Transactions: Multi-document transactions for ACID compliance.

1. Problem Statement

Case Study: FinTrust Wallet – Secure Money Transfers

FinTrust is a digital wallet platform where users can:

- Transfer money to each other instantly.
- Pay for purchases and receive refunds.
- Store balances and transaction histories.

Critical challenges:

- When Alice sends $100 to Bob, Alice's balance must decrease and Bob's must increase—**both or neither**.
- If a payment and a transaction log update don't both succeed, the system risks lost money or double credits.
- Regulatory compliance (banking, fintech) demands that transaction records are always correct—even if the server crashes or the network fails.

The challenge:
How can FinTrust guarantee that every transfer, payment, or refund is always complete, correct, and never leaves the system in a broken state, even when updating multiple documents or collections?

---

2. Learning Objectives

By the end of this tutorial, you will:

- Understand what ACID means in MongoDB and why it matters for critical data.

- Know how to use MongoDB's multi-document transactions for atomic, consistent, isolated, and durable operations.
- Learn the syntax and workflow for starting, committing, and aborting transactions.
- Apply best practices for error handling, performance, and regulatory compliance.
- Recognize when to use (and not use) transactions in MongoDB.

---

3. Concept Introduction with Analogy

Analogy: The Bank Vault Double-Lock

Imagine a bank transfer:

- The teller must take $100 from Alice's vault and add $100 to Bob's vault.
- If the teller can't open Bob's vault, Alice's money must be put back—**no partial moves allowed**.
- Every step is logged, and if the power fails mid-transfer, the system automatically reverses everything.

MongoDB's multi-document transactions are like this double-lock vault system:
They guarantee that all related changes succeed or none do, keeping everyone's money safe.

---

4. Technical Deep Dive

A. What is ACID?

ACID stands for:

- **Atomicity**: All operations in a transaction succeed or none do.
  If Alice's balance decreases, Bob's must increase, or both are rolled back.
- **Consistency**: Transactions always leave the database in a valid state.
  No user can have a negative balance after a transfer.
- **Isolation**: Transactions don't interfere; each sees a consistent snapshot.
  No one sees Bob's balance increase until Alice's decrease is committed.
- **Durability**: Once committed, the changes survive crashes or power failures.

Why is this important?
Without ACID, financial systems can lose money, double-spend, or create audit nightmares—unacceptable in banking, fintech, and regulated industries.

B. MongoDB Transactions: How They Work

- **Before MongoDB 4.0**: Only single-document operations were ACID-compliant.
- **MongoDB 4.0+**: Multi-document transactions are supported, just like relational databases.
- **How it works**:
  - Start a session.
  - Begin a transaction.

- Perform multiple reads/writes across collections.
- Commit to apply all changes, or abort to roll everything back.

- **Snapshot Isolation**: Each transaction sees a consistent view of the data, unaffected by others.

C. When Should You Use Transactions?

- **Best for:**
  - Financial transfers (wallets, payments, refunds).
  - Inventory management (reserve stock and update orders together).
  - Any workflow where multiple documents/collections must be updated as a unit.
- **Not always needed:**
  - Analytics, reporting, or mostly read-only operations.
  - Single-document updates (already atomic in MongoDB).

D. Transaction Syntax and Workflow

1. Start a session:

```
const session = db.getMongo().startSession();
```

2. Start a transaction:

```
session.startTransaction();
```

3. Perform operations (all must use the session):

```
db.users.updateOne(
  { _id: aliceId },
  { $inc: { balance: -100 } },
  { session }
);
db.users.updateOne(
  { _id: bobId },
  { $inc: { balance: 100 } },
  { session }
);
db.transactions.insertOne(
  { from: aliceId, to: bobId, amount: 100, date: new Date() },
  { session }
);
```

4. Commit or abort:

```
try {
  session.commitTransaction();
} catch (e) {
  session.abortTransaction();
  throw e;
} finally {
  session.endSession();
}
```

Key Notes:

- If any operation fails, abort the transaction—no changes are saved.
- All operations must use the same session object.
- Transactions can span collections and (since MongoDB 4.2) sharded clusters.

