Lecture 6: Transactions, Concurrency & Recovery

What is a Transaction?

A transaction is a single unit of work that consists of one or more SQL statements. It must be completed fully or not at all to maintain data integrity.

ACID Properties of Transactions

ACID stands for:

Property	Description	Example
A - Atomicity	All operations in a transaction are performed or none are.	Transfer money from A to B. Both debit and credit must happen, or none.
C - Consistency	The transaction must leave the database in a valid state.	After transferring money, the total balance remains the same.
I - Isolation	Transactions occur independently without interference.	One transaction reading balance should not see the intermediate results of another.
D - Durability	Once a transaction is committed, it persists even after a system failure.	If power fails after transfer completes, data must remain updated.

1. Atomicity Example:

```
BEGIN;

UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;

COMMIT;
```

If one update fails, the other **must not** proceed. Rollback ensures atomicity.

2. Consistency Example:

A bank transaction should not leave a negative balance if a rule prohibits it.

```
-- Withdraw money
UPDATE accounts SET balance = balance - 500 WHERE account_id = 1;

-- Check rule
-- If balance < 0, rollback.
```

3. Isolation Example:

If two users withdraw money at the same time, one must **wait or execute serially** to prevent inconsistencies.

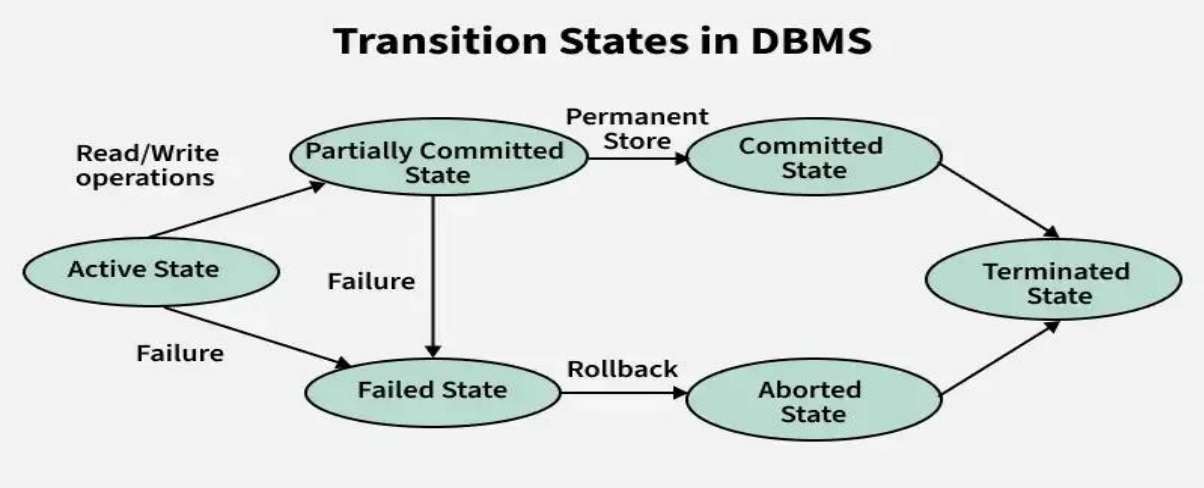
4. Durability Example:

Once COMMIT is done, the data is stored on disk. Even if the server crashes, changes remain.

Transaction States

A transaction goes through several states during its lifecycle:

[Active] → [Partially Committed] → [Committed] → [Terminated]
↓
[Failed] → [Aborted]



1. Active
- The transaction is executing its operations.

Example: Reading balance from Account A.

2. Partially Committed
- The last statement has been executed, but not yet committed.

Example: Balance debited but not yet saved to disk.

3. Committed
- All changes are **permanently saved** to the database.

Example: Money successfully transferred and logged.

4. Failed
- Something went wrong (system crash, constraint error).

Example: Insufficient balance or power outage before COMMIT.

