

CS262: Advanced Topics in Computer Systems

Fall 2026

Lecture 4: Transactions

Rishabh Iyer

(adapted from slides by Stoica and Ghodsi)

Today's Papers

- **Granularity of Locks and Degrees of Consistency in a Shared Database**
 - J.N. Gray, R.A. Lorie, G.R. Putzolu, I.L. Traiger.
 - Modelling in Data Base Management Systems, 1976.
- **Principles of Transaction-Oriented Database Recovery**
 - Theo Haerder and Andreas Reuter
 - ACM Computing Surveys, 1983.

What is a Transaction?

- A transaction is an application-defined sequence of operations that should appear as one logical state transition.
- Canonical example: Transfer \$100 from Alice to Bob.
- Today's lecture: How can we build systems that provide this abstraction **correctly** yet **efficiently**?

ACID Properties of Transactions

- **Atomicity:** all actions in the transaction happen, or none happen.
- **Consistency:** if each transaction is consistent, and the database starts consistent, it ends up consistent.
 - Example: Balance cannot be negative, cannot schedule meeting on Feb 30.
 - Primarily about **application semantics/invariants**, not transaction machinery.
- **Isolation:** execution of one transaction is isolated from others.
- **Durability:** if a transaction commits, its effects persist.

Why Are Transactions Hard?

- **Concurrency**

- Multiple transactions execute at the same time.
- Main challenge: **Isolation**
- Solution: **Concurrency control**

- **Failures**

- A transaction or the system can fail midway through execution.
- Main challenges: **Atomicity + Durability**
- Solution: **Recovery**

- **Today:**

- Gray et al. → concurrency control
- Haerder & Reuter → recovery

Why Not Execute Transactions Serially?

- Serial execution is easy to reason about and trivially provides isolation.
- But it sacrifices concurrency:
 - Transactions accessing unrelated data cannot execute simultaneously.
 - One transaction may wait for I/O while useful work is available.
- **Goal:** maximize concurrency while preserving isolation.

Defining Serializability

- **Serializability** is the strongest standard isolation level.
- A **schedule** is an interleaving of *operations* from different transactions.
- A schedule is **serializable** if its effect is equivalent to some serial execution of those transactions.
 - Intuition: concurrent transactions should behave as though they executed one at a time, in some order.
- Weaker isolation levels exist; we will return to them in the Gray et al. presentation.

Conflict Serializability (I)

- Semantic equivalence is difficult to reason about efficiently.
- Instead, database systems commonly use a stronger, syntactic notion.
- Two operations conflict if:
 - They belong to different transactions.
 - They access the same data item.
 - At least one is a write.
- Therefore:
 - $(R,R) \rightarrow$ no conflict. (R,W) (W,R) $(W,W) \rightarrow$ conflict
- **Intuition:** non-conflicting operations commute; conflicting operations impose an ordering.

Conflict Serializability (II)

- Two schedules are conflict equivalent if:
 - They contain the same operations.
 - Every pair of conflicting operations appears in the same relative order.
- A schedule is conflict serializable if it is conflict equivalent to some serial schedule.

Checking Conflict Serializability

Build a dependency graph:

- One node per transaction.
- Add an edge $T_i \rightarrow T_j$ if:
 - an operation of T_i conflicts with an operation of T_j , and
 - T_i 's operation appears first in the schedule.

Result:

- A schedule is conflict serializable iff the dependency graph is acyclic.
- If there is a cycle, no serial ordering can satisfy all conflict orderings.
- If the graph is acyclic, a topological ordering gives an equivalent serial order.

From Serializability to Locking

- Conflict serializability tells us which schedules are correct.
 - But the DBMS needs to enforce this **online**, as transactions execute.
- Natural mechanism: locks.
 - Conflicting accesses wait.
- But **locking each access is not enough**. For example:
 - On A: T1 goes before T2 $\rightarrow T1 \rightarrow T2$
 - Later on B: T2 goes before T1 $\rightarrow T2 \rightarrow T1$
 - Now we have a cycle.
- Need: **one consistent ordering across all conflicts**.

Two-Phase Locking (2PL)

Broken into two phases:

- Growing phase: acquire locks, release none.
- Shrinking phase: release locks, acquire no new ones.

Why does this work?

- Once a transaction releases a lock and lets another transaction pass, it can no longer acquire new locks and “get back in front.”
- Therefore all conflicts are consistent with a single ordering of transactions.

Thus, 2PL guarantees conflict serializability.

A Small Nuance: Strict 2PL

Basic 2PL still allows locks to be released before commit.

- This can expose tentative state and complicate rollback.

Strict 2PL:

- Hold locks until commit/abort.
- Prevents other transactions from depending on uncommitted state.
- Simplifies recovery.

So We Need Locks ...

But what exactly do we lock?

- Database?
- Table?
- Page?
- Record?

Tradeoff:

- Coarse locks → low overhead, low concurrency.
- Fine locks → high concurrency, high overhead.

→ Gray et al.: Granularity of Locks and Degrees of Consistency