# Database Fundamentals

Welcome to the essential guide for understanding database systems. Whether you're building your first application or expanding your technical knowledge, mastering databases is crucial for modern software development.

## What is a database?

A **database** is an organised system that stores data in a structured manner, enabling you to reliably read, write, query, and manage information. Relational database management systems (RDBMS) organise data into **tables** containing rows and columns, and they support Structured Query Language (SQL) for data manipulation and retrieval.

## Core concepts

### Table

A named collection of rows (records) and columns (fields) that structures related data

### Row & Column

Rows represent individual records (tuples), whilst columns define attributes with specific names and data types

### Primary Key (PK)

A unique identifier for each row, ensuring no duplicates and enabling fast lookups

### Foreign Key (FK)

A column that references the primary key of another table, enforcing relational integrity

### Index

A data structure that dramatically speeds up data retrieval operations by creating efficient lookup paths

### Transaction

A group of database operations that execute together, following ACID properties: they either all commit successfully or rollback completely

# Sample Schema: Our Learning Database

Throughout this guide, we'll work with a practical e-commerce schema featuring customers, products, orders, order items, and employees. This real-world example will help you understand how databases model business relationships and data structures.

## Schema Structure

Our database consists of five interconnected tables that demonstrate common relationship patterns. The **customers** table stores user information, whilst **products** maintains our inventory. The **orders** table links customers to their purchases, and **order_items** creates a many-to-many relationship between orders and products. Finally, the **employees** table demonstrates self-referencing relationships through a manager hierarchy.

```
-- Customers
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(150) UNIQUE,
    signup_date DATE
);

-- Products
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL(10,2)
);

-- Orders
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date DATE,
    total_amount DECIMAL(10,2),
    status VARCHAR(20),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

-- Order items (many-to-many)
CREATE TABLE order_items (
    order_item_id SERIAL PRIMARY KEY,
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    qty INT NOT NULL,
    unit_price DECIMAL(10,2),
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);

-- Employees (self-referencing)
CREATE TABLE employees (
    emp_id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    manager_id INT,
    salary DECIMAL(10,2),
    FOREIGN KEY (manager_id) REFERENCES employees(emp_id)
);
```

## Sample Data

Let's populate our schema with representative data that we'll query throughout the examples. This includes three customers, four products, three orders with multiple items, and a four-person employee hierarchy.

```
-- Sample customers
INSERT INTO customers (name, email, signup_date) VALUES
 ('Alice', 'alice@example.com', '2023-01-10'),
 ('Bob', 'bob@example.com', '2023-02-05'),
 ('Cathy', 'cathy@example.com', '2023-03-01');

-- Sample products
INSERT INTO products (name, category, price) VALUES
 ('T-shirt', 'Apparel', 19.99),
 ('Running Shoe', 'Footwear', 89.95),
 ('Socks', 'Apparel', 5.00),
 ('Water Bottle', 'Accessories', 12.00);

-- Sample orders
INSERT INTO orders (customer_id, order_date, total_amount, status) VALUES
 (1, '2023-05-01', 39.98, 'completed'),
 (2, '2023-05-02', 101.95, 'completed'),
 (1, '2023-05-10', 12.00, 'shipped');

-- Order line items
INSERT INTO order_items (order_id, product_id, qty, unit_price) VALUES
 (1, 1, 2, 19.99),
 (2, 2, 1, 89.95),
 (2, 3, 3, 5.00),
 (3, 4, 1, 12.00);

-- Employee hierarchy
INSERT INTO employees (name, manager_id, salary) VALUES
 ('CEO', NULL, 200000),
 ('Alice Mgr', 1, 150000),
 ('Developer', 2, 90000),
 ('Tester', 2, 70000);
```

# Database Modelling: Designing Tables

Effective table design is the foundation of a robust database. Thoughtful planning during the design phase prevents costly restructuring later and ensures your database performs efficiently as it scales.

