Chapter 6: Database Fundamentals

1. SQL vs NoSQL Databases

The fundamental choice in database architecture. Understanding the differences is crucial for system design.

SQL Databases (Relational)

Structure: Data organized in tables with predefined schema.

```
Example: E-commerce Database

┌─────────────────────────────────────────┐
│         USERS TABLE                     │
├──────┬──────────┬──────────┬────────────┤
│ ID   │  Name    │  Email   │  City      │
├──────┼──────────┼──────────┼────────────┤
│ 1    │ John     │ j@x.com  │ NYC        │
│ 2    │ Jane     │ ja@x.com │ LA         │
│ 3    │ Bob      │ b@x.com  │ Chicago    │
└──────┴──────────┴──────────┴────────────┘

┌──────────────────────────────────────────────┐
│         ORDERS TABLE                         │
├──────┬─────────┬─────────┬───────────────────┤
│ ID   │ User_ID │  Total  │  Order_Date       │
├──────┼─────────┼─────────┼───────────────────┤
│ 101  │    1    │  $150   │  2024-01-15       │
│ 102  │    2    │  $200   │  2024-01-16       │
│ 103  │    1    │  $75    │  2024-01-17       │
└──────┴─────────┴─────────┴───────────────────┘

Relationships defined by foreign keys (User_ID → Users.ID)
```

Key Characteristics:

```
1. STRUCTURED SCHEMA
   - Must define tables and columns upfront
   - Data types enforced
   - Relationships defined via foreign keys

2. ACID TRANSACTIONS
   - Atomicity, Consistency, Isolation, Durability
   - Strong guarantees for data integrity

3. SQL LANGUAGE
   - Powerful query language
   - Joins, aggregations, complex queries

4. VERTICAL SCALING
   - Typically scale by upgrading hardware
   - Horizontal scaling possible but complex (sharding)
```

SQL Example:

```
sql
```

```
-- Create tables with schema
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    total DECIMAL(10, 2),
    order_date DATE,
    status VARCHAR(20)
);

-- Insert data
INSERT INTO users (name, email) VALUES ('John Doe', 'john@example.com');
INSERT INTO orders (user_id, total, order_date, status)
VALUES (1, 150.00, '2024-01-15', 'completed');

-- Complex query with JOIN
SELECT
    u.name,
    u.email,
    COUNT(o.id) as order_count,
    SUM(o.total) as total_spent
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE o.status = 'completed'
GROUP BY u.id, u.name, u.email
HAVING SUM(o.total) > 100
ORDER BY total_spent DESC;

-- Result:
┌────────────┬──────────────────┬─────────────┬─────────────┐
│ Name       │ Email            │ Order_Count │ Total_Spent │
├────────────┼──────────────────┼─────────────┼─────────────┤
│ John Doe   │ john@example.com │ 5           │ $1,250.00   │
│ Jane Smith │ jane@example.com │ 3           │ $450.00     │
└────────────┴──────────────────┴─────────────┴─────────────┘
```

Popular SQL Databases:

- **PostgreSQL** - Feature-rich, open source

- **MySQL** - Popular, widely used
- **Oracle** - Enterprise, powerful
- **SQL Server** - Microsoft, enterprise
- **SQLite** - Embedded, serverless

NoSQL Databases (Non-Relational)

Structure: Flexible schema, various data models.

Types of NoSQL Databases

1. Document Databases (MongoDB, CouchDB)

```json
```

```
// Example: MongoDB Document
{
  "_id": "507f1f77bcf86cd799439011",
  "name": "John Doe",
  "email": "john@example.com",
  "address": {
    "street": "123 Main St",
    "city": "NYC",
    "zip": "10001"
  },
  "orders": [
    {
      "order_id": 101,
      "total": 150.00,
      "date": "2024-01-15",
      "items": [
        {"product": "Laptop", "price": 1000},
        {"product": "Mouse", "price": 50}
      ]
    },
    {
      "order_id": 102,
      "total": 75.00,
      "date": "2024-01-17"
    }
  ],
  "preferences": {
    "newsletter": true,
    "notifications": ["email", "sms"]
  }
}

// Flexible schema - each document can have different fields!
// No need for joins - all related data in one document
```

MongoDB Query Examples:

```
javascript
```

```
// Find users in NYC
db.users.find({ "address.city": "NYC" })

// Find users with orders > $100
db.users.find({ "orders.total": { $gt: 100 } })

// Aggregate total spending
db.users.aggregate([
  { $unwind: "$orders" },
  { $group: {
      _id: "$_id",
      name: { $first: "$name" },
      totalSpent: { $sum: "$orders.total" }
    }
  },
  { $sort: { totalSpent: -1 } }
])

// Update nested field
db.users.updateOne(
  { "_id": "507f1f77bcf86cd799439011" },
  { $set: { "address.city": "LA" } }
)

// Add to array
db.users.updateOne(
  { "_id": "507f1f77bcf86cd799439011" },
  { $push: { "orders": { order_id: 103, total: 200 } } }
)
```

---

2. Key-Value Databases (Redis, DynamoDB)

```
Simple key → value mapping

┌─────────────────┬─────────────────────────────┐
│      Key       │          Value          │
├─────────────────┼─────────────────────────────┤
│ user:1000      │ {"name":"John","age":30}    │
│ session:abc123  │ {"user_id":1000,"expires":..│
│ cart:user:1000  │ ["item1","item2","item3"]   │
│ counter:pageviews│ 1523421                  │
└─────────────────┴─────────────────────────────┘

Super fast lookups - O(1) complexity
```

Redis Example:

```python
import redis
import json

r = redis.Redis(host='localhost', port=6379, decode_responses=True)

# Store user data
user = {"name": "John", "email": "john@example.com", "age": 30}
r.set('user:1000', json.dumps(user))

# Get user data
user_data = json.loads(r.get('user:1000'))

# Store with expiration (session)
r.setex('session:abc123', 3600, json.dumps({"user_id": 1000}))

# Increment counter
r.incr('pageviews')

# Store list (shopping cart)
r.lpush('cart:user:1000', 'item1', 'item2', 'item3')
cart_items = r.lrange('cart:user:1000', 0, -1)
```

