CSE 541: Database Systems I

Logging & Recovery

Slides Credit: Tianzheng Wang

ACID Properties

Ensured despite concurrent accesses and system failures

- Atomicity
  - Actions in a transaction are either all applied, or not at all
  - Commit – apply all read/write actions
  - Abort – rollback all changes so far, as if nothing happened
- **Consistency**
  - Must leave the database in a consistent state, no anomalies etc.
- **Isolation**
  - As if the transaction were the only one in the system
- Durability
  - Successful changes must be correctly persisted in storage

Concurrency Control (CC) mainly address C & I.

ACID Properties

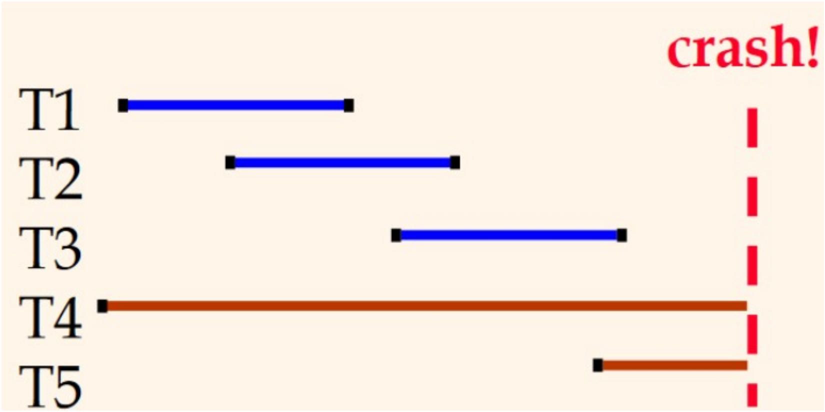
Ensured despite concurrent accesses and system failures

- Atomicity
  - Actions in a transaction are either all applied, or not at all
  - Commit – apply all read/write actions
  - Abort – rollback all changes so far, as if nothing happened
- Consistency
  - Must leave the database in a consistent state, no anomalies etc.
- Isolation
  - As if the transaction were the only one in the system
- Durability
  - Successful changes must be correctly persisted in storage

Need a scheme to guarantee A & D

Some Transactions may not Commit

- User-aborted transactions
- Power failure, crashes, bug in DBMS
- Data may get lost or corrupted if not handled correctly



Desired upon crash/recovery:

- T1, T2, T3's changes should be preserved in the database
- T4 and T5's changes shouldn't

Committed transactions must survive failures and crashes

Recoverability & Cascading Aborts

Recoverable schedule: a transaction (T1) who read changes by another transaction (T2) should not commit before T2 does

- I.e., Should wait for all depending transactions to commit first

- Avoids cascading aborts
  - If T2 aborts, T1 would have to abort

- Have seen it in SS2PL

Guaranteeing Durability

Ensure data reaches storage from the buffer pool

Key question: When should we write to storage?

One alternative is “Force”

- Write every change immediately to storage (write-through)
  - No need to “redo” changes
- High response time, slow
- Not desirable

Desired: “No Force”

- Only write to storage when needed
  - What if a crash happens before we wrote back all changes?

Guaranteeing Atomicity

Ensure either all or no changes are persisted

One alternative is "No Steal"

- Keep all modifications in buffer pool until commit
  - No need to "undo" changes in case the transaction aborts

Observation: No guarantee that a transaction's whole footprint will always stay in the buffer pool

- T2 wants to load a page to the buffer pool, and a frame that contains a page of T1 is chosen

Desired: "Steal" – allow early write-back before commit

- Both T1 and T2 can proceed → Increased throughput
- But what if T1 decides to abort or there is a crash?
  - Violating atomicity: part of T1's changes are persisted

Steal and No Force

- Allow to write back dirty pages before commit
  - Need to “roll back” (undo) changes on storage if the transaction aborts
- Buffered pages may not be written back upon commit
  - Write back pages later when needed
  - Simply “roll forward” (redo) the changes if there is a crash before pages are written back

	No Steal	Steal
Force	Poor performance	
No Force		**Desired**, but risks atomicity and durability