## Designing a table

01

### Choose meaningful column names

Use clear, descriptive names that convey purpose. Prefer `customer_email` over `email1`

02

### Select appropriate data types

Match types to data: VARCHAR for text, INT for whole numbers, DECIMAL for currency, DATE for dates

03

### Define primary keys

Every table needs a unique identifier. Use SERIAL or AUTO_INCREMENT for automatic generation

04

### Specify NOT NULL constraints

Mark essential columns as NOT NULL to enforce data integrity and prevent incomplete records

### Example: Products table design rationale

**Design decision:** `product_id` as SERIAL PRIMARY KEY

**Rationale:** Provides unique, auto-incrementing identifier for each product

**Design decision:** `name` as VARCHAR(100)

**Rationale:** Variable-length text accommodates product names efficiently without wasting space

**Design decision:** `category` as VARCHAR(50)

**Rationale:** Shorter field for classification; consider an enum or separate category table for large catalogues

**Design decision:** `price` as DECIMAL(10,2)

**Rationale:** Fixed precision avoids floating-point errors in financial calculations

## Normalisation basics

Normalisation eliminates data redundancy and improves consistency by organising data into well-structured tables. The first three normal forms cover most practical needs.

1

### First Normal Form (1NF)

Each column contains atomic (indivisible) values. No repeating groups or arrays in a single column

2

### Second Normal Form (2NF)

Meets 1NF and eliminates partial dependencies. All non-key columns depend on the entire primary key (relevant for composite keys)

3

### Third Normal Form (3NF)

Meets 2NF and eliminates transitive dependencies. Non-key columns depend only on the primary key, not on other non-key columns

### Normalisation in action

Suppose you initially designed `orders(order_id, customer_name, customer_email, order_date)`. This duplicates customer information with every order. Following 3NF principles, we split this into `customers` and `orders` tables, storing customer details once and referencing them via `customer_id` foreign key.

# Database Modelling: Relationships

Relationships define how tables connect and interact within your database. Understanding these patterns is essential for modelling real-world business logic accurately.

## Relationship types

### One-to-One

Rare in practice. Example: `users` and `user_profiles` where each user has exactly one profile

**Implementation:** Add a unique foreign key in one table referencing the other

### One-to-Many

Most common relationship. Example: one customer places many orders

**Implementation:** Place the foreign key (`customer_id`) in the "many" table (`orders`)

### Many-to-Many

Complex relationships requiring a junction table. Example: orders contain multiple products, products appear in multiple orders

