Transactions, Isolation, and Locking: Concurrency Without Panic

What Transactions Actually Are

A transaction is a **unit of work** that's either fully completed or fully rolled back. No half-done states.

```
BEGIN;
  UPDATE accounts SET balance = balance - 100 WHERE id = 1;
  UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

Either both UPDATEs happen, or neither does. This is **atomicity**.

ACID Without the Buzzwords

Atomicity: All or Nothing

What it means: A transaction's changes are indivisible.

Example:

```
BEGIN;
  INSERT INTO orders (user_id, total) VALUES (1, 100);
  UPDATE users SET total_spent = total_spent + 100 WHERE id = 1;
COMMIT;
```

If the UPDATE fails (e.g., constraint violation), the INSERT is rolled back too.

No partial commits.

Consistency: Constraints Are Enforced

What it means: The database moves from one valid state to another.

Example:

- Before transaction: `SUM(balances) = 10000`
- After transaction: `SUM(balances) = 10000` (money moved, not created)

Constraints (foreign keys, checks) are enforced at commit time.

Isolation: Transactions Don't Interfere (Mostly)

What it means: Concurrent transactions appear to run serially (depending on isolation level).

The problem: Transaction A reads data while Transaction B is modifying it. What does A see?

Answer: Depends on the **isolation level** (more on this below).

Durability: Committed Data Survives Crashes

What it means: Once COMMIT returns, the data is safe (even if the server crashes 1ms later).

How: Write-ahead log (WAL). Changes are written to disk before COMMIT returns.

Trade-off: Durability = disk I/O = slower commits.

Isolation Levels: The Core Concept

SQL defines **four isolation levels** to balance consistency vs performance:

1. **READ UNCOMMITTED** (lowest isolation, highest concurrency)
2. **READ COMMITTED** (most common)
3. **REPEATABLE READ** (stricter)
4. **SERIALIZABLE** (strictest, lowest concurrency)

Higher isolation = more safety, lower performance.

The Concurrency Anomalies

Different isolation levels protect against different anomalies:

1. **Dirty Read:** Reading uncommitted data from another transaction
2. **Non-Repeatable Read:** Same query returns different results within a transaction
3. **Phantom Read:** New rows appear in a range query
4. **Serialization Anomaly:** Interleaved execution produces results impossible in serial execution

Isolation level chart:

Level	Dirty Read	Non-Repeatable Read	Phantom Read	Serialization Anomaly
READ UNCOMMITTED	❌ Allowed	❌ Allowed	❌ Allowed	❌ Allowed
READ COMMITTED	✅ Prevented	❌ Allowed	❌ Allowed	❌ Allowed
REPEATABLE READ	✅ Prevented	✅ Prevented	❌ Allowed (Postgres: ✅)	❌ Allowed
SERIALIZABLE	✅ Prevented	✅ Prevented	✅ Prevented	✅ Prevented

Isolation Level Examples

READ UNCOMMITTED: The Wild West

Not supported in Postgres. MySQL supports it (don't use it).

What happens:

- Transaction A can see uncommitted changes from Transaction B
- If B rolls back, A saw data that **never existed**

Use case: Almost never. Maybe for reporting where perfect accuracy doesn't matter.

READ COMMITTED: The Default

Postgres and MySQL default.

What it does: You only see committed data. But the data can change between queries.

Example: Non-Repeatable Read

```
Timeline:
-----------------------------------------
T1: BEGIN;
T1: SELECT balance FROM accounts WHERE id = 1;
    --> balance = 100

T2: BEGIN;
T2: UPDATE accounts SET balance = 200 WHERE id = 1;
T2: COMMIT;

T1: SELECT balance FROM accounts WHERE id = 1;
    --> balance = 200 (changed!)
T1: COMMIT;
```

Result: Same SELECT in T1 returns different values.

When it's a problem: If you're making decisions based on the first read (e.g., "if balance > 50, withdraw 50").

When it's fine: Most web apps. You're not making complex multi-step decisions.

REPEATABLE READ: Consistent Snapshot

Postgres default for explicit transactions.

What it does: You see a snapshot of the database as of the transaction start.

Example: Preventing Non-Repeatable Read

```
Timeline:
-----------------------------------------
T1: BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
T1: SELECT balance FROM accounts WHERE id = 1;
    --> balance = 100