Solution: Logging

The Log: History of actions executed by the DBMS

- Remembers each action, every change to the database
  - i.e., Redo and Undo information
  - As well as control information (e.g., commit/abort actions)
- Sequentially written to storage
  - Use high-performance, parallel disks/SSDs/persistent memory for better performance
- Upon recovery, examine log records to redo and undo changes
- Consists of log records which describe changes made to the database

Log Records

A log record describes an action performed on the data

- Each uniquely identified by a log sequence number (LSN)
  - No two log records will have the same LSN
  - LSN monotonically increases
  - **Smaller LSN == change happened earlier**
  - Often use the cumulative byte count of log records or an offset into a log file
    - Can fetch log record directly by its LSN
- Only remember the difference, instead of the whole page
  - Minimize storage space needed, improved performance
  - E.g., Update an 8-byte record in a 4KB page, only the 8-byte change (+ its old value) is recorded, not a whole page

Major Log Record Types

- Page update – denotes an update to a page
  - Generated after modifying a pinned page in the buffer pool
- Commit
  - Written when a transaction decides to commit
  - Contains the committing transaction's ID
  - A transaction is considered "committed" once all log records preceding its commit log record are persisted (inclusive)
- Abort – denotes the decision to abort a transaction
  - Undo is then started
- End – completion of other operations following the commit/abort decision
  - E.g., clean up transaction metadata