E. ACID in Practice: Guarantees and Limitations

- **Atomicity**: All or nothing—no partial changes.
- **Consistency**: Schema rules and business logic enforced at commit.
- **Isolation**: Other operations don't see changes until commit.
- **Durability**: Once committed, data is safe even after crashes.

Performance Consideration:

- Transactions add overhead; use only when needed for business logic and regulatory compliance.

---

5. Step-by-Step Data Modeling & Code Walkthrough

Let's walk through a real FinTrust wallet transfer:

A. User and Transaction Document Models

User:

```
{
  "_id": ObjectId("665f4d7e8b3e6c1e24a7b3e1"),
  "name": "Alice",
  "balance": 500.00
}
```

Transaction:

```
{
  "_id": ObjectId("..."),
  "from": ObjectId("665f4d7e8b3e6c1e24a7b3e1"),
  "to": ObjectId("665f4d7e8b3e6c1e24a7b3e2"),
  "amount": 100.00,
  "date": ISODate("2025-05-30T10:00:00Z"),
  "status": "completed"
}
```

B. Atomic Money Transfer with Transaction

```
const session = db.getMongo().startSession();
session.startTransaction();

try {
  // 1. Deduct from Alice
  db.users.updateOne(
    { _id: ObjectId("665f4d7e8b3e6c1e24a7b3e1") },
    { $inc: { balance: -100 } },
    { session }
  );
```

```
// 2. Add to Bob
db.users.updateOne(
  { _id: ObjectId("665f4d7e8b3e6c1e24a7b3e2") },
  { $inc: { balance: 100 } },
  { session }
);

// 3. Log the transaction
db.transactions.insertOne(
  {
    from: ObjectId("665f4d7e8b3e6c1e24a7b3e1"),
    to: ObjectId("665f4d7e8b3e6c1e24a7b3e2"),
    amount: 100,
    date: new Date(),
    status: "completed"
  },
  { session }
);

// 4. Commit
session.commitTransaction();
} catch (e) {
  session.abortTransaction();
  throw e;
} finally {
  session.endSession();
}
```

Explanation:

- If any step fails (e.g., Alice doesn't have enough balance), all changes are rolled back—no partial transfers or missing logs.

C. What Happens If There's an Error?

- If the server crashes or a write fails before `commitTransaction()`, MongoDB automatically aborts and rolls back all changes.
- If `commitTransaction()` succeeds, all changes are durable and visible to others.

D. Best Practices for MongoDB Transactions

- **Keep transactions short** to reduce lock contention and improve performance.
- **Always check for errors** and handle aborts gracefully.
- **Use transactions only when necessary**—single-document updates don't need them.
- **Monitor performance** and tune write concerns for your business needs

---

6. Interactive Challenge / Mini-Project

Your Turn!
You're building a new feature for FinTrust:

1. **Refund a payment:** If a user disputes a payment, you must:

- Add the refund amount back to the sender's balance.
- Subtract the amount from the recipient's balance.
- Update the original transaction's status to "refunded."
- Log a new transaction record as a refund.

2. **Ensure:**
   - If any step fails (e.g., recipient doesn't have enough balance to refund), no changes are made.
   - All operations are ACID-compliant.

Write a MongoDB transaction (pseudo-code or JavaScript) to implement this.

7. Common Pitfalls & Best Practices

Pitfall	Best Practice
Forgetting to use session in all ops	Always pass `{ session }` to each operation
Long-running transactions	Keep transactions short and focused
Not handling errors/aborts	Always use `try/catch/finally` and abort on error
Using transactions for simple updates	Use only when needed for multi-document changes
Not checking preconditions (e.g., balance)	Validate inside the transaction

8. Optional: Programmer's Workflow Checklist

- Start a session and transaction for multi-document updates.
- Pass `{ session }` to every read/write in the transaction.
- Use try/catch/finally to commit or abort as needed.
- Validate all business logic inside the transaction.
- Test failure scenarios to ensure rollback works.
- Monitor performance and adjust write concerns as needed.