Database Transactions

An ACID trip

What is a Transaction?

> A **transaction (txn)** is a logical unit of work, composed of one or more database requests, that must be entirely completed or entirely aborted; no intermediate states are acceptable

Key principle: All or nothing!

Real-World Example: Bank Transfer

```sql
CREATE DATABASE bank;
USE bank;
CREATE TABLE account (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL,
    balance DECIMAL(10, 2) NOT NULL
);

INSERT INTO account (username, balance)
    VALUES('john', 100.10), ('jane', 617.50);
```

Transaction in Action

```sql
START TRANSACTION;

UPDATE account
SET Balance = Balance - 10
WHERE username = 'john';

UPDATE account
SET Balance = Balance + 10
WHERE username = 'jane';

COMMIT;

SELECT * FROM account;
```

Question: What happens if we crash between the two UPDATEs?

(Simulate using two sql sessions)

The ACID Properties

Every transaction must have:

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

These four properties are the foundation of reliable database systems.

Atomicity (A in ACID)

> Either all database requests in a transaction complete or none complete

e.g. bank Transfer: $100 from Account A to Account B, with faults

✓ ATOMIC

- Money deducted from A AND added to B (or neither)

✕ NOT ATOMIC

- Money deducted from A but NOT added to B (inconsistency!)

Time	Request
t0	Start txn
t1	Deduct $100 from A
t2	Add $100 to B
t3	Commit txn

Consistency (C in ACID)

> When a transaction completes, the database must be in a consistent state (entity integrity, referential integrity, constraints, etc.)

e.g. account balance cannot be negative

✓ CONSISTENT: Transaction rejected if it violates this rule
✕ INCONSISTENT: Negative balance allowed (breaks business logic)

Note: Individual requests in a transaction may temporarily violate constraints, but the complete transaction must not.

e.g deleting a parent row and cascading to existence-dependent children

Isolation (I in ACID)

> Data changes made by an in-progress transaction are not visible to another transaction until the first transaction completes

Critical in multi-user environments! (due to inter-leaved execution of txns)

Transaction 1: Read balance ($1000), add $100
Transaction 2: Read balance ($1000), subtract $50

✓ ISOLATED: Transactions execute in order, no conflicts
✗ NOT ISOLATED: Both read $1000, results conflict

Isolation Example

- Starting balance: $1,000
- T1: Read balance and add $100
- T2: Read balance and subtract $50
- Correct balance (serial execution): $1,000 + $100 - $50 = $1050

Interleaved execution:

Time	T1	T2
t0	Start	
t1	Read ($1,000) and modify +$100 ($1,100)	
t2		Start
t3		Read ($1,000 vs $1,100) and modify -$50 ($950 vs $1,050)
t4		Commit ($950 ✓ vs $1050 ✕)
t5	Commit ($1,100 ✕)	

Serializability

Key concept: Concurrent execution schedule yields the same result as if transactions executed serially

- DBMS doesn't execute serially (too slow!)
- Instead, carefully schedules concurrent execution
- Outcome appears as if they ran one-by-one

Durability (D in ACID)

> When a transaction is committed, it cannot be undone or lost, even in the event of a system failure

✓ DURABLE: Data committed to disk; survives power failure

✗ NOT DURABLE: Data only in memory; lost if system crashes

🤔 Active Learning: ACID Quiz

Match the scenario to the ACID property:

A. Power outage occurs, but committed data remains after restart

B. Transfer deducts from one account and credits another, or neither

C. One transaction doesn't see another's uncommitted changes

D. Foreign key constraints are maintained

The Transaction Log

How does a DBMS provide ACID guarantees?

Using a **transaction log**!

Transaction Log Structure

e.g. ordering qty 2 of a product

LSN	Txn ID	Operation	Table	Old Value	New Value	Prev LSN	Next LSN
100	500	BEGIN	-	-	-	NULL	101
101	500	UPDATE	Orders	id-6: status='PENDING'	id-6: status='PROCESSING'	100	102
102	500	UPDATE	Inventory	SKU-12345: qty=150	SKU-12345: qty=148	101	103
103	500	COMMIT	-	-	-	102	NULL

Write-Ahead Logging (WAL)

Mental model: Write down what you're about to do, *then* do it

- Log entry created **before** executing the database request
- Every materialized change has a log entry
- A log entry may exist without a materialized change

Benefit: Can recover from failures at any point

Database Recovery

When database crashes and restarts:

1. DBMS examines transaction log
2. Rolls back all uncommitted transactions
3. Persists any committed transactions not yet flushed to disk

The transaction log is stored separately from database files.