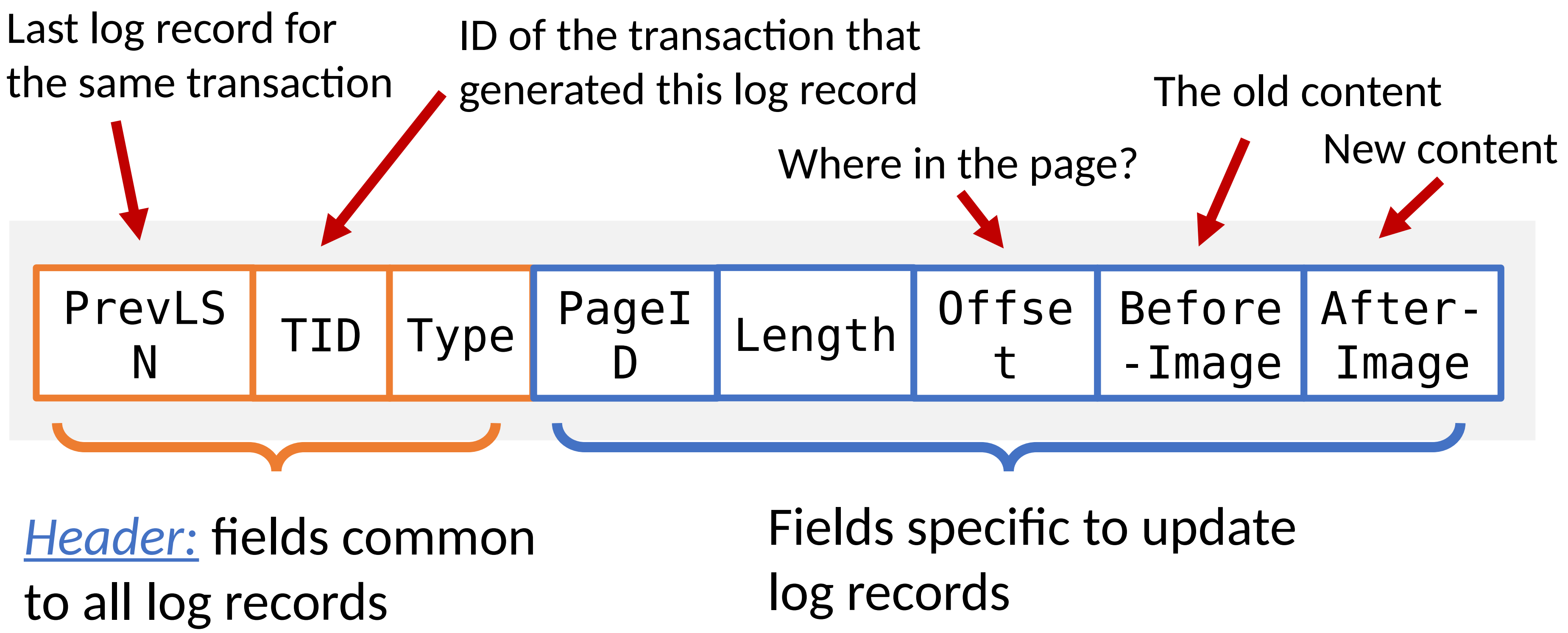
Major Log Record Types

- Compensation Log Record (CLR)
  - Written before an update record is undone (rolled back)
    - During forward processing (abort) or during crash recovery
  - CLRs are never undone
  - CLRs are redone during repeated crash, just like “normal” redo log records
  - Points to the next record to undo using undoNextLSN
    - undoNextLSN == prevLSN in update record
- Other types (system implementation dependent)
  - E.g., skip records to denote the end of a log file

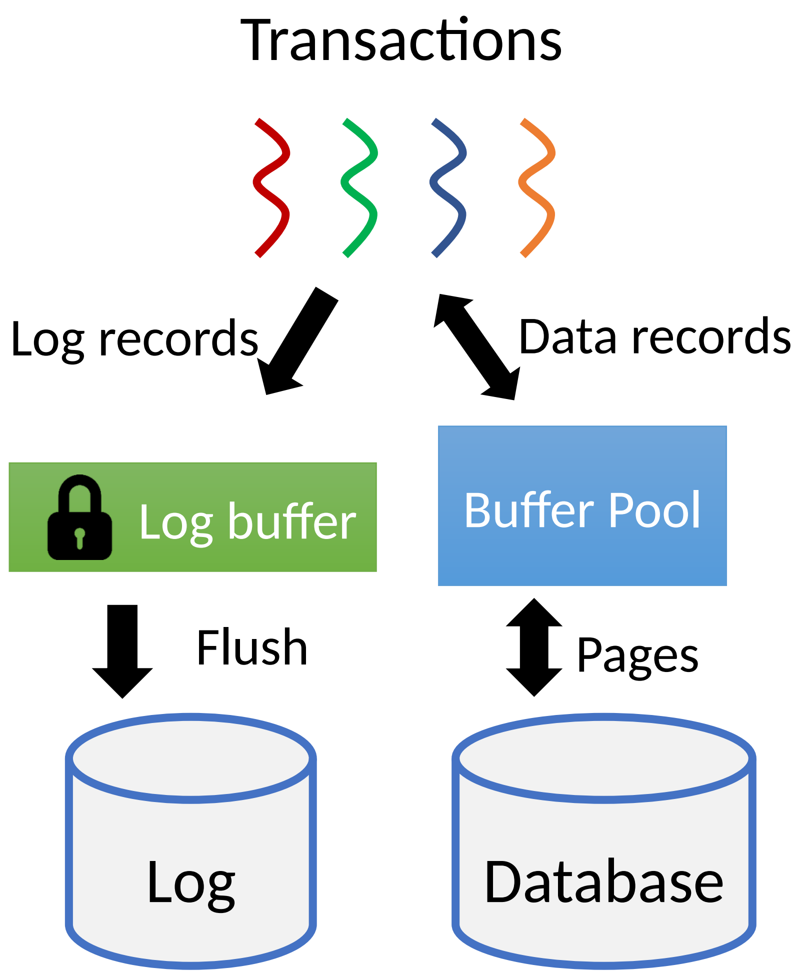
Real systems may not use all of these record types, depending on the specific logging protocol

Log Record Structure

- **Example: Update Log Record**



Overall Architecture



<u>Log buffer:</u> Accumulate log records before they reach storage

- Aka ‘log tail’
- Typically small (e.g., 8MB, 32MB)
- Typically a single buffer shared among all transactions
  - Protected by a latch
- Periodically flushed to storage in batches to leverage fast sequential storage write
  - Flush when: **(1)** timeout, **(2)** buffer full, **(3)** transaction commit – **write-ahead logging** protocol

Write-Ahead Logging (WAL)

Any change to the DB is first durably recorded in the log

- Log records must be persisted in storage <u>before</u> transaction commits ("flush-before-commit")
- Log records must be persisted in storage <u>before</u> dirty pages are written back to storage

