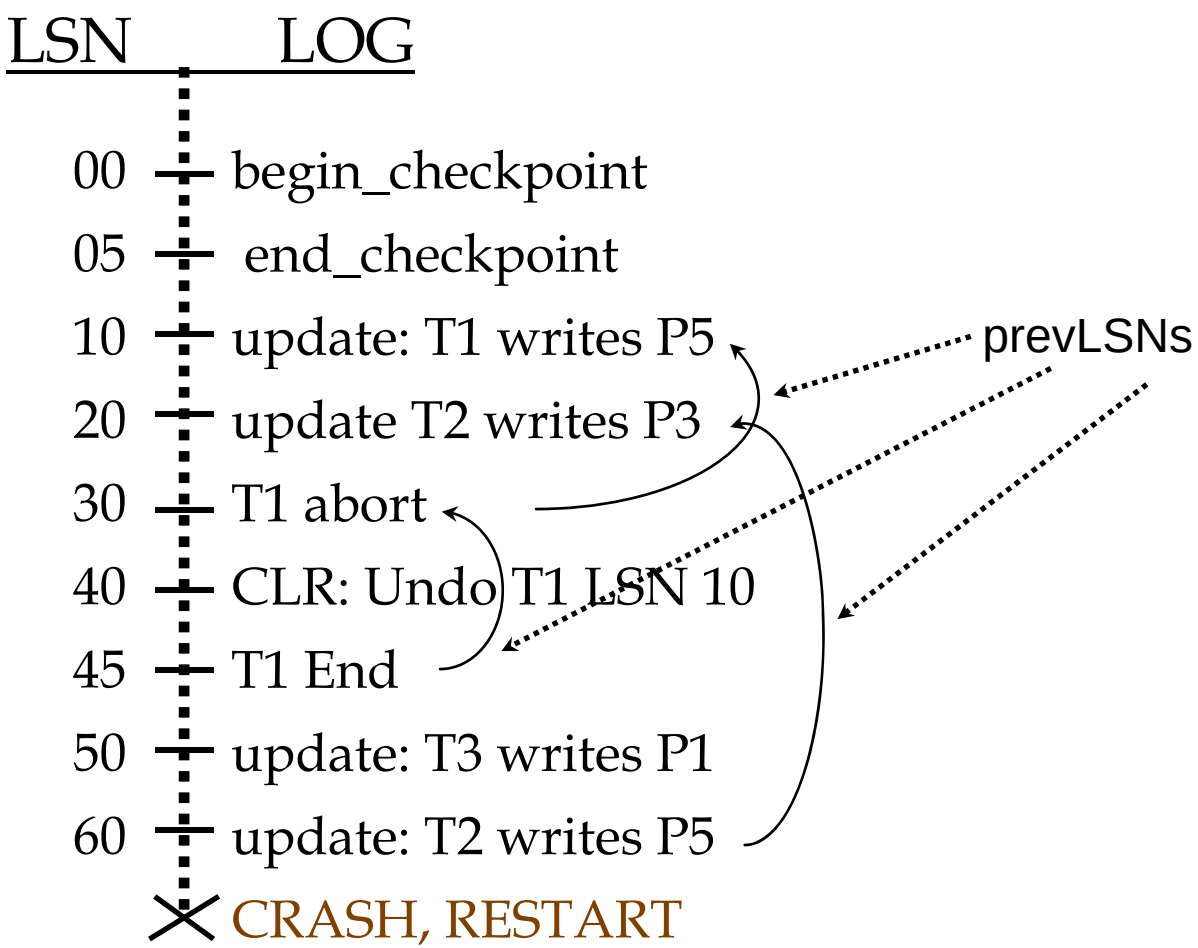
- Choose (and remove) largest LSN among ToUndo.
- If this LSN is a CLR and undonextLSN==NULL
  - Write an End record for this Xact.
- If this LSN is a CLR, and undonextLSN != NULL
  - Add undonextLSN to ToUndo
- Else this LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo.

Until ToUndo is empty.