---

3. Column-Family Databases (Cassandra, HBase)

```
Column-oriented storage - great for analytics

Row Key: user:1000
┌──────────────┬─────────────────────────────────────────┐
│Column Family│     Columns              │
├──────────────┼─────────────────────────────────────────┤
│ profile     │ name: "John"                  │
│           │ email: "john@example.com"          │
│           │ age: 30                      │
├──────────────┼─────────────────────────────────────────┤
│ orders      │ order:101: {"total": 150, ...}       │
│           │ order:102: {"total": 75, ...}        │
│           │ order:103: {"total": 200, ...}       │
├──────────────┼─────────────────────────────────────────┤
│ analytics   │ last_login: "2024-01-20"           │
│           │ total_orders: 15                  │
│           │ lifetime_value: 2500               │
└──────────────┴─────────────────────────────────────────┘

Benefits:
- Can add columns dynamically
- Efficient for reading/writing specific columns
- Scales horizontally easily
```

Cassandra Example:

```sql
```

```
-- CQL (Cassandra Query Language) - SQL-like but different!

-- Create keyspace (database)
CREATE KEYSPACE ecommerce WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': 3
};

-- Create table
CREATE TABLE users (
    user_id UUID PRIMARY KEY,
    name TEXT,
    email TEXT,
    orders MAP<TIMESTAMP, FROZEN<ORDER>>
);

-- Insert data
INSERT INTO users (user_id, name, email)
VALUES (uuid(), 'John Doe', 'john@example.com');

-- Query by primary key (super fast!)
SELECT * FROM users WHERE user_id = 123e4567-e89b-12d3-a456-426614174000;

-- Note: Can't do complex joins or WHERE on non-key columns
-- Must design data model around query patterns!
```

---

4. Graph Databases (Neo4j, ArangoDB)

```
Nodes and Relationships - great for connected data

Example: Social Network

    (John)
      |
  [FRIEND_OF]
      |
    (Jane)──[LIKES]──>(Post #1)
      |               |
  [FOLLOWS]       [TAGGED_IN]
      |               |
    (Bob)───────────────────(John)
      |
  [WORKS_AT]
      |
  (Company ABC)

Nodes: John, Jane, Bob, Post #1, Company ABC
Relationships: FRIEND_OF, LIKES, FOLLOWS, TAGGED_IN, WORKS_AT
```

Neo4j Example (Cypher Query Language):

```cypher
```

```
-- Create nodes
CREATE (john:Person {name: 'John', age: 30})
CREATE (jane:Person {name: 'Jane', age: 28})
CREATE (post:Post {title: 'My vacation', date: '2024-01-15'})

-- Create relationships
CREATE (john)-[:FRIEND_OF]->(jane)
CREATE (jane)-[:LIKES]->(post)
CREATE (john)-[:POSTED]->(post)

-- Find friends of friends (2 hops away)
MATCH (john:Person {name: 'John'})-[:FRIEND_OF*2]-(friend_of_friend)
RETURN DISTINCT friend_of_friend.name

-- Find shortest path between two people
MATCH path = shortestPath(
  (john:Person {name: 'John'})-[:FRIEND_OF*]-(bob:Person {name: 'Bob'})
)
RETURN path

-- Find influencers (people with most followers)
MATCH (person:Person)<-[:FOLLOWS]-(follower)
RETURN person.name, COUNT(follower) as followers
ORDER BY followers DESC
LIMIT 10

-- Friend recommendations (friends of friends who aren't already friends)
MATCH (me:Person {name: 'John'})-[:FRIEND_OF]->(friend)-[:FRIEND_OF]->(fof)
WHERE NOT (me)-[:FRIEND_OF]->(fof) AND me <> fof
RETURN fof.name, COUNT(DISTINCT friend) as mutual_friends
ORDER BY mutual_friends DESC
```

SQL vs NoSQL Comparison

Feature	SQL	NoSQL
Schema	Fixed (rigid)	Flexible (dynamic)
Data Model	Tables/Relations	Various (doc/kv/etc)
Scaling	Vertical (harder)	Horizontal (easier)
Transactions	ACID (strong)	BASE (eventual)
Queries	Complex SQL	Simple lookups
Joins	Built-in	Manual/Embedded
Consistency	Strong	Eventual

```
│ Performance     │ Complex queries     │ Simple operations    │
├─────────────────┼─────────────────────┼──────────────────────┤
│ Best For        │ - Financial systems │ - Real-time web      │
│                 │ - Complex queries   │ - Big data           │
│                 │ - Transactions      │ - Rapid development  │
│                 │ - Reporting         │ - Horizontal scaling │
└─────────────────┴─────────────────────┴──────────────────────┘
```

When to Use What

Use SQL when:

```
✓ Data structure is well-defined and stable
✓ Need complex queries and joins
✓ Require ACID transactions (financial data)
✓ Data integrity is critical
✓ Strong consistency required
✓ Relationships between entities are important

Examples:
- Banking systems (transactions, accounts)
- E-commerce orders (inventory, payments)
- ERP systems (complex business logic)
- Reporting and analytics (complex aggregations)
```

Use NoSQL when:

```
✓ Schema is flexible or evolving
✓ Need to scale horizontally (millions of users)
✓ Simple key-based lookups
✓ Can tolerate eventual consistency
✓ High write throughput needed
✓ Dealing with unstructured data

Examples:
- Social media (user profiles, posts, feeds)
- IoT sensor data (millions of writes/sec)
- Real-time analytics (clickstream data)
- Content management (flexible documents)
- Gaming leaderboards (fast key-value)
- Session stores (temporary data)
```

Use Both (Polyglot Persistence):

```
Modern applications often use multiple databases!

Example E-commerce Architecture:
┌─────────────────────────────────────────────────────┐
│           Application                  │
└───┬──────────┬────────────┬────────────┬─────────────┘
    │      │      │      │
    ↓      ↓      ↓      ↓
┌────────┐ ┌────────┐ ┌────────┐ ┌────────┐
│ MySQL │ │ Redis │ │ElasticMongo │
└────────┘ └────────┘ └────────┘ └────────┘
    │      │      │      │
Orders   Session   Search   Product
Payments  Cache    Catalog  Details

- MySQL: Transactional data (orders, payments)
- Redis: Session storage, caching
- Elasticsearch: Product search
- MongoDB: Product catalog (flexible schema)
```

---

2. ACID Properties

ACID ensures database transactions are processed reliably.