T2: BEGIN;
T2: UPDATE accounts SET balance = 200 WHERE id = 1;
T2: COMMIT;

T1: SELECT balance FROM accounts WHERE id = 1;
    --> balance = 100 (unchanged in T1's snapshot!)
T1: COMMIT;
```

Result: T1 sees a consistent view. T2's changes are invisible to T1.

When it's useful: Multi-step reads that must be consistent (reports, analytics).

Postgres-specific: Prevents phantom reads too (stricter than SQL standard).

SERIALIZABLE: Total Isolation

What it does: Guarantees that concurrent transactions produce the same result as if they ran one at a time.

How: Detects conflicts and aborts transactions.

Example: Serialization Anomaly Prevented

```
Scenario: Two users booking the last seat on a flight.

T1: BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
T1: SELECT COUNT(*) FROM bookings WHERE flight_id = 123;
    --> 99 (one seat left)

T2: BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
T2: SELECT COUNT(*) FROM bookings WHERE flight_id = 123;
    --> 99 (one seat left)

T1: INSERT INTO bookings (flight_id, user_id) VALUES (123, 1);
T1: COMMIT;  --> Success

T2: INSERT INTO bookings (flight_id, user_id) VALUES (123, 2);
T2: COMMIT;  --> ERROR: could not serialize access
```

Result: T2 is aborted. Only one user gets the seat.

Cost: Higher chance of aborts, lower concurrency.

When to use: Financial transactions, inventory management, anything where correctness > performance.

MVCC: How Postgres Does Isolation

MVCC (Multi-Version Concurrency Control): Instead of locking rows, Postgres keeps multiple versions.

How it works:

1. Each transaction has a transaction ID (xid)
2. Each row has metadata: `xmin` (created by xid), `xmax` (deleted/updated by xid)
3. When you read a row, Postgres checks: "Is this version visible to my transaction?"

Example:

```
Initial state:
  Row: id=1, balance=100, xmin=10, xmax=NULL (visible to all)

T1 (xid=20): UPDATE accounts SET balance = 200 WHERE id = 1;
  New row: id=1, balance=200, xmin=20, xmax=NULL
  Old row: id=1, balance=100, xmin=10, xmax=20

T2 (xid=21, started before T1 committed):
  Sees old row (xmin=10, xmax=20, 21 > 20 but T1 hasn't committed yet)

T2 (after T1 commits):
  Sees old row (REPEATABLE READ snapshot rule)
```

Benefits:

- Readers don't block writers
- Writers don't block readers (unless updating the same row)

Cost:

- Old row versions accumulate (need VACUUM)
- Slightly more complex internals

MySQL InnoDB: Different Approach

InnoDB uses MVCC too, but with different defaults:

- **READ COMMITTED** by default
- **REPEATABLE READ** available, but with caveats (phantom reads possible)
- **SERIALIZABLE** uses locking (not true MVCC)

Locking: When Transactions Collide

Implicit Locks

What happens when you UPDATE a row:

```
UPDATE users SET balance = balance - 100 WHERE id = 1;
```

Postgres acquires a row-level lock on the row with `id = 1`.

Other transactions:

- Can still SELECT the row (MVCC)
- Can't UPDATE/DELETE it (blocked until first transaction commits/rolls back)

This prevents lost updates.

Explicit Locks: SELECT FOR UPDATE

Use case: "Lock this row so I can update it later."

```
BEGIN;
  SELECT balance FROM accounts WHERE id = 1 FOR UPDATE;
  -- (balance = 100)
  -- Now row is locked. Other transactions can't update it.
  UPDATE accounts SET balance = 90 WHERE id = 1;
COMMIT;
```

What FOR UPDATE does:

- Acquires a row lock
- Other transactions trying to SELECT FOR UPDATE / UPDATE block
- Ensures no one changes the row between your SELECT and UPDATE

When to use:

- Read-modify-write patterns
- Preventing race conditions (e.g., inventory checks)

SELECT FOR SHARE: Shared Lock

```sql
SELECT * FROM orders WHERE id = 123 FOR SHARE;
```

What it does:

- Allows other transactions to read (and also lock FOR SHARE)
- Blocks UPDATEs/DELETEs

Use case: Rare. Mostly used with referential integrity.

SKIP LOCKED: Non-Blocking Queue