→ Transaction considered "committed" if all of its **log records** have been persisted in storage

Clients and other transactions can see result

ARIES

Recovery algorithm designed for Steal, No Force

Assumptions:

- Log records stored in stable storage (e.g., disks)
- Concurrency control (e.g., 2PL) is in effect
- Steal, No-Force buffer management (desirable option)
- Crash may happen during recovery (“repeated crashes”)
- Atomic read/write operations

Most modern systems implement some variant of ARIES

Recovery-Related Data Structures

Active Transactions Table

Transaction ID	Status	Last LSN
T1	In progress	10
T2	Committed	40
T3	In progress	60

LastLSN: LSN of the transaction's most recent log record

Dirty Pages Table: changes may not yet be reflected on storage

Page ID	Rec LSN
P1	50
P3	20
P5	10

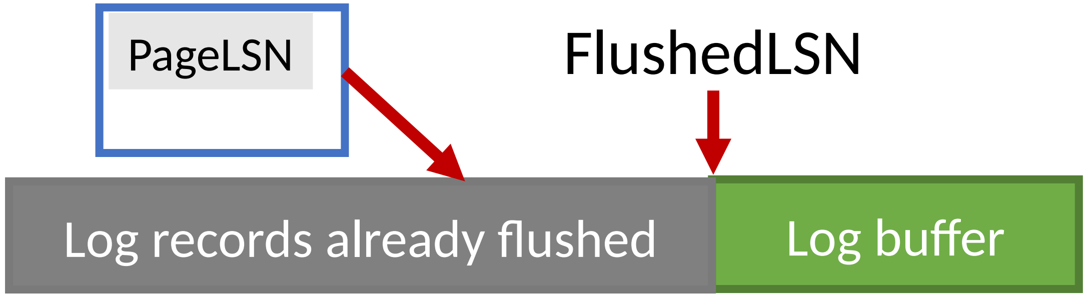
RecLSN: LSN of the first log record that caused the page to become dirty.

- Might be the earliest log record that has to be redone during recovery

Both tables maintained during normal operation, reconstructed in the Analysis phase during recovery

Additional LSNs

- Each data page also contains a PageLSN
  - LSN of the log record for an update to the page
  - Accelerates the Redo Phase
    - No need to redo log records with LSN < PageLSN
    - Every record is at most redone once
- System-wide FlushedLSN (aka Durable LSN)
  - Max LSN of flushed log records
  - Before a page is written to storage, PageLSN FlushedLSN



Checkpointing

Save a snapshot of the database (i.e., flush the buffer pool)

- Can reduce the amount of work done during recovery
  - Truncate at the oldest recLSN
- Fuzzy or hard (stop-the-world)
  - Fuzzy checkpoint – no need to stop forward processing
- Step 1 – write a begin_checkpoint log record
  - Denotes the beginning of a checkpoint operation
- Step 2 – write an end_checkpoint log record
  - Includes Dirty Pages Table and Active Transactions Table

  ➔ Accurate as of the time of begin_checkpoint
- Step 3 – Force log + record LSN of begin_checkpoint in a master record
  - Often stored as part of the file systems metadata (e.g., file name)

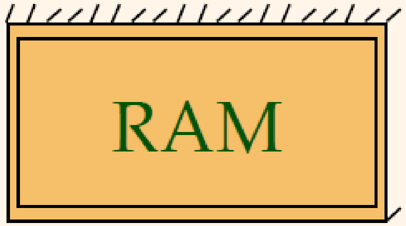
Recovery starts at the most recent begin_checkpoint record

ARIES Big Picture

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

ARIES Recovery Phases

Invoked after a crash upon DBMS restart

Three Phases:

1. **Analysis** – Identify these at the time of the crash:
   - Dirty pages in the buffer pool
   - Active transactions
2. **Redo** – Repeat all actions recorded by the log
   - DB state will be restored to what it was at the time of crash
3. **Undo** – Undo actions of transactions that didn’t commit
   - DB state reflects only actions of committed transactions
   - Changes made during undo are logged, so that they are not repeated in case of repeated crashes