A - Atomicity

"All or Nothing" - Transaction either completes fully or not at all.

```
Example: Bank Transfer

Transfer $100 from Account A to Account B

Steps:
1. Deduct $100 from Account A
2. Add $100 to Account B

┌─────────────────────────────────────────────────┐
│       WITH ATOMICITY                    │
├─────────────────────────────────────────────────┤
│ BEGIN TRANSACTION                       │
│   UPDATE accounts SET balance = balance - 100 WHERE id = 'A';   │
│   UPDATE accounts SET balance = balance + 100 WHERE id = 'B';   │
│ COMMIT; -- Both succeed                 │
│                                 │
│ Result: A: $900, B: $1100  ✓ Correct      │
└─────────────────────────────────────────────────┘


┌─────────────────────────────────────────────────┐
│     WITHOUT ATOMICITY (Danger!)         │
├─────────────────────────────────────────────────┤
│ UPDATE accounts SET balance = balance - 100 WHERE id = 'A'; ✓│
│ -- Power failure! --                    │
│ UPDATE accounts SET balance = balance + 100 WHERE id = 'B'; ✗│
│                                 │
│ Result: A: $900, B: $1000  ✗ $100 lost! │
└─────────────────────────────────────────────────┘
```

SQL Example:

```
sql
```

```sql
-- With atomicity
BEGIN TRANSACTION;

-- Deduct from sender
UPDATE accounts SET balance = balance - 100
WHERE user_id = 1 AND balance >= 100;

-- Check if previous update affected a row
-- (ensures sender had enough money)
IF (ROW_COUNT() = 0) THEN
    ROLLBACK;  -- Abort transaction
    RAISE EXCEPTION 'Insufficient funds';
END IF;

-- Add to receiver
UPDATE accounts SET balance = balance + 100
WHERE user_id = 2;

-- Insert transfer record
INSERT INTO transfers (from_user, to_user, amount, timestamp)
VALUES (1, 2, 100, NOW());

COMMIT;  -- All steps succeed together!

-- If ANY step fails:
-- - All changes are rolled back
-- - Database returns to state before transaction
-- - Money doesn't disappear or duplicate
```

Real-World Application:

python

```python
import psycopg2

def transfer_money(from_account, to_account, amount):
    """Transfer money with atomicity"""
    conn = psycopg2.connect(...)
    cursor = conn.cursor()

    try:
        # Start transaction
        cursor.execute("BEGIN;")

        # Deduct from sender
        cursor.execute(
            "UPDATE accounts SET balance = balance - %s "
            "WHERE id = %s AND balance >= %s",
            (amount, from_account, amount)
        )

        if cursor.rowcount == 0:
            raise Exception("Insufficient funds")

        # Add to receiver
        cursor.execute(
            "UPDATE accounts SET balance = balance + %s WHERE id = %s",
            (amount, to_account)
        )

        # Log transaction
        cursor.execute(
            "INSERT INTO transfer_log (from_id, to_id, amount) VALUES (%s, %s, %s)",
            (from_account, to_account, amount)
        )

        # Commit - all changes applied together
        conn.commit()
        print("Transfer successful")

    except Exception as e:
        # Rollback - all changes undone
        conn.rollback()
        print(f"Transfer failed: {e}")

    finally:
        cursor.close()
        conn.close()
```

C - Consistency

"Data stays valid" - Transaction brings database from one valid state to another.

```
Example: Database Constraints

Constraint: balance >= 0 (no negative balances)

┌─────────────────────────────────────────┐
│       WITH CONSISTENCY                  │
├─────────────────────────────────────────┤
│ Account A balance: $50                  │
│                                         │
│ Try to transfer $100:                   │
│ BEGIN TRANSACTION;                      │
│   UPDATE accounts                       │
│   SET balance = balance - 100           │
│   WHERE id = 'A';                       │
│                                         │
│   -- Constraint violated! (50 - 100 = -50) │
│   ROLLBACK; -- Transaction rejected     │
│ END;                                    │
│                                         │
│ Result: Balance stays $50  ✓ Valid      │
└─────────────────────────────────────────┘
```

Types of Consistency Rules:

```sql
```

```sql
-- 1. Primary Key Constraint (uniqueness)
CREATE TABLE users (
    id SERIAL PRIMARY KEY,  -- Can't have duplicate IDs
    email VARCHAR(255)
);

-- 2. Foreign Key Constraint (referential integrity)
CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id)  -- user_id must exist in users table
);

-- Trying to insert order for non-existent user fails:
INSERT INTO orders (user_id, total) VALUES (999, 100);
-- ERROR: violates foreign key constraint

-- 3. Check Constraint (custom validation)
CREATE TABLE accounts (
    id SERIAL PRIMARY KEY,
    balance DECIMAL(10, 2) CHECK (balance >= 0),  -- No negative balances
    account_type VARCHAR(20) CHECK (account_type IN ('checking', 'savings'))
);

-- Trying to set negative balance fails:
UPDATE accounts SET balance = -100 WHERE id = 1;
-- ERROR: violates check constraint

-- 4. Unique Constraint (no duplicates)
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE  -- Each email must be unique
);

-- Trying to insert duplicate email fails:
INSERT INTO users (email) VALUES ('john@example.com');
INSERT INTO users (email) VALUES ('john@example.com');
-- ERROR: duplicate key value violates unique constraint

-- 5. Not Null Constraint (required field)
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,  -- Name is required
    price DECIMAL(10, 2) NOT NULL
);

-- Trying to insert without name fails:
```

```sql
INSERT INTO products (price) VALUES (19.99);
-- ERROR: null value in column "name" violates not-null constraint
```

Complex Consistency Example:

```sql
-- Business rule: Order total must equal sum of order items

CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    total DECIMAL(10, 2) NOT NULL
);

CREATE TABLE order_items (
    id SERIAL PRIMARY KEY,
    order_id INTEGER REFERENCES orders(id),
    product_id INTEGER,
    quantity INTEGER,
    price DECIMAL(10, 2)
);

-- Trigger to enforce consistency
CREATE OR REPLACE FUNCTION check_order_total()
RETURNS TRIGGER AS $$
DECLARE
    calculated_total DECIMAL(10, 2);
BEGIN
    SELECT SUM(quantity * price) INTO calculated_total
    FROM order_items
    WHERE order_id = NEW.order_id;

    IF calculated_total <> (SELECT total FROM orders WHERE id = NEW.order_id) THEN
        RAISE EXCEPTION 'Order total does not match sum of items';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER verify_order_total
AFTER INSERT OR UPDATE ON order_items
FOR EACH ROW EXECUTE FUNCTION check_order_total();
```