Concurrency Control

> Coordinating the simultaneous execution of transactions in a multiuser database system, while maintaining ACID properties (particularly isolation)

Why needed? Three main problems:

1. Lost updates
2. Uncommitted data (dirty reads)
3. Inconsistent retrievals

Problem 1: Lost Updates

Scenario: Two transactions update the same data, one update is overwritten

Initial balance: $1000

Time	Transaction T1	Transaction T2	Balance
t1	READ(1000)		1000
t2		READ(1000)	1000
t3	Balance = 1000 - 200		1000
t4		Balance = 1000 + 300	1000
t5	WRITE(800)		**800**
t6		WRITE(1300)	**1300**

Problem: The $200 withdrawal is completely lost!

Lost Updates: What Should Happen

Correct serial execution:

Time	Transaction T1	Transaction T2	Balance
t1-t5	Withdraw $200		1000 → **800**
t6-t10		Deposit $300	800 → **1100**

Expected: $1100 ✓

Without control: $1300 ✗ (lost the withdrawal!)

Problem 2: Uncommitted Data (Dirty Read)

Scenario: T2 reads T1's data before T1 commits (T1 later rolls back)

Time	Transaction T1	Transaction T2	Balance
t1-t4	Withdraw $500		1000 → **500**
t5		READ(500)	500
t6		Check: 500 ≥ 600? NO	500
t7		Reject withdrawal	500
t8	ROLLBACK		**1000**

Problem: T2 made a decision based on data that was rolled back!

Problem 3: Inconsistent Retrievals

Scenario: Transaction accesses data before and after another transaction modifies it

Initial state: A=$500, B=$300, C=$200 (Total = $1000)

Time	T1: Transfer $100 A→C	T2: Calculate Sum	A	B	C
t1		READ(A) = 500	500	300	200
t2	WRITE(A = 400)		**400**	300	200
t3	WRITE(C = 300)		400	300	**300**
t4		READ(B) = 300	400	300	300
t5		READ(C) = 300	400	300	300

Sum = 500 + 300 + 300 = 1100 ✗ (Money appeared from nowhere!)

🤔 Active Learning: Identify the Problem

For each scenario, identify which concurrency problem occurred:

1. A transaction reads account balance $500, another transaction deposits $200 and commits, first transaction overwrites with $400
2. A report shows inventory totals before and after a transfer between warehouses
3. An analytics query reads a customer's order that gets rolled back

The Scheduler

Special DBMS process that establishes operation order within concurrent transactions

Goals:

- Ensure isolation and serializability
- Use CPU and storage resources efficiently
- Prevent the three concurrency problems

How? Using concurrency control techniques!

Conflicting Operations Matrix

	T2: Read	T2: Write
T1: Read	✅ No Conflict	⚠️ Conflict
T1: Write	⚠️ Conflict	⚠️ Conflict

Key insight: Only Read-Read operations don't conflict!

Three Concurrency Control Techniques

1. **Locking (Pessimistic)**
    - Most commonly used
    - Assume conflicts are likely
2. **Timestamping**
    - Use timestamps to order transactions
3. **Optimistic**
    - Assume conflicts are rare
    - Detect conflicts at commit time

Locking (Pessimistic)

> Guarantees exclusive use of a data item to a transaction

Why "pessimistic"?

- Assumes conflicting transactions are likely
- Preemptively guards with exclusive access

Lock Manager:

- Assigns locks before data access
- Polices lock usage
- Releases locks after transaction completes

Locking Granularity

From worst to best:

Level	Granularity	Trade-off
Database	Entire DB	🔴 Serializes everything
Table	Whole table	🟠 Better, still coarse
Page	4KB disk page	🟡 Multiple rows
Row	Single row	🟢 Good balance
Field	Single field	🟢 Best, most complex

Most common: Row-level locking

Lock Types: Binary vs Shared/Exclusive

Binary Lock:

- Two states: locked or unlocked
- Exclusive access only
- Problem: Penalizes concurrent reads!