5. Aborted
- The transaction is rolled back. The system undoes all changes.

Example: If an error occurs, reverse all debits and credits.

Rollback Example:

```
BEGIN;

UPDATE accounts SET balance = balance - 500 WHERE account_id = 1;

-- Error: balance negative
ROLLBACK;
```

Summary

Property/State	Key Idea	Use Case / Outcome
Atomicity	All-or-nothing	Debit + Credit both succeed or roll back
Consistency	Valid state transitions	Enforce business rules
Isolation	Prevent dirty reads or lost updates	Concurrent withdrawals
Durability	Data remains after commit	Even if a crash happens after COMMIT
Aborted	The transaction rolled back	On error, restore previous state

Concurrency Issues in Transactions

When multiple transactions execute at the same time without proper isolation, **data anomalies** may occur. These problems happen when transactions interfere with each other's reads or writes.

1. Lost Update

Definition:

Two transactions **read the same data and update it**, but the final value reflects only one update — the other is **lost**.

Example:

```
-- Transaction 1
BEGIN;
SELECT balance FROM accounts WHERE id = 1;  -- Returns 1000
-- Waits...


-- Transaction 2
BEGIN;
SELECT balance FROM accounts WHERE id = 1;  -- Returns 1000
UPDATE accounts SET balance = 900 WHERE id = 1;
COMMIT;

-- Back to Transaction 1
UPDATE accounts SET balance = 800 WHERE id = 1;
COMMIT;  -- Overwrites 900 with 800 (Lost Update)
```

Prevention: Use proper isolation levels or locks.

2. Dirty Read

Definition:

A transaction reads data **modified by another uncommitted transaction**. If the other transaction is rolled back, the data read was invalid or "dirty".

Example:

```
-- Transaction 1
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;

-- Transaction 2
BEGIN;
SELECT balance FROM accounts WHERE id = 1;  -- Reads decreased balance (Dirty Read)
COMMIT;

-- Back to Transaction 1
ROLLBACK;  -- Now T2 has read incorrect data
```

Prevention: Use READ COMMITTED or higher isolation level.

3. Non-Repeatable Read

Definition:

A transaction **reads the same row twice** and gets different data because another transaction modified it in the meantime.

Example:

```
-- Transaction 1
BEGIN;
SELECT salary FROM employees WHERE id = 101;  -- Returns 5000

-- Transaction 2
BEGIN;
UPDATE employees SET salary = 6000 WHERE id = 101;
COMMIT;

-- Back to Transaction 1
SELECT salary FROM employees WHERE id = 101;  -- Returns 6000 (changed)
```

Prevention: Use REPEATABLE READ or SERIALIZABLE isolation level.

4. Phantom Read

Definition:

A transaction re-executes a query and **sees new rows** that were inserted by another transaction.

Example:

```
-- Transaction 1
BEGIN;
SELECT * FROM orders WHERE amount > 1000;  -- Returns 3 rows

-- Transaction 2
BEGIN;
INSERT INTO orders VALUES (105, 'New Order', 1500);
COMMIT;

-- Back to Transaction 1
SELECT * FROM orders WHERE amount > 1000;  -- Returns 4 rows (Phantom row)
```

Prevention: Use SERIALIZABLE isolation level.

Summary Table

Issue	Description	Caused By	Prevent Using
Lost Update	Final update overwrites the previous one	Concurrent writes	Pessimistic locking / higher isolation
Dirty Read	Read uncommitted data from another transaction	Read before COMMIT	READ COMMITTED
Non-Repeatable	Same row read twice returns different results	Update by another transaction	REPEATABLE READ
Phantom Read	Re-query returns extra rows	Insert/Delete during transaction	SERIALIZABLE

Isolation Levels (for Reference)

Isolation Level	Prevents
READ UNCOMMITTED	Nothing
READ COMMITTED	Dirty Reads
REPEATABLE READ	Dirty, non-repeatable
SERIALIZABLE	All (full isolation)

Serializability & Locking Techniques (2PL, Shared & Exclusive Locks)

Serializability in DBMS

Serializability ensures that the **concurrent execution of transactions** results in a **database state that would be obtained if the transactions were executed serially** (i.e., one after the other, without overlapping).
It is a **measure of correctness** for a concurrent schedule.