I - Isolation

"Transactions don't interfere" - Concurrent transactions execute as if serial.

```
Problem Without Isolation:

Two users booking last seat on a flight simultaneously:

Time    User A                    User B
─────────────────────────────────────────────────────────────

10:00   Check seats available: 1
10:01                             Check seats available: 1
10:02   Book seat (seats = 0)
10:03                             Book seat (seats = -1)

Result: -1 seats available! ✗ OVERBOOKING!
```

Isolation Levels:

Isolation Level	Dirty Read	Non-Repeatable	Phantom Read
Read Uncommitted (Weakest)	Yes	Yes	Yes
Read Committed (Default)	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable (Strongest)	No	No	No

1. Dirty Read Problem:

```
sql
```

```sql
-- Transaction A
BEGIN;
UPDATE accounts SET balance = 1000 WHERE id = 1;
-- Not yet committed!

-- Transaction B (reads uncommitted data)
SELECT balance FROM accounts WHERE id = 1;
-- Sees 1000 (dirty read!)

-- Transaction A
ROLLBACK;  -- Undoes the change!

-- Transaction B just read data that never existed!
```

2. Non-Repeatable Read Problem:

sql

```sql
-- Transaction A
BEGIN;
SELECT balance FROM accounts WHERE id = 1;
-- Reads: $1000

-- Transaction B
BEGIN;
UPDATE accounts SET balance = 500 WHERE id = 1;
COMMIT;

-- Transaction A (reads again in same transaction)
SELECT balance FROM accounts WHERE id = 1;
-- Now reads: $500 (different value!)

-- Same query, different result within one transaction!
```

3. Phantom Read Problem:

sql

```sql
-- Transaction A
BEGIN;
SELECT COUNT(*) FROM orders WHERE user_id = 1;
-- Reads: 5 orders

-- Transaction B
INSERT INTO orders (user_id, total) VALUES (1, 100);
COMMIT;

-- Transaction A (counts again)
SELECT COUNT(*) FROM orders WHERE user_id = 1;
-- Reads: 6 orders (phantom row appeared!)
```

Preventing Conflicts with Locks:

```
sql
```

```sql
-- Pessimistic locking (lock before read)
BEGIN;

-- SELECT FOR UPDATE locks the rows
SELECT * FROM seats
WHERE flight_id = 123 AND available = true
FOR UPDATE;

-- Other transactions must wait!
-- Safe to book now

UPDATE seats SET available = false, user_id = 1
WHERE seat_id = 456;

COMMIT;

-- Optimistic locking (check version before write)
BEGIN;

-- Read with version number
SELECT id, quantity, version FROM inventory WHERE product_id = 789;
-- quantity: 10, version: 5

-- User takes time to decide...

-- Try to update with version check
UPDATE inventory
SET quantity = 9, version = 6
WHERE product_id = 789 AND version = 5;

IF (ROW_COUNT() = 0) THEN
  -- Someone else updated it! Version changed.
  ROLLBACK;
  RAISE EXCEPTION 'Inventory was modified by another transaction';
END IF;

COMMIT;
```

D - Durability

"Changes are permanent" - Once committed, data survives crashes.

```
┌─────────────────────────────────────────┐
│         WITH DURABILITY                 │
├─────────────────────────────────────────┤
│ 10:00 User completes purchase           │
│ 10:01 Transaction COMMITTED             │
│ 10:02 Data written to disk              │
│ 10:03 ⚡ POWER FAILURE ⚡                 │
│ 10:05 System restarts                   │
│ 10:06 Database recovered                │
│ 10:07 Purchase is still there! ✓        │
└─────────────────────────────────────────┘

┌─────────────────────────────────────────┐
│         WITHOUT DURABILITY              │
├─────────────────────────────────────────┤
│ 10:00 User completes purchase           │
│ 10:01 Transaction said "committed"      │
│ 10:02 Data only in memory (not disk)    │
│ 10:03 ⚡ POWER FAILURE ⚡                 │
│ 10:05 System restarts                   │
│ 10:06 Database recovered                │
│ 10:07 Purchase is LOST! ✗               │
└─────────────────────────────────────────┘
```

How Durability is Achieved:

```
Write-Ahead Logging (WAL):

1. Transaction makes changes
   ↓
2. Changes written to LOG file first (sequential write - fast)
   ↓
3. Log flushed to disk (fsync)
   ↓
4. Transaction marked as committed
   ↓
5. Changes written to data files (can happen later)

If crash occurs:
- Replay log on restart
- Recover all committed transactions
- Rollback uncommitted ones

Example WAL entries:
┌──────────────────────────────────────────┐
│ LSN   │ TxnID │ Operation                │
├──────────────────────────────────────────┤
│ 1001  │ T1    │ BEGIN                    │
│ 1002  │ T1    │ UPDATE accounts SET bal=900 │
│ 1003  │ T1    │ UPDATE accounts SET bal=1100│
│ 1004  │ T1    │ COMMIT                   │
│ 1005  │ T2    │ BEGIN                    │
│ 1006  │ T2    │ INSERT INTO orders...    │
│ -- CRASH HERE --                         │
│                                          │
│ On restart:                              │
│ - T1 is committed (has COMMIT in log) → Keep│
│ - T2 is not committed (no COMMIT) → Discard │
└──────────────────────────────────────────┘
```

Database Durability Features:

```sql
```

```
-- PostgreSQL: Configure durability vs performance trade-off

-- Full durability (safest, slower)
fsync = on                          -- Force writes to disk
synchronous_commit = on             -- Wait for disk confirmation
wal_sync_method = fdatasync         -- How to force writes

-- Relaxed durability (faster, small risk)
synchronous_commit = off            -- Don't wait for disk (recent commits might be lost on crash)

-- Check if transaction is durable
COMMIT;  -- Returns only after data is on disk (with synchronous_commit=on)
```

---

3. Database Indexes and How They Work

Indexes make queries fast by creating optimized data structures for lookups.