Shared/Exclusive Lock:

- Three states: unlocked, shared (read), exclusive (write)
- Multiple transactions can hold shared locks
- Only one transaction can hold exclusive lock

Exclusive Locks

```sql
START TRANSACTION;

-- 1. LOCK ACQUIRED: Exclusive lock grabbed here
SELECT stock_quantity FROM products WHERE product_id = 1 FOR UPDATE;

-- 2. LOCK HELD: Still holding the lock during all operations
UPDATE products SET stock_quantity = stock_quantity - 5 WHERE product_id = 1;
INSERT INTO orders (product_id, quantity) VALUES (1, 5);

-- 3. LOCK RELEASED: Lock released here
COMMIT;  --
```

Transaction T1:

```sql
START TRANSACTION;
SELECT balance FROM accounts WHERE account_id = 101 FOR UPDATE;
-- Lock acquired
```

Transaction T2 (concurrent):

```sql
START TRANSACTION;
SELECT balance FROM accounts WHERE account_id = 101 FOR UPDATE;
-- BLOCKS HERE - must wait for T1 to finish
```

Shared Locks

```
-- FOR SHARE: "I'm reading, don't change it"
START TRANSACTION;
SELECT balance FROM accounts WHERE account_id = 101 FOR SHARE;
-- Others can read with FOR SHARE, but cannot update

-- FOR UPDATE: "I'm about to change this, stay away"
START TRANSACTION;
SELECT balance FROM accounts WHERE account_id = 101 FOR UPDATE;
-- Nobody else can read with locks or write
```

Lock Compatibility Matrix:

	T2: Regular SELECT	T2: FOR SHARE	T2: FOR UPDATE	T2: UPDATE/DELETE
T1: Regular SELECT	✅ Compatible	✅ Compatible	✅ Compatible	✅ Compatible
T1: FOR SHARE	✅ Compatible	✅ Compatible	❌ Blocks	❌ Blocks
T1: FOR UPDATE	✅ Compatible	❌ Blocks	❌ Blocks	❌ Blocks
T1: UPDATE/DELETE	✅ Compatible	❌ Blocks	❌ Blocks	❌ Blocks

Locking Problems

Problem 1: Non-serializable schedules

- Transaction locks → unlocks → locks same data again
- Solution: **Two-Phase Locking (2PL)**
  - Growing phase: acquire all locks
  - Shrinking phase: release all locks

Problem 2: Deadlocks

- T1 has lock A, waits for B
- T2 has lock B, waits for A
- Solution: Deadlock detection & prevention (e.g., ordered lock acquisition)

Timestamping

Core idea: Assign unique, monotonically increasing timestamp to each transaction

How it works:

- Each database value has two timestamp fields:
  - Timestamp of latest read
  - Timestamp of latest write
- If transaction accesses data modified by more recent transaction:
  - Abort and retry with new timestamp

Less common than locking in practice

Optimistic Concurrency Control

Assumption: Majority of operations don't conflict

Three phases:

1. **Read phase:** Updates made to private copy
2. **Validation phase:** Check for conflicting commits
3. **Write phase:** Apply changes if validation passes

If conflict detected → abort and retry

Optimistic vs Pessimistic

Factor	Optimistic (OCC)	Pessimistic (Locks)
Low Contention	✅ Higher throughput	Lower (lock overhead)
High Contention	Lower (many retries)	✅ Better throughput
Read-Heavy	✅ Much higher	Lower (lock overhead)
Write-Heavy	Depends	✅ Often better
Latency	Low (no waiting)	Higher (blocking)
Wasted Work	High on conflicts	None

Key Takeaways

- ✅ **ACID properties** ensure reliable transactions
- ✅ **Transaction log** with WAL enables recovery
- ✅ **Concurrency control** prevents lost updates, dirty reads, and inconsistent retrievals
- ✅ **Locking** is most common (pessimistic approach)
- ✅ **Optimistic control** works well for low-contention scenarios
- ✅ **Choice of technique** depends on workload characteristics