# Example of Recovery


Xact Table
lastLSN
status
Dirty Page Table
recLSN
flushedLSN

ToUndo



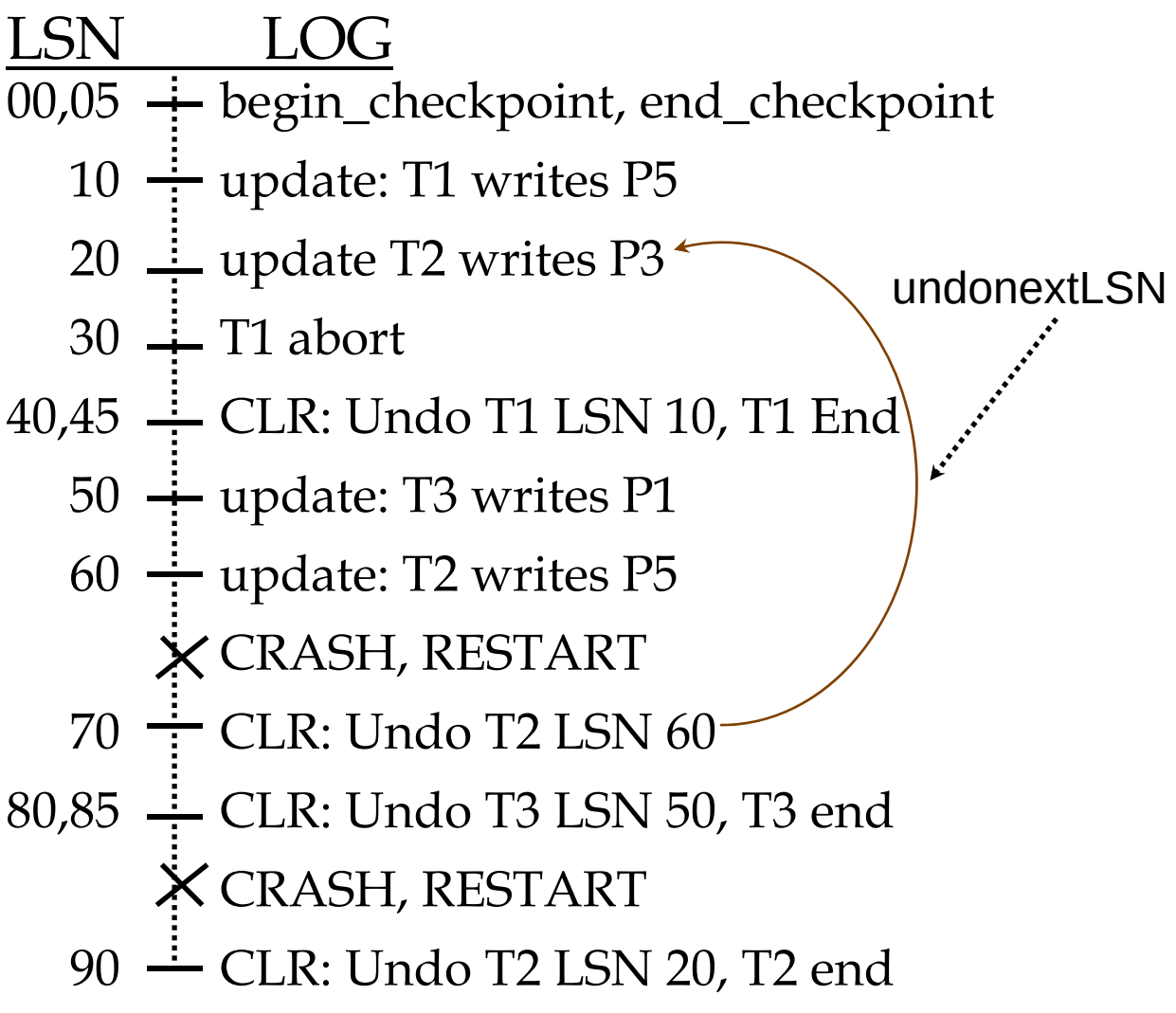
# Example

- **After the first crash**
  - Analysis identifies P1, P3, P5 are dirty pages
  - Log record 45 shows T1 is a completed transaction. So transaction table identifies T2 and T3 are active at the time of crash.
  - The redo phase starts at log record 10 which is the minimum LSN in the dirty page table and reapplies all actions for updates and CLRs.
  - ToUndoSet consists of LSNs 60 for T2 and 50 for T3.
  - The update is Undone so CLR is written for T3 with end record.
  - For T2 there are two CLR record to be written. After writing one CLR record, the system crashes again.
  - While second restart, the page LSNs are used to detect the situation and avoid writing the pages again.

# Example: Crash During Restart!


Xact Table
lastLSN
status
Dirty Page Table
recLSN
flushedLSN

ToUndo



# Additional Crash Issues

- What happens if system crashes during Analysis?
  - All the work is lost. Restart the analysis phase.
- What happens if the system crashes during REDO?
  - Some of the changes during redo may have written to disk prior to the crash. Restart starts with the Analysis phase and then redo phase.
  - Some update log records whoch have done will not be redone because the pageLSN is now equal to the update record's LSN.
- What happens if the system crashes during UNDO ?
  - Same as in case of REDO.

# Media Recovery

- Periodically making a copy of the database
- It is similar to taking a fuzzy checkpoint.
  - DBMS operations should be continued.
- When a database object or a file is currupted, a copy of that object is brought to up-to-date by using a log to indentify and apply the changes of committed transactions and undo the changes of uncommitted transactions.

# Other Approaches

- ARIES has several advantages
- System R has employed shadow paging approach.
  - No logging
- Database is a collection of pages
- When a transaction modifies a page it modifies to a shadow page. Page table is modified accordingly.
- Aborting a transaction
  - Discard shadow page table and shadow pages.
- Committing a transaction
  - Make a shadow page public and discard original data pages.
- Such a scheme suffers from several problems
  - Data becomes fragmented, storage overhead,..

# Summary of Logging/Recovery

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

# Summary, Cont.

- Checkpointing: A quick way to limit the amount of log to scan on recovery.
- Recovery works in 3 phases:
  - Analysis: Forward from checkpoint.
  - Redo: Forward from oldest recLSN.
  - Undo: Backward from end to first LSN of oldest Xact alive at crash.
- Upon Undo, write CLRs.
- Redo “repeats history”: Simplifies the logic!