Without Index

```
Table: users (1 million rows)

Query: SELECT * FROM users WHERE email = 'john@example.com';

Without index:
┌─────────────────────────────────────┐
│  Full Table Scan                    │
│  (Sequential search through all rows)│
├─────────────────────────────────────┤
│ Row 1: email = 'alice@...'     ✗    │
│ Row 2: email = 'bob@...'       ✗    │
│ Row 3: email = 'charlie@...'   ✗    │
│ ...                                 │
│ Row 543,234: email = 'john@...' ✓   │
│ ...continue checking...             │
│ Row 1,000,000: email = 'zoe@...' ✗  │
└─────────────────────────────────────┘

Time: O(n) = ~1,000,000 comparisons
Slow! Takes seconds.
```

With Index

```
Index on email column:
(B-Tree structure - sorted and hierarchical)
```

```
                   [M]
                  /   \
                 /     \
           [D]           [S]
          /   \         /   \
        [B]   [G]     [P]   [W]
        / \   / \     / \   / \
      [...email addresses...]

Query: SELECT * FROM users WHERE email = 'john@example.com';

With index:
1. Start at root: 'john' < 'M', go left
2. At [D]: 'john' > 'D', go right
3. At [G]: 'john' > 'G', go right
4. Found in leaf node!

Time: O(log n) = ~20 comparisons
Fast! Milliseconds.

50,000x faster for 1 million rows!
```

Types of Indexes

1. B-Tree Index (Default)

Structure: Balanced tree, sorted data

```
B-Tree for numbers [1, 5, 10, 15, 20, 25, 30, 35, 40]


                [20]
               /    \
              /      \
          [10]        [30]
          /  \        /   \
      [5]   [15]  [25]   [35]
      / \   / \    / \    / \
    [1] .. .. .. .. .. .. .. [40]

Properties:
- Always balanced (all paths same length)
- Sorted (enables range queries)
- Each node has multiple keys
- Efficient for: =, <, >, <=, >=, BETWEEN
```

SQL Examples:

```sql
-- Create B-Tree index (default)
CREATE INDEX idx_users_email ON users(email);

-- Queries that use the index:
SELECT * FROM users WHERE email = 'john@example.com';  -- Exact match
SELECT * FROM users WHERE email LIKE 'john%';          -- Prefix match
SELECT * FROM users WHERE email > 'a@example.com';     -- Range
SELECT * FROM users WHERE email BETWEEN 'a' AND 'n';   -- Range

-- Queries that DON'T use the index:
SELECT * FROM users WHERE email LIKE '%john%';         -- Wildcard at start
SELECT * FROM users WHERE LOWER(email) = 'john@...';   -- Function on column
```

---

2. Hash Index

Structure: Hash table for exact matches only

```
Hash Function: email → hash value → bucket

john@example.com → hash() → 12345 → Bucket 5
jane@example.com → hash() → 67890 → Bucket 3
bob@example.com  → hash() → 24680 → Bucket 7

┌─────────┬──────────────────┐
│ Bucket │ Data Pointers  │
├─────────┼──────────────────┤
│   0    │ ...          │
│   1    │ ...          │
│   2    │ ...          │
│   3    │ → jane@...      │
│   4    │ ...          │
│   5    │ → john@...      │
│   6    │ ...          │
│   7    │ → bob@...       │
└─────────┴──────────────────┘

Pros:
- O(1) lookup for exact matches (faster than B-Tree)

Cons:
- Can't do range queries
- Can't do sorting
- Not widely supported (PostgreSQL has it)
```

SQL Example:

```sql
-- Create hash index (PostgreSQL)
CREATE INDEX idx_users_email_hash ON users USING HASH (email);

-- Good query:
SELECT * FROM users WHERE email = 'john@example.com';  -- O(1) lookup

-- Won't use index:
SELECT * FROM users WHERE email > 'a@example.com';     -- Can't do ranges
SELECT * FROM users ORDER BY email;                    -- Can't sort
```

---

3. Full-Text Search Index

For searching within text content

```sql
-- Create full-text index (PostgreSQL)
CREATE INDEX idx_posts_content_fts
ON posts USING GIN (to_tsvector('english', content));

-- Search queries
-- Find posts containing "database" and "optimization"
SELECT * FROM posts
WHERE to_tsvector('english', content) @@ to_tsquery('database & optimization');

-- Find posts with "quick brown fox" (phrase)
SELECT * FROM posts
WHERE to_tsvector('english', content) @@ phraseto_tsquery('quick brown fox');

-- Ranked search results
SELECT
    title,
    content,
    ts_rank(to_tsvector('english', content), query) as rank
FROM posts, to_tsquery('database | optimization') query
WHERE to_tsvector('english', content) @@ query
ORDER BY rank DESC;
```

---

4. Composite Index (Multi-Column)

Index on multiple columns

```sql
```

```
-- Create composite index
CREATE INDEX idx_users_city_age ON users(city, age);

-- Column order matters!

-- Uses index (city is first):
SELECT * FROM users WHERE city = 'NYC';                ✓
SELECT * FROM users WHERE city = 'NYC' AND age > 25;    ✓

-- Doesn't use index (skips first column):
SELECT * FROM users WHERE age > 25;                   ✗

-- Analogy: Phone book sorted by (LastName, FirstName)
-- Can find "Smith, John" ✓
-- Can find all "Smith" ✓
-- Cannot efficiently find all "John" (without last name) ✗
```

Composite Index Structure:

```
Index on (city, age):

              NYC, 25-35
             /        \
            /          \
     LA, 20-30          NYC, 36-50
       / \                /  \
      /   \              /    \
   [data]  [data]     [data]   [data]

Efficient queries:
1. WHERE city = 'NYC'                    (uses city)
2. WHERE city = 'NYC' AND age = 30       (uses both)
3. WHERE city = 'NYC' AND age > 25       (uses both)

Inefficient queries:
4. WHERE age = 30                        (skips city, can't use index)
```

Index Design Best Practices:

```
sql
```

```
-- Bad: Too many columns
CREATE INDEX idx_bad ON users(city, state, age, income, gender);
-- Rarely used, takes up space, slows down writes

-- Good: Most selective column first
CREATE INDEX idx_good ON users(email, created_at);
-- email is unique, very selective

-- Good: Cover query
CREATE INDEX idx_cover ON users(city, age, name);
-- Query can be answered entirely from index:
SELECT name FROM users WHERE city = 'NYC' AND age = 30;
-- No need to access table!

-- Partial index (only index subset)
CREATE INDEX idx_active_users ON users(email)
WHERE active = true;
-- Only indexes active users, saves space
```

Index Performance Impact

Operation	No Index	With Index	Difference
SELECT (1M rows)	5,000 ms	5 ms	1,000x
INSERT	1 ms	2 ms	2x slower
UPDATE	1 ms	3 ms	3x slower
DELETE	1 ms	2 ms	2x slower

```
Key Points:
- Indexes speed up reads dramatically
- Indexes slow down writes (must update index too)
- Balance: Index frequently queried columns
- Don't index everything!
```