```sql
SELECT * FROM jobs WHERE status = 'pending' ORDER BY created_at LIMIT 1 FOR UPDATE
SKIP LOCKED;
```

What it does:

- Tries to lock a row
- If row is already locked, **skips it** instead of blocking

Use case: Job queues (multiple workers, each grabs an available job).

NOWAIT: Fail Fast

```sql
SELECT * FROM accounts WHERE id = 1 FOR UPDATE NOWAIT;
```

What it does:

- Tries to lock a row
- If row is locked, **immediately errors** instead of blocking

Use case: User-facing apps where you don't want to wait (show "Resource busy" error).

Deadlocks: When Transactions Block Each Other

Scenario:

```
T1: BEGIN;
T1: UPDATE accounts SET balance = balance - 100 WHERE id = 1;  (locks row 1)

T2: BEGIN;
T2: UPDATE accounts SET balance = balance + 100 WHERE id = 2;  (locks row 2)

T1: UPDATE accounts SET balance = balance + 100 WHERE id = 2;  (waits for T2)
T2: UPDATE accounts SET balance = balance - 100 WHERE id = 1;  (waits for T1)
```

Result: Deadlock. Each transaction waits for the other.

Database action: Detects deadlock, aborts one transaction.

Error: `ERROR: deadlock detected`

Avoiding Deadlocks

1. **Consistent lock order:** Always lock rows in the same order (e.g., by ID ascending)
2. **Keep transactions short:** Less time holding locks = less chance of conflict
3. **Use timeouts:** `SET lock_timeout = '5s';`

Retrying After Deadlock

Application code:

```
async function transferMoney(from, to, amount) {
  let retries = 3;
  while (retries > 0) {
    try {
      await db.query('BEGIN');
      await db.query('UPDATE accounts SET balance = balance - $1 WHERE id = $2',
[amount, from]);
      await db.query('UPDATE accounts SET balance = balance + $1 WHERE id = $2',
[amount, to]);
      await db.query('COMMIT');
      return;
    } catch (err) {
      await db.query('ROLLBACK');
      if (err.code === '40P01') {  // Deadlock
        retries--;
        if (retries === 0) throw err;
        await sleep(100 * Math.random());  // Backoff
      } else {
        throw err;
      }
    }
  }
}
```

Transaction Isolation in ORMs

The Problem: ORMs Hide Transactions

Prisma example:

```
await prisma.user.update({ where: { id: 1 }, data: { balance: 100 } });
```

This is **auto-committed**. No explicit transaction.

If you need a transaction:

```
await prisma.$transaction(async (tx) => {
  await tx.user.update({ where: { id: 1 }, data: { balance: 100 } });
  await tx.user.update({ where: { id: 2 }, data: { balance: 200 } });
});
```

ORM Transaction Pitfalls

Pitfall 1: Long transactions with network round-trips

```
await prisma.$transaction(async (tx) => {
  const user = await tx.user.findUnique({ where: { id: 1 } });

  // Expensive API call (network I/O)
  const result = await fetch('https://external-api.com/slow');

  await tx.user.update({ where: { id: 1 }, data: { balance: result.value } });
});
```

Problem: Transaction is open for **seconds** (API call), holding locks.

Fix: Do external I/O **outside** the transaction:

```
const result = await fetch('https://external-api.com/slow');

await prisma.$transaction(async (tx) => {
  await tx.user.update({ where: { id: 1 }, data: { balance: result.value } });
});
```

Pitfall 2: Implicit isolation level

Prisma uses the database default (READ COMMITTED for Postgres). If you need REPEATABLE READ:

```
await prisma.$executeRaw`SET TRANSACTION ISOLATION LEVEL REPEATABLE READ`;
```

Or set it at the connection level.

Pitfall 3: Forgotten rollback on error

```
await prisma.$transaction(async (tx) => {
  await tx.user.update({ where: { id: 1 }, data: { balance: 100 } });
  throw new Error('Oops');  // Prisma auto-rolls back
});
```

Prisma handles this, but **manual SQL doesn't always**:

```
const client = await pool.connect();
try {
  await client.query('BEGIN');
  await client.query('UPDATE ...');
  throw new Error('Oops');  // You must ROLLBACK!
  await client.query('COMMIT');
} catch (err) {
  await client.query('ROLLBACK');  // Don't forget this!
  throw err;
} finally {
```

