

CS262: Advanced Topics in Computer Systems

Fall 2026

Lecture 4: Principles of Transaction Recovery

Rishabh Iyer

This Paper in a Nutshell

Recovery is a consequence of normal-system design choices.

- What system allows during execution determines what can be left behind after a crash.

Performance and recovery complexity are traded against each other.

- More freedom during normal execution usually means more work at recovery time.

The key question is “what guarantees must recovery provide given these design choices?”

- Good design minimizes common-case overhead while keeping recovery correct.

Paper’s main contribution: a framework for reasoning about this tradeoff systematically.

What Must Recovery Accomplish?

From ACID:

- Atomicity: incomplete transactions must leave no effects.
- Durability: committed transactions must leave all effects.

After a crash, recovery must therefore ensure:

- UNDO effects of incomplete transactions.
- REDO effects of committed transactions that are missing.

What Survives A Crash?

During normal execution:

- Some DB state is in memory, some DB state has reached persistent storage.
- If a log exists, some log records are in memory, some log records are durable.

After a crash:

- Volatile state is lost.
- Recovery starts from:
 - the persistent database
 - the durable log

Complication: “what is on disk” does not necessarily equal “the latest database state.”

Three Views of a Database

- **Current database**
 - What the running DBMS currently sees.
 - Includes newer page contents still only in the buffer.
- **Materialized database**
 - What the DBMS sees immediately after restart, before applying the log.
 - This is the state recovery starts from.
- **Physical database**
 - Everything physically present on storage.
 - May include old or unreachable page versions.
- Recovery must transform materialized DB into the most recent txn-consistent state.

Modify vs Write vs Propagate

The paper separates three events that are easy to conflate:

- Modify
 - Change a page in memory. Changes the **current** DB.
- Write
 - Put a page image onto persistent storage. Changes the **physical** DB.
- Propagate
 - Make that page version part of the materialized DB, i.e., the version the DBMS sees after restart.

Write == Propagate for in-place/direct updates. They are distinct for indirect/shadow page-style updates.

The Three Key Design Choices

During normal execution, the DBMS makes three largely independent choices:

- **Propagation:** Atomic or non-atomic.
 - Do page updates become restart-visible together, or independently?
- **Buffer mgmt:** Steal vs no steal
 - Can pages modified by an incomplete transaction leave the buffer?
- **Commit:** Force vs no force
 - Must all modified pages be propagated before commit completes?

These choices determine what the materialized database can look like after a crash and thus determine what recovery needs to do.

Implications for Recovery

ATOMIC vs. ~ATOMIC

- ATOMIC → restart sees a well-defined previously propagated state.
- ~ATOMIC → restart may see a mixture of old and new pages.

STEAL

- Incomplete transaction state may already be present.
- → UNDO may be required.

~FORCE

- Committed transaction state may still be missing.
- → REDO may be required.

Which Choices Do DBMS Typically Make?

The performance-friendly choices are often:

- ~ATOMIC → simple update-in-place storage.
- STEAL → buffer manager can evict dirty pages freely.
- ~FORCE → commit need not flush scattered data pages.

So:

- ~ATOMIC + STEAL + ~FORCE gives a cheap, flexible normal path but complicates recovery. Yet most systems choose this because recovery is rare and we want to make the common case fast.

Write-Ahead Logging (WAL)

Make the log durable **before** the data-page state can be required for recovery.

So, suppose T1 changes: X: 10 $\rightarrow$ 20, we log: <T1, X, old=10, new=20>

This log now supports:

- UNDO $\rightarrow$ restore 10
- REDO $\rightarrow$ restore 20

So WAL works, but what are its limitations? Hint: what happens if failures are **very** rare?

Checkpointing

If failures are rare, the system may run for a long time between crashes.

With WAL alone:

- The log can grow very large.
- Recovery may need to scan/replay a long history.

Checkpointing:

- Periodically establish a point from which recovery can start.
- Bound the amount of log that must be processed after a crash.

Tradeoff: more work during normal execution → less work during recovery.

Checkpointing Flavours

Clean / synchronous checkpoint

- Pause or constrain transaction activity.
- Propagate dirty state to establish a clean recovery point.
- More work at checkpoint time, simpler recovery.

Fuzzy checkpoint

- Transactions continue running.
- Do not force all dirty pages.
- Record recovery state in the log instead.
- Cheap checkpoint, more sophisticated recovery.

Shift work between normal execution and recovery while keeping recovery bounded.

Design Exercise

We're going to try and implement recovery for the fast path assuming the system is:

- Not atomic, allows steal, no force, and uses fuzzy checkpoints.

Let's focus on 3 key questions:

- How do we know whether a logged update is already on disk?
- How can we bound log growth and recovery-time traversal of logs?
- Which updates should we REDO?

Another question (we won't discuss today): What if recovery itself crashes?

Is An Update on Disk?

Problem:

- The log contains an update to page P. P may or may not already contain that update.

Solution: Log sequence numbers (LSNs)

- Give every log record a monotonically increasing LSN.
- Store on each page its pageLSN: latest update reflected in that page.

During REDO:

- $\text{pageLSN} \geq \text{record LSN} \rightarrow$ already applied, skip.
- Otherwise $\rightarrow$ REDO.

Bounding Log Growth & Recovery Traversal

At a fuzzy checkpoint, record two data structures:

- Dirty Page Table
 - Which pages may not yet be on disk.
 - recLSN: earliest log record that may need REDO.
- Transaction Table
 - Which transactions are still active.
 - Where each transaction's log chain currently ends (lastLSN).
 - Tells recovery what may still need UNDO.
- Together they identify the oldest log history recovery might still need.

Which Updates To Redo?

Naive idea: REDO only committed transactions.

Counter example:

- T1 inserts into a B-tree leaf. This causes a page split. T1 is still uncommitted.
- T2 updates the new leaf and commits.
- Crash.

If we REDO only T2: T2's target page may not exist yet. Instead:

- REDO history
- Reconstruct the physical DB at crash
- UNDO losers

Key idea: reconstruct first, clean up second.

Summary: Txns, Concurrency & Recovery

Transactions provide the ACID abstraction

- Hide concurrency and failures from applications.

Concurrency control trades performance for isolation

- Serializability defines the strongest isolation guarantee.
- 2PL enforces it; lock granularity and weaker isolation levels recover concurrency.

Recovery follows from normal-execution choices

- ~ATOMIC + STEAL + ~FORCE makes the common path fast.
- The cost is UNDO + REDO + logging.

Good recovery pushes work off the common path