Checking Index Usage:

```
sql
```

```
-- PostgreSQL: Explain query plan
EXPLAIN ANALYZE
SELECT * FROM users WHERE email = 'john@example.com';

-- Output with index:
Index Scan using idx_users_email on users  (cost=0.42..8.44 rows=1 width=100) (actual time=0.025..0.026 rows=1 loops=1)
  Index Cond: (email = 'john@example.com'::text)
Planning Time: 0.123 ms
Execution Time: 0.052 ms

-- Output without index:
Seq Scan on users  (cost=0.00..25000.00 rows=1 width=100) (actual time=45.234..3421.567 rows=1 loops=1)
  Filter: (email = 'john@example.com'::text)
  Rows Removed by Filter: 999999
Planning Time: 0.089 ms
Execution Time: 3421.623 ms

-- 65,000x slower without index!
```

When NOT to Use Indexes:

```sql
-- 1. Small tables (< 1000 rows)
-- Full scan might be faster than index lookup

-- 2. Columns with low cardinality (few distinct values)
CREATE INDEX idx_bad ON users(gender);  -- Only 'M', 'F', 'Other'
-- Not selective enough, won't help much

-- 3. Frequently updated columns
CREATE INDEX idx_bad ON page_views(view_count);
-- view_count changes constantly, index constantly rebuilt

-- 4. Columns used in expressions
-- Index on 'email' doesn't help:
SELECT * FROM users WHERE LOWER(email) = 'john@example.com';
-- Need function-based index instead:
CREATE INDEX idx_lower_email ON users(LOWER(email));
```

---

4. Normalization vs Denormalization

Two opposite approaches to organizing data.

Normalization

Minimize redundancy by splitting data into multiple tables.

Example: E-commerce Database

Denormalized (Bad):

```
Orders Table (Everything in one place):
┌────┬─────────┬──────────────┬─────────┬────────┬─────┐
│ ID │  Name   │    Email     │ Product│ Price  │ Qty │
├────┼─────────┼──────────────┼─────────┼────────┼─────┤
│ 1  │ John    │ john@x.com   │ Laptop │ $1000  │  1  │
│ 2  │ John    │ john@x.com   │ Mouse  │ $50    │  2  │
│ 3  │ Jane    │ jane@x.com   │ Laptop │ $1000  │  1  │
└────┴─────────┴──────────────┴─────────┴────────┴─────┘

Problems:
- Name and email duplicated (data redundancy)
- If John changes email, must update multiple rows
- If product price changes, must update all orders
- Wasted storage space
```

Normalized (Good):

```
Users Table:
┌────┬──────────┬──────────────┐
│ ID │   Name   │    Email     │
├────┼──────────┼──────────────┤
│ 1  │  John    │ john@x.com   │
│ 2  │  Jane    │ jane@x.com   │
└────┴──────────┴──────────────┘

Products Table:
┌────┬──────────┬──────────┐
│ ID │  Name    │  Price   │
├────┼──────────┼──────────┤
│ 10 │ Laptop   │ $1000    │
│ 11 │ Mouse    │  $50     │
└────┴──────────┴──────────┘

Orders Table:
┌────┬──────────┬────────────┬──────────┐
│ ID │ User_ID  │ Product_ID │   Qty    │
├────┼──────────┼────────────┼──────────┤
│ 1  │    1     │    10      │    1     │
│ 2  │    1     │    11      │    2     │
│ 3  │    2     │    10      │    1     │
└────┴──────────┴────────────┴──────────┘

Benefits:
- No duplication
- Update once (e.g., email change only in users table)
- Consistent data
- Less storage
```

Normal Forms

1NF (First Normal Form)

Rule: No repeating groups, atomic values only

```
❌ Not 1NF:
┌────┬──────────┬──────────────────────┐
│ ID │   Name   │    Phone_Numbers     │
├────┼──────────┼──────────────────────┤
│ 1  │  John    │ 555-1234, 555-5678   │ ← Multiple values in one cell
│ 2  │  Jane    │ 555-9999             │
└────┴──────────┴──────────────────────┘

✓ 1NF:
┌────┬──────────┬─────────────┐
│ ID │   Name   │    Phone    │
├────┼──────────┼─────────────┤
│ 1  │  John    │ 555-1234    │
│ 1  │  John    │ 555-5678    │ ← Separate rows
│ 2  │  Jane    │ 555-9999    │
└────┴──────────┴─────────────┘
```

2NF (Second Normal Form)

Rule: 1NF + No partial dependencies (all non-key columns depend on entire primary key)

❌ Not 2NF:

Order_Items Table (Composite key: Order_ID + Product_ID):

Order_ID	Product_ID	Product_Name	Quantity
1	10	Laptop	1
1	11	Mouse	2

Problem: Product_Name depends only on Product_ID, not on full key

✓ 2NF:

Order_Items Table:

Order_ID	Product_ID	Quantity
1	10	1
1	11	2

Products Table:

Product_ID	Product_Name
10	Laptop
11	Mouse

3NF (Third Normal Form)

Rule: 2NF + No transitive dependencies (non-key columns don't depend on other non-key columns)

❌ Not 3NF:

Employees Table:

ID	Name	Zip	City
1	John	10001	New York
2	Jane	90001	Los Angeles

Problem: City depends on Zip, not on ID (transitive dependency)

✓ 3NF:

Employees Table:

ID	Name	Zip
1	John	10001
2	Jane	90001

Zip_Codes Table:

Zip	City
10001	New York
90001	Los Angeles

Denormalization

Add redundancy for performance by combining tables.

When to Denormalize:

```
Scenario: E-commerce product listing

Normalized (requires JOIN):
┌─────────────────────────────────────┐
│ SELECT p.name, p.price, c.name      │
│ FROM products p                     │
│ JOIN categories c ON p.category_id = c.id │
│                                     │
│ Every query needs JOIN = slower     │
└─────────────────────────────────────┘

Denormalized (single table):
┌────┬────────┬────────┬───────────────┐
│ ID │  Name  │ Price  │ Category_Name │
├────┼────────┼────────┼───────────────┤
│ 1  │ Laptop │ $1000  │ Electronics   │
│ 2  │ Mouse  │ $50    │ Electronics   │
│ 3  │ Desk   │ $300   │ Furniture     │
└────┴────────┴────────┴───────────────┘

Benefits:
- No JOIN needed = faster queries
- Simpler queries
- Better for read-heavy workloads

Trade-offs:
- Data duplication
- Update anomalies (must update category name in multiple places)
- More storage
```