Example

LSN	Log
10	Update: T1 writes P5
20	Update: T2 writes P3
30	T2 commit
40	T2 end
50	Update: T3 writes P1
60	Update: T3 writes P3

Crash and restart

T1 and T3:

- Didn't commit
- Need to be undone
- "Loser transactions"

T2: Committed

Analysis Phase

- Start with the previous checkpoint's active transactions and dirty pages tables (if a checkpoint exists)
- Scan the log to collect new entries and update existing entries in active transactions and dirty pages tables

LSN	Log
10	Update: T1 writes P5
20	Update: T2 writes P3
30	T2 commit
40	T2 end
50	Update: T3 writes P1
60	Update: T3 writes P3

Crash and restart

Active transactions at crash time:

- T1
- T3

Dirty pages at crash time:

- P1
- P3
- P5

Redo Phase

“Roll forward” and repeat history up to crash time

- Including updates that need to be undone later
- Start from the smallest recLSN in Dirty Pages Table
  - Redo every update and CLR record, unless
    - Target page not in Dirty Pages Table, or
    - Target page has recLSN > log record LSN, or
    - Target page’s pageLSN >= log record LSN

Oldest update that may not have been applied

- Redo
  - Load affected page to buffer pool
  - Apply after-image
  - Usually single-threaded – no locking needed
  - Set pageLSN to log record LSN
  - No additional logging needed

Undo Phase

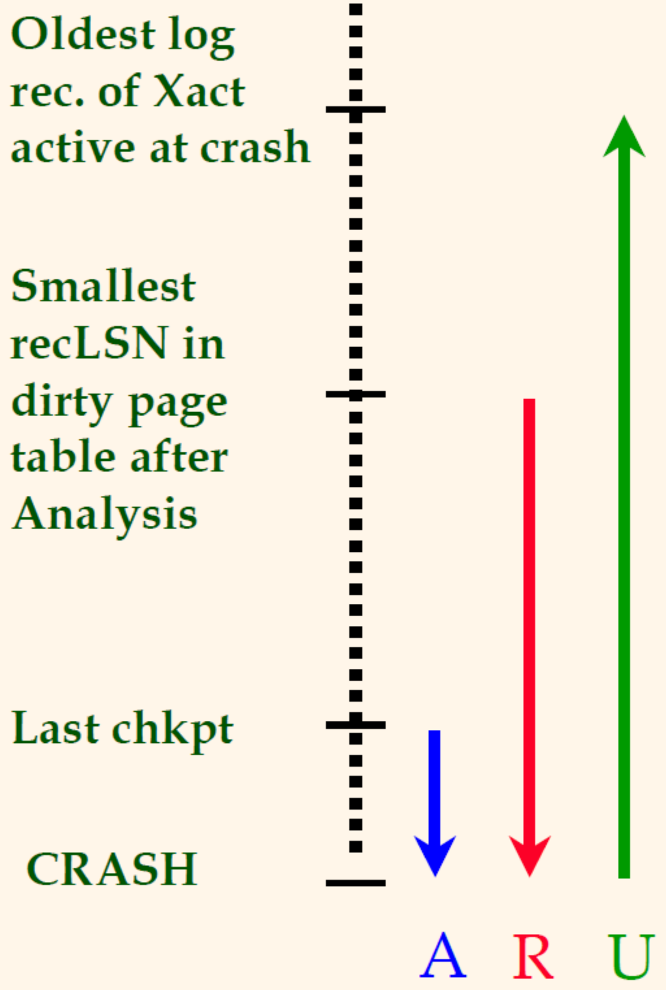
“Roll back” actions done by loser transactions

- Load affected page to buffer pool
- Apply before-image
- Actions must be undone in reverse of the order they appear in the log (scan log backwards)
- Set ToUndo = { lastLSNs of all active transactions }

Repeat until ToUndo is empty:

- Choose the largest lastLSN record in ToUndo
  - If it is a CLR
    - Has valid undoNextLSN → add undoNextLSN to ToUndo
    - No undoNextLSN and end record written → discard the CLR
  - If it is an update: undo and write a CLR, add prevLSN to ToUndo

Summary: ARIES Recovery Phases



- 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.