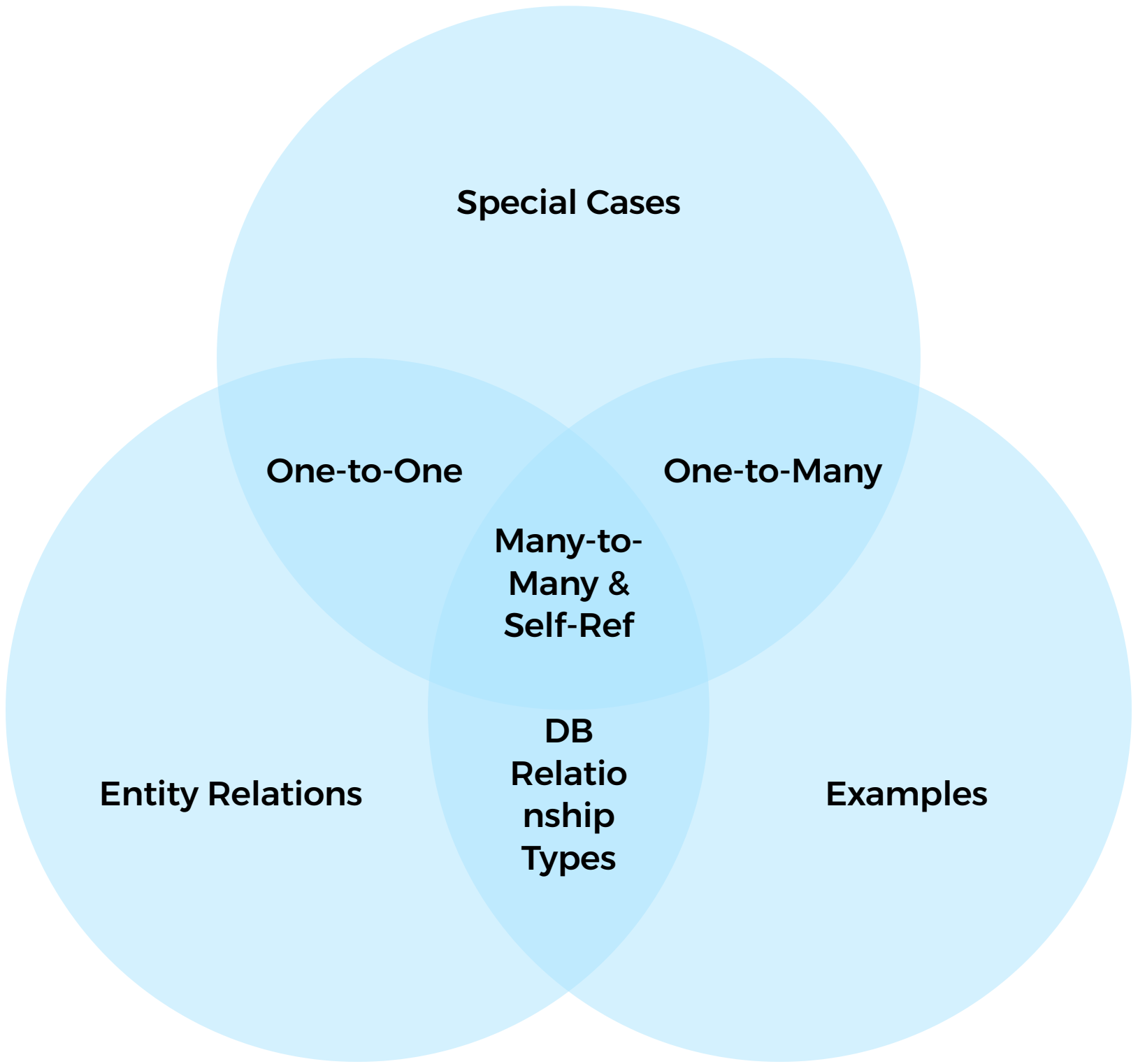
**Implementation:** Create a join table (`order_items`) containing foreign keys to both related tables, plus any relationship-specific attributes like quantity

### Self-Referencing

A table references itself. Example: employee-manager hierarchy

**Implementation:** Add a nullable foreign key (`manager_id`) referencing the same table's primary key (`emp_id`)

## Relationship visualisation



# Many-to-Many Relationships in Practice

Many-to-many relationships require special handling through junction tables (also called join tables or bridge tables). These intermediate tables resolve the complexity whilst allowing you to store additional relationship-specific data.

## The order_items junction table

In our e-commerce schema, `order_items` serves as the junction table between `orders` and `products`. It contains foreign keys to both tables, plus additional attributes like `qty` and `unit_price` that describe the specific relationship.

## Querying across many-to-many relationships

To retrieve data spanning a many-to-many relationship, you join through the junction table. This example shows how to list all order details including customer names and product information.

```
-- Query across the many-to-many relationship
SELECT
    o.order_id,
    c.name AS customer,
    p.name AS product,
    oi.qty,
    oi.unit_price,
    (oi.qty * oi.unit_price) AS line_total
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN order_items oi ON oi.order_id = o.order_id
JOIN products p ON oi.product_id = p.product_id
ORDER BY o.order_id, p.name;
```

### Why junction tables matter

Without the `order_items` junction table, you couldn't properly model the fact that one order contains multiple products, each with its own quantity and price. Junction tables transform complex many-to-many relationships into two manageable one-to-many relationships.

# Database Optimisation: Schema Design

Optimisation begins at the schema level, long before you write queries. Strategic design decisions about indexes, data types, and table structure profoundly impact database performance as your data grows.

## Schema-level optimisation strategies

### Indexes: Strategic speed boosts

Create indexes on columns frequently used in WHERE clauses, JOIN conditions, and ORDER BY statements. Indexes dramatically reduce scan times by creating efficient lookup paths.

```
CREATE INDEX
idx_orders_customer_date
    ON orders(customer_id,
order_date);
```

### Primary and foreign key constraints

Beyond enforcing referential integrity, primary keys often create clustered indexes (database-dependent), whilst foreign keys enable the query optimiser to make intelligent decisions about join strategies.