Real-World Example: Social Media Feed

Normalized (Slow):

```sql
```

```sql
-- Get user's feed (posts from friends)
SELECT
    p.id,
    p.content,
    p.created_at,
    u.name as author_name,
    u.profile_pic,
    COUNT(l.id) as like_count,
    COUNT(c.id) as comment_count
FROM posts p
JOIN friendships f ON f.friend_id = p.user_id
JOIN users u ON u.id = p.user_id
LEFT JOIN likes l ON l.post_id = p.id
LEFT JOIN comments c ON c.post_id = p.id
WHERE f.user_id = 123  -- Current user
GROUP BY p.id, p.content, p.created_at, u.name, u.profile_pic
ORDER BY p.created_at DESC
LIMIT 20;

-- Problem: 4 JOINs, GROUP BY, COUNT
-- On 1000 friends with 10k posts: very slow!
```

Denormalized (Fast):

```
sql
```

```sql
-- Pre-computed feed table
CREATE TABLE user_feeds (
    user_id INTEGER,
    post_id INTEGER,
    post_content TEXT,
    author_name VARCHAR(100),
    author_pic VARCHAR(255),
    like_count INTEGER,
    comment_count INTEGER,
    created_at TIMESTAMP,
    PRIMARY KEY (user_id, post_id)
);

-- Simple query, no JOINs!
SELECT *
FROM user_feeds
WHERE user_id = 123
ORDER BY created_at DESC
LIMIT 20;

-- 100x faster!

-- Trade-off: Must update feed when:
-- - Friend posts new content
-- - Post gets liked/commented
-- - Friend's profile picture changes
```

Normalization vs Denormalization Comparison

Feature	Normalization	Denormalization
Data Redundancy	Minimal	High
Storage Space	Less	More
Update Speed	Fast	Slow
Read Speed	Slow (JOINs)	Fast (no JOINs)
Data Integrity	Strong	Weaker
Consistency	Easy	Complex
Query Complexity	Complex	Simple
Best For	OLTP systems	OLAP systems
	Write-heavy	Read-heavy
	Frequent updates	Reporting
	Consistency	Performance

```
└────────────────────┴────────────────────┴────────────────────┘

Decision Framework:
- Normalize by default
- Denormalize specific hot paths for performance
- Use caching before denormalizing
- Monitor and measure before deciding
```

---

5. Connection Pooling

Managing database connections efficiently.

The Problem

```
Without Connection Pooling:

Request 1 arrives
   ↓
Open new DB connection (50ms)
   ↓
Execute query (10ms)
   ↓
Close connection (10ms)
   ↓
Total: 70ms

With 1000 concurrent requests:
- Open 1000 connections
- Database overwhelmed!
- Server runs out of resources
- Connections time out
- ☠️ System crashes
```

The Solution

```
Connection Pool:


┌─────────────────────────────────────────┐
│     Application Server             │
│                             │
│  ┌──────────────────────────────────┐  │
│  │   Connection Pool            │  │
│  │  (Maintains 10-50 connections)  │  │
│  │                             │  │
│  │  ┌────┐  ┌────┐  ┌────┐  ┌────┐  │  │
│ │ │Conn│ │Conn│ │Conn│ │Conn│ │ │
│ │ │ 1 │ │ 2 │ │ 3 │ │ 4 │ │ │
│ │ └──────┘ └──────┘ └──────┘ └──────┘ │ │
│ │ (pre-opened, ready to use) │ │
│ └─────────────────────────────────────────┘ │
└──────────────────────────────────────────────┘
     │ │ │ │
     └──┼──┼──┘
       ↓ ↓
    ┌──────────────┐
    │ Database │
    └──────────────┘

Request arrives:
1. Borrow connection from pool (instant!)
2. Execute query (10ms)
3. Return connection to pool (instant!)
Total: 10ms (7x faster!)
```

How Connection Pooling Works

```
Lifecycle:

1. INITIALIZATION
  ┌─────────────────────────────┐
  │ App starts │
  │ Pool creates 10 connections │
  │ Connections stay open │
  └─────────────────────────────┘

2. REQUEST HANDLING
  ┌─────────────────────────────┐
  │ Request arrives │
  │ Get connection from pool│
  │ Connection already open!│
  │ Execute query │
  │ Return to pool │
  └─────────────────────────────┘

3. POOL MANAGEMENT
  ┌─────────────────────────────┐
  │ Monitor connections │
  │ Replace dead ones │
  │ Grow pool if needed │
  │ Shrink pool if idle   │
  └─────────────────────────────┘
```

Implementation Examples

Python with psycopg2

```
python
```

```
from psycopg2 import pool
import threading
import time

# Create connection pool
connection_pool = pool.SimpleConnectionPool(
    minconn=5,         # Minimum connections
    maxconn=20,        # Maximum connections
    host="localhost",
    database="mydb",
    user="user",
    password="password"
)

def execute_query(user_id):
    """Execute query using pooled connection"""
    # Get connection from pool
    conn = connection_pool.getconn()

    try:
        cursor = conn.cursor()

        # Execute query
        cursor.execute(
            "SELECT * FROM users WHERE id = %s",
            (user_id,)
        )
        result = cursor.fetchone()

        print(f"[Thread {threading.current_thread().name}] Got user: {result}")

        return result

    finally:
        # Always return connection to pool!
        cursor.close()
        connection_pool.putconn(conn)

# Simulate 100 concurrent requests
threads = []
for i in range(100):
    t = threading.Thread(target=execute_query, args=(i,))
    threads.append(t)
    t.start()

# Wait for all to complete
```

```
for t in threads:
    t.join()

# Close pool on shutdown
connection_pool.closeall()
```

Python with SQLAlchemy (Better)

```python
from sqlalchemy import create_engine, text
from sqlalchemy.orm import sessionmaker

# Create engine with connection pool
engine = create_engine(
    'postgresql://user:password@localhost/mydb',
    pool_size=10,              # Normal pool size
    max_overflow=20,           # Additional connections if pool exhausted
    pool_timeout=30,           # Wait 30s for available connection
    pool_recycle=3600,         # Recycle connections after 1 hour
    pool_pre_ping=True,        # Test connection before using
    echo=False                 # Don't log SQL (set True for debugging)
)

# Create session factory
Session = sessionmaker(bind=engine)

def get_user(user_id):
    """Get user using pooled connection"""
    session = Session()  # Gets connection from pool

    try:
        result = session.execute(
            text("SELECT * FROM users WHERE id = :id"),
            {"id": user_id}
        )
        user = result.fetchone()
        return user

    finally:
        session.close()  # Returns connection to pool

# Usage
user = get_user(123)
```