```
  client.release();
}
```

Real-World Transaction Patterns

Pattern 1: Optimistic Locking (Version Column)

Goal: Prevent lost updates without explicit locks.

Schema:

```
CREATE TABLE products (
  id INT PRIMARY KEY,
  name TEXT,
  stock INT,
  version INT NOT NULL DEFAULT 0
);
```

Application code:

```
// Read product
const product = await db.query('SELECT * FROM products WHERE id = $1', [productId]);

// User modifies stock (in UI)
const newStock = product.stock - 1;

// Update with version check
const result = await db.query(
  'UPDATE products SET stock = $1, version = version + 1 WHERE id = $2 AND version =
$3',
  [newStock, productId, product.version]
);

if (result.rowCount === 0) {
  throw new Error('Conflict: Product was modified by another user');
}
```

How it works:

- If another user updated the product, `version` has changed
- Your UPDATE matches 0 rows (version mismatch)
- You detect the conflict and retry or notify the user

Benefits:

- No locks held between read and write
- Scales well (no blocking)

Pattern 2: SELECT FOR UPDATE for Inventory

Goal: Ensure stock doesn't go negative.

```
await db.query('BEGIN');

const product = await db.query(
  'SELECT stock FROM products WHERE id = $1 FOR UPDATE',
  [productId]
);

if (product.stock < quantity) {
  await db.query('ROLLBACK');
  throw new Error('Insufficient stock');
}

await db.query(
  'UPDATE products SET stock = stock - $1 WHERE id = $2',
  [quantity, productId]
);

await db.query('COMMIT');
```

Benefits: No one can change stock between SELECT and UPDATE.

Pattern 3: Two-Phase Commit (Distributed Transactions)

Goal: Coordinate transactions across multiple databases.

Example: Transfer money between accounts in different databases.

Phase 1: Prepare

- DB1: BEGIN; UPDATE accounts SET balance = balance - 100; PREPARE TRANSACTION 'tx1';
- DB2: BEGIN; UPDATE accounts SET balance = balance + 100; PREPARE TRANSACTION 'tx2';

Phase 2: Commit or Rollback

- If both succeed: COMMIT PREPARED 'tx1'; COMMIT PREPARED 'tx2';
- If either fails: ROLLBACK PREPARED 'tx1'; ROLLBACK PREPARED 'tx2';

Complexity: High. Most apps avoid this (use eventual consistency instead).

Practical Guidelines

When to Use Transactions

✅ **Always:**

- Multi-step writes that must be atomic (e.g., order + payment)
- Read-modify-write patterns (check stock, then decrement)

✅ **Sometimes:**

- Batch inserts (faster, but not always necessary)
- Complex queries with consistency requirements

❌ **Never:**

- Long-running transactions (holding locks)

- Transactions spanning external API calls
- Read-only queries (unless you need REPEATABLE READ snapshot)

When to Use Higher Isolation Levels

READ COMMITTED (default): Most web apps.

REPEATABLE READ: Reports, analytics, multi-step reads.

SERIALIZABLE: Financial transactions, inventory with strict accuracy.

Keep Transactions Short

Bad:

```
await db.query('BEGIN');
await processHeavyLogic();  // 10 seconds
await db.query('UPDATE ...');
await db.query('COMMIT');
```

Good:

```
await processHeavyLogic();  // Do this first
await db.query('BEGIN');
await db.query('UPDATE ...');
await db.query('COMMIT');
```

Rule: Only database I/O should be inside transactions.

Key Takeaways

1. **Transactions are all-or-nothing.** COMMIT or ROLLBACK, no middle ground.
2. **Isolation levels trade consistency for performance.** READ COMMITTED is usually fine.
3. **MVCC (Postgres) allows readers and writers to coexist** without blocking.
4. **SELECT FOR UPDATE locks rows** for read-modify-write patterns.
5. **Deadlocks happen.** Detect and retry with backoff.
6. **ORMs hide transaction details.** Be explicit when you need atomicity.
7. **Keep transactions short.** Don't hold locks during external I/O.
8. **Use optimistic locking (version columns)** for long-running user interactions.
9. **SERIALIZABLE prevents all anomalies** but has lower concurrency.
10. **Vacuum regularly (Postgres)** to clean up old MVCC versions.

Next up: NULLs and three-valued logic—because NULL isn't what you think it is.