### Appropriate data types

Choosing smaller data types reduces memory usage and I/O costs. Use INT instead of BIGINT when values won't exceed 2 billion. Select VARCHAR lengths that match actual data patterns rather than defaulting to maximum sizes.

### Table partitioning (advanced)

For very large tables, partition by date range, hash, or list values. Partitioning reduces scanning costs by allowing the database to skip irrelevant partitions entirely during queries.

# Database Optimisation: Query Tuning

Once your schema is well-designed, query-level optimisation ensures individual operations execute efficiently. Understanding how databases execute queries empowers you to write faster, more efficient SQL.

## Query-level optimisation techniques

### Write sargable conditions

Sargable (Search ARGument ABLE) conditions allow indexes to be used. Write `WHERE customer_id = 1` instead of `WHERE UPPER(customer_id) = 1`

### Avoid SELECT * in production

Retrieve only the columns you need. This reduces network transfer, memory usage, and I/O operations

### Use EXPLAIN to understand execution

View query plans with EXPLAIN or EXPLAIN ANALYZE to see which indexes are used and identify performance bottlenecks

### Example: The power of indexes

#### Without index

```
-- Full table scan
SELECT * FROM orders
WHERE customer_id = 1;
```

Database must examine every row in the orders table to find matches

#### With index

```
CREATE INDEX idx_orders_customer
   ON orders(customer_id);

SELECT * FROM orders
WHERE customer_id = 1;
```

Database uses index to jump directly to relevant rows, potentially reducing scan time from seconds to milliseconds

## Understanding query execution plans

```
EXPLAIN SELECT * FROM orders WHERE customer_id = 1;

-- Sample output:
-- Index Scan using idx_orders_customer on orders
-- Index Cond: (customer_id = 1)
-- Cost: 0.15..8.17 rows=3
```

## Advanced indexing strategies

### Composite indexes

Index multiple columns together when queries filter on both. Order matters: `(customer_id, order_date)` helps queries filtering by customer_id first, then date

### Covering indexes

Include all columns needed by a query in the index, allowing the database to satisfy the query entirely from the index without accessing the table

```
CREATE INDEX idx_orders_cover
ON orders(customer_id, order_date,
status);
```

### Strategic denormalisation

In read-heavy systems with expensive joins, consider storing redundant data. For example, snapshot customer names in orders. This trades update complexity and storage space for faster reads

# Creating Basic SQL Queries

Mastering fundamental SQL operations forms the foundation for all database work. These essential patterns appear in virtually every database application you'll build.

## Core SELECT operations

### Basic retrieval

```
SELECT name, email
FROM customers;
```

Fetch specific columns from a table. Always specify column names rather than using SELECT * in production code

### Filtering with WHERE

```
SELECT * FROM orders
WHERE status = 'completed';

SELECT * FROM orders
WHERE order_date >= '2023-05-01'
  AND order_date < '2023-06-01';
```

Filter rows based on conditions using comparison operators and logical AND/OR

## Sorting and limiting results

```
-- Get the 5 most expensive products
SELECT name, category, price
FROM products
ORDER BY price DESC
LIMIT 5;
```

ORDER BY sorts results (use DESC for descending, ASC for ascending). LIMIT restricts the number of rows returned, useful for pagination or "top N" queries.

## Data manipulation statements

### INSERT

```
INSERT INTO products
    (name, category, price)
VALUES
    ('Cap', 'Apparel', 8.50);
```

### UPDATE

```
UPDATE products
SET price = price * 0.9
WHERE category = 'Apparel';
```

### DELETE

```
DELETE FROM orders
WHERE order_date < '2022-01-01'
  AND status = 'cancelled';
```

**Always include WHERE with UPDATE and DELETE**

Without a WHERE clause, UPDATE and DELETE affect *every row* in the table. Always double-check your WHERE conditions before executing these statements in production databases.