Node.js with pg (PostgreSQL)

javascript

```
const { Pool } = require('pg');

// Create connection pool
const pool = new Pool({
  host: 'localhost',
  database: 'mydb',
  user: 'user',
  password: 'password',
  port: 5432,

  // Pool configuration
  min: 5,                    // Minimum connections
  max: 20,                   // Maximum connections
  idleTimeoutMillis: 30000,  // Close idle connections after 30s
  connectionTimeoutMillis: 2000, // Wait 2s for available connection
});

// Execute query using pool
async function getUser(userId) {
  const client = await pool.connect();  // Get from pool

  try {
    const result = await client.query(
      'SELECT * FROM users WHERE id = $1',
      [userId]
    );
    return result.rows[0];

  } finally {
    client.release();  // Return to pool
  }
}

// Simplified query (pool handles connection automatically)
async function getUserSimple(userId) {
  const result = await pool.query(
    'SELECT * FROM users WHERE id = $1',
    [userId]
  );
  return result.rows[0];
}

// Handle 1000 concurrent requests easily!
Promise.all(
  Array.from({length: 1000}, (_, i) => getUser(i))
).then(users => {
  console.log(`Processed ${users.length} queries`);
});

// Graceful shutdown
process.on('SIGINT', async () => {
  await pool.end();
  process.exit(0);
});
```

Pool Configuration Parameters

Parameter	Description
min_size	Always keep this many connections
	Typical: 5-10
max_size	Never exceed this many connections
	Typical: 20-100
	Formula: (CPU cores × 2) + disk
timeout	Wait time for available connection
	Typical: 5-30 seconds
max_lifetime	Recreate connection after this time
	Typical: 1-2 hours
	Prevents stale connections
idle_timeout	Close idle connections after this
	Typical: 10-30 minutes
test_on_borrow	Verify connection works before use
	Slight performance cost but safer

Sizing Formula:

```
Max Pool Size = (Number of CPU cores × 2) + Number of disks

Example server: 8 cores, 2 disks
Max pool size = (8 × 2) + 2 = 18 connections

Why?
- CPU-bound queries: 2× cores (can context switch)
- Disk I/O: +1 per disk (parallel I/O)

Don't make pool too large!
- Too many connections waste memory
- Database has connection limit
- Context switching overhead
```

Common Pitfalls

1. Connection Leak

```python
# ❌ BAD: Connection never returned!
def bad_query(user_id):
    conn = pool.getconn()
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users WHERE id = %s", (user_id,))
    return cursor.fetchone()
    # Connection not returned! Pool will run out!

# ✓ GOOD: Always use try/finally
def good_query(user_id):
    conn = pool.getconn()
    try:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM users WHERE id = %s", (user_id,))
        return cursor.fetchone()
    finally:
        cursor.close()
        pool.putconn(conn)  # Always returns connection
```

2. Pool Exhaustion

```python
```

```
# Problem: More concurrent requests than pool size
Pool size: 10 connections
Concurrent requests: 50

Result:
- 10 requests get connections immediately
- 40 requests wait
- After timeout, requests fail!

Solution 1: Increase pool size
pool_size = 50

Solution 2: Add queue with timeout
Solution 3: Scale horizontally (more app servers)
```

3. Long-Running Transactions

python

```
# ❌ BAD: Holds connection for 10 minutes!
conn = pool.getconn()
cursor = conn.cursor()
cursor.execute("BEGIN")
cursor.execute("SELECT * FROM orders FOR UPDATE")

time.sleep(600)  # Process for 10 minutes

cursor.execute("COMMIT")
pool.putconn(conn)

# This connection is unavailable for 10 minutes!
# Other requests timeout waiting for connections

# ✓ GOOD: Keep transactions short
conn = pool.getconn()
cursor = conn.cursor()

# Do heavy computation OUTSIDE transaction
data = expensive_computation()

# Quick transaction
cursor.execute("BEGIN")
cursor.execute("UPDATE orders SET processed = true WHERE id = %s", (order_id,))
cursor.execute("COMMIT")

pool.putconn(conn)
```

Monitoring Connection Pools

python

```
from sqlalchemy import event, create_engine

engine = create_engine('postgresql://...')

@event.listens_for(engine, "connect")
def receive_connect(dbapi_conn, connection_record):
    print("New connection established")

@event.listens_for(engine, "checkout")
def receive_checkout(dbapi_conn, connection_record, connection_proxy):
    print("Connection checked out from pool")

@event.listens_for(engine, "checkin")
def receive_checkin(dbapi_conn, connection_record):
    print("Connection returned to pool")

# Get pool statistics
def print_pool_stats():
    pool = engine.pool
    print(f"Pool size: {pool.size()}")
    print(f"Connections in use: {pool.checkedout()}")
    print(f"Connections available: {pool.size() - pool.checkedout()}")
    print(f"Overflow: {pool.overflow()}")
```

Key Takeaways

1. **SQL vs NoSQL:**
    - SQL: Structured, ACID, complex queries, relations
    - NoSQL: Flexible, scalable, simple lookups, various models
    - Use both (polyglot persistence) in modern apps
2. **ACID Properties:**
    - Atomicity: All or nothing
    - Consistency: Valid state always
    - Isolation: Transactions don't interfere
    - Durability: Changes survive crashes
3. **Indexes:**
    - B-Tree: Default, good for ranges
    - Hash: Fast exact matches only

    - Composite: Multi-column indexes
    - Trade-off: Fast reads, slower writes

4. **Normalization:**
    - Minimize redundancy
    - Use for write-heavy OLTP
    - Denormalize for read-heavy scenarios

5. **Connection Pooling:**
    - Reuse connections
    - 10-20 connections typical
    - Always return connections
    - Monitor pool health

Practice Problems

1. Design a database schema for Twitter (users, tweets, follows, likes). Should it be normalized or denormalized? Why?
2. A query takes 5 seconds on a table with 1 million rows. You add an index and it takes 50ms. The table gets 10,000 inserts per day. Calculate the performance impact.
3. Your connection pool has 20 connections but you're getting timeout errors. Diagnose and solve the problem.

Next Chapter Preview

In Chapter 7, we'll explore **Database Scaling**:

- Read replicas
- Master-slave replication
- Database sharding
- Partitioning strategies
- Handling distributed data

Ready to continue?