Why **Serializability**?

When transactions run **concurrently**, they may interfere with each other and cause issues like:

- **Lost Updates**
- **Dirty Reads**
- **Inconsistent Results**

Serializability prevents such anomalies by ensuring the result is **equivalent to a serial schedule**.

Types of Serializability

Type	Description
Conflict Serializability	Based on non-conflicting operations (read/write order)
View Serializability	Based on transaction outputs being equivalent

1. Conflict Serializability

- Based on conflicting operations: Read/Write or Write/Write on the **same data item**.
- A schedule is conflict-serializable **if it can be transformed** into a serial schedule by **swapping non-conflicting operations**.

Conflicting operations:

- **Read–Write** (RW)
- **Write–Read** (WR)
- **Write–Write** (WW)

Example:

T1: Read(A) → Write(A)
T2: Read(A) → Write(A)
This schedule may not be serializable due to a conflict on A.

2. View Serializability

- Weaker than conflict-serializability but still ensures correctness.
- Two schedules are view-equivalent if:
  1. They read the same initial values.
  2. Transactions read the same values from other transactions.
  3. The final write on each item is the same.

View-serializability is harder to check, but more permissive**.**

Serial Schedules vs Concurrent Schedules

Serial Schedule:

Transactions run **one after another** — safest but least efficient.
Example:

T1: R(A), W(A), R(B), W(B)
T2: R(C), W(C)

Concurrent Schedule:

Transactions **interleave** their operations.
Safe if it is **serializable**.

Real-Time Example: Online Banking

Scenario:

- Two users are transferring money from the **same account** simultaneously.

Transactions:

- **T1**: Transfer ₹500 from A to B.
- **T2**: Transfer ₹300 from A to C.

If they are not serializable, the **balance may be incorrectly deducted** or deducted **twice** from the same initial amount.

Enforcing serializability ensures correctness.

5. How to Enforce Serializability

Techniques:

Technique	Ensures Serializability	Notes
Two-Phase Locking (2PL)	Yes	Guarantees conflict-serializability
Timestamp Ordering	Yes	Uses timestamps to order operations
Validation-Based	Sometimes	Checks for serializability at commit

Summary Table

Concept	Description	Ensures
Serializability	Concurrent execution = some serial execution	Correctness
Conflict Serializable	Uses operation conflicts to verify order	Easy to test
View Serializable	Based on what data is read/written	More complex

I. Locking Techniques

Locking prevents unwanted interference by restricting access to data items.

Shared & Exclusive Locks

1. Shared Lock (S-Lock)

- Purpose: Used for reading data.
- Multiple transactions can hold shared locks on the same data item simultaneously.
- Does not allow writing.

Example:

Transaction T1: SELECT balance FROM accounts WHERE id = 1; -- Shared lock on row
Transaction T2: SELECT balance FROM accounts WHERE id = 1; -- Also allowed

2. Exclusive Lock (X-Lock)

- Purpose: Used for writing/updating data.
- Only one transaction can hold an exclusive lock on a data item.
- No other transaction (read or write) can access the item until the lock is released.

Example:

```
Transaction T1: UPDATE accounts SET balance = balance - 100 WHERE id = 1;  -- Exclusive lock

Transaction T2: SELECT balance FROM accounts WHERE id = 1;  -- Blocked until T1 finishes
```

Exclusive Lock (X)	Used for writing/modifying data	Only one exclusive lock at a time

Example:

```
-- Transaction 1: reads a row

LOCK TABLE accounts IN SHARE MODE;



-- Transaction 2: cannot write to that row until T1 is done

LOCK TABLE accounts IN EXCLUSIVE MODE;
```

- T2 (ATM 2): Withdraw ₹700

Problem Without Locking:

Both read balance ₹1000 and deduct money → Final balance becomes ₹3000–1200 = ₹-200 (invalid)

With Locking:

- T1 acquires Exclusive Lock on the account row.
- T2 waits until T1 commits or rolls back.
- Final balance is computed correctly.

Lock Compatibility Matrix

	Shared (S)	**Exclusive (X)**
Shared (S)	✓ Allowed	✗ Not allowed
Exclusive (X)	✗ Not allowed	✗ Not allowed

Two-Phase Locking (2PL)

Definition:

A concurrency control protocol that ensures conflict-serializability using a two-phase process:

Phases of 2PL

Phase	**Description**
Growing Phase	Transaction acquires all required locks (S or X)

Phase	Description
Shrinking Phase	Transaction releases locks; no new locks after this point

2PL Example:
T1:
-- **Growing phase**
LOCK X(A);
LOCK X(B);

-- Perform update
UPDATE A SET value = value - 100;
UPDATE B SET value = value + 100;

-- Shrinking phase
COMMIT; -- Releases locks
T1 must acquire all locks before releasing any — this guarantees serializability.

Real-World Case Study

Online Shopping System

Scenario: Two users update the stock of the same product.
- Transaction T1: Decreases stock by 1 (exclusive lock).
- Transaction T2: Tries to read stock (shared lock).

Without proper locking:
- T2 might read partially updated data, leading to wrong decisions.

With 2PL and proper locks:
- T2 must wait until T1 completes, ensuring consistency.

Inventory Management

Scenario: Two warehouse managers update product stock:
- T1: Adds 100 units
- T2: Removes 50 units

Without locking, race conditions may occur, → inaccurate stock levels.

With 2PL:
- Both transactions lock the stock record before updating.
- Only one proceeds at a time → The correct final value is preserved.

Summary Table

Technique	Description	Prevents	Real Case
Serializability	Ensures the correct order of concurrent transactions	All anomalies	Online shopping, banking
Shared Lock	Allows multiple readers	Prevents writes	Report generation
Exclusive Lock	Allows only one writer	Prevents conflicts	Balance updates
2PL	Locking is done in two phases	Lost updates, dirty reads	Inventory control

Recovery Techniques in DBMS

When a database system crashes due to power failure, software bugs, or hardware issues, **recovery techniques** ensure the system can restore the database to a **consistent state**, either by **redoing** or **undoing** transactions.

1. **Log-Based Recovery**

What is a Log?

A **log** is a sequence of records maintained by the DBMS, stored on stable storage, that **tracks all changes** made to the database.

Log Format:

Each log entry typically contains:

<Transaction ID, Data item, Old value, New value>

Recovery Scenarios:

Undo (Rollback):
If a transaction **fails before commit**, undo its changes using the **log**.
Redo (Rollforward):

If a transaction **commits but changes aren't yet in the database**, redo using the log.
Example:

Transaction T1:

BEGIN T1
Write(A, 100 → 200)
Write(B, 50 → 60)
COMMIT T1
Log:

<T1, A, 100, 200>
<T1, B, 50, 60>
<COMMIT T1>
After a crash:

- Since T1 **committed**, we **REDO** its changes using the log.

Write-Ahead Logging (WAL):

- **Log is written before** changes are made to the database.
- Ensures durability.

2. **Shadow Paging:**
An **alternative to log-based recovery**, **shadow paging** keeps two versions of database pages:

- **Current page table**
- **Shadow page table** (stable, unchanged)

How it works:

1. When a transaction starts, create a **shadow copy** of the current page table.
2. Changes are made to a **new page** (not in-place).
3. On **commit**, update the page table to point to new pages.
4. If the system crashes before commit, **discard changes** and use the shadow copy.

Example**:**

- **Page A** (original value 100)
- Transaction modifies Page A → new version (value 200)
- Commit → switch the pointer in the page table to the new page

If a crash occurs **before commit**, the system will use the **unchanged shadow page** (value 100), preserving consistency.
Advantages of Shadow Paging:

- No need for undo/redo logs.
- Atomic commit guaranteed by page pointer switch.

Disadvantages:

- Expensive in terms of copying and disk I/O.
- Difficult to handle concurrent transactions.
- Not widely used in modern systems.

Summary Table

Technique	**Key Idea**	**Redo/Undo**	**Use Case**
Log-Based	Record all changes in a log	✓ Yes	Most modern DBMS

Technique	Key Idea	Redo/Undo	Use Case
Shadow Paging	Use a copy of the page tables	✗ No	Simple recovery, not scalable

Real-World Usage

- **Log-Based Recovery** is used in:
  - Oracle
  - PostgreSQL
  - MySQL (InnoDB)
  - SQL Server
- **Shadow Paging** is more common in **academic examples** or **small embedded systems**.

Managing Concurrent Transactions in Banking Systems

What is Concurrency in Banking?

In real-world banking systems, **many users perform transactions simultaneously** (deposits, withdrawals, transfers). Concurrency is necessary for performance, but it brings **challenges** like inconsistent data, lost updates, or race conditions.

Common Issues in Concurrent Banking Transactions

Issue	Example
Lost Update	Two users withdraw from the same account → the final balance is wrong
Dirty Read	One transaction reads data modified by another uncommitted transaction.
.Unrepeatable Read	A row read twice gives different values in the same transaction
.Phantom Read	A query returns new rows when run again in the same transaction.

How Banking Systems Prevent These Issues

1. ACID Properties

Property	Role in Banking System
Atomicity	A transfer either completes fully or not at all.
Consistency	Account totals always satisfy business rules.
.Isolation	Multiple transfers don't interfere with each other.
Durability	Once committed, a transaction is not lost

. . .2. **Concurrency Control Mechanisms**

Two-Phase Locking (2PL)

- Ensures **serializability** by dividing the transaction into:
  - **Growing phase** (acquire locks)
  - **Shrinking phase** (release locks)

Example:

T1: Transfer ₹100 from A to B
-- Locks A and B
-- Updates both
-- Commits and releases locks

Isolation Levels (SQL Standard)

Level	Prevents
Read Uncommitted	✗ (no guarantees)
Read Committed	Prevents dirty reads
Repeatable Read	Prevents dirty & unrepeatable reads
Serializable	Prevents all anomalies

Banking systems often use **Repeatable Read** or **Serializable**.

3. Example: Concurrent Withdrawals

Scenario:

- Account A has ₹1000
- T1 withdraws ₹500
- T2 withdraws ₹600 at the same time

Without control:

- Both read ₹1000
- Final balance could be ₹-100

With 2PL or Serializable Isolation:

- T1 locks A, withdraws, commits
- T2 then reads updated balance ₹500 → denies overdraw

Balance integrity maintained.

4. Recovery with Logging

If a system crash occurs:

- Logs are used to **redo** committed transactions
- **Undo** uncommitted ones

Example:

<T1, A, 1000, 500>
<COMMIT T1>
After crash → T1 is redone

Real-World Techniques Used

Technique	Banking Use
WAL (Write-Ahead Logging)	Ensures durability and recovery
Strict 2PL	Enforces isolation and consistency
Serializable Isolation	Ensures no interference
Deadlock Detection	Avoid stuck transactions

Summary

Banking systems handle concurrency by combining:

- Transaction Management (ACID)
- Locking Mechanisms (2PL, Shared/Exclusive)
- High Isolation Levels (Serializable)
- Logging for Recovery

This ensures **accurate, consistent, and secure financial operations**, even with thousands of users simultaneously accessing the system.