## Calculated columns and aliases

### Expressions in SELECT

```
SELECT
    name,
    price,
    price * 0.9 AS discounted_price
FROM products;
```

Perform calculations in your SELECT clause. Use AS to give calculated columns meaningful names

### Table aliases for readability

```
SELECT
    c.name AS customer_name,
    o.order_id,
    o.total_amount
FROM customers c
JOIN orders o
    ON c.customer_id = o.customer_id;
```

Aliases make queries more readable, especially with multiple tables. Common convention: use short, lowercase abbreviations

# Understanding SQL Joins

Joins are the mechanism for combining data from multiple tables, enabling you to reconstruct the relationships you modelled in your schema. Different join types serve distinct purposes in data retrieval.

## Join types overview

We'll explore six essential join patterns: INNER, LEFT, RIGHT, FULL OUTER, CROSS, and SELF joins. Each serves specific use cases in relating table data.

### 1. INNER JOIN: Matching rows only

Returns only rows where the join condition finds matches in both tables. This is the most common join type, used when you need data that exists in both related tables.

```
SELECT
    o.order_id,
    c.name AS customer,
    o.total_amount
FROM orders o
INNER JOIN customers c
    ON o.customer_id = c.customer_id;
```

**Result:** Only orders that have a matching customer in the customers table

**Use when:** You need complete, related data from both tables

# LEFT JOIN: Including unmatched rows

LEFT JOIN (or LEFT OUTER JOIN) returns all rows from the left table plus matching rows from the right table. When no match exists, the right table's columns contain NULL values.

## Practical example: All customers and their orders

This query lists every customer, including those who haven't placed any orders yet. Customers without orders will have NULL values in the order-related columns.

```
SELECT
    c.customer_id,
    c.name,
    o.order_id,
    o.total_amount
FROM customers c
LEFT JOIN orders o
    ON c.customer_id = o.customer_id
ORDER BY c.customer_id;
```

| 3 | 5 | 8 |
| --- | --- | --- |
| **Total customers** | **Total orders** | **Result rows** |
| All customer records returned | Matched order records | Customers × their orders (customers with no orders appear once with NULL order_id) |

### When to use LEFT JOIN

LEFT JOIN is invaluable for finding "orphan" records or creating reports that must include all primary entities regardless of related data. Common use cases include customer lists showing purchase history (or lack thereof) and product catalogues indicating which items have sold.

# RIGHT JOIN: The mirror of LEFT JOIN

RIGHT JOIN (or RIGHT OUTER JOIN) is the mirror image of LEFT JOIN—it returns all rows from the right table plus matching rows from the left table. In practice, most developers prefer LEFT JOIN and simply swap table positions for better readability.

## RIGHT JOIN syntax

```
SELECT
    o.order_id,
    o.total_amount,
    c.name
FROM orders o
RIGHT JOIN customers c
    ON o.customer_id = c.customer_id;
```

## Equivalent LEFT JOIN

```
SELECT
    o.order_id,
    o.total_amount,
    c.name
FROM customers c
LEFT JOIN orders o
    ON c.customer_id = o.customer_id;
```

Both queries produce identical results: all customers with their orders (or NULL for customers without orders). The LEFT JOIN version is generally preferred for clarity because the primary entity (customers) appears first in the FROM clause.

### RIGHT JOIN in practice

RIGHT JOIN is supported in MySQL and PostgreSQL but is used less frequently than LEFT JOIN. Many developers find it more intuitive to place the "complete" table first and use LEFT JOIN, making queries easier to understand when returning to code months later.

# FULL OUTER JOIN: Everything from both sides

FULL OUTER JOIN returns all rows from both tables, matching them where possible and filling unmatched columns with NULL. This join type shows the complete picture from both tables, highlighting both matches and gaps.

## FULL OUTER JOIN syntax

Supported natively in PostgreSQL, FULL OUTER JOIN reveals which customers have no orders and which orders lack customer records (perhaps due to data integrity issues or soft-deleted customers).

```
-- PostgreSQL syntax
SELECT
    c.customer_id,
    c.name,
    o.order_id,
    o.total_amount
FROM customers c
FULL OUTER JOIN orders o
    ON c.customer_id = o.customer_id;
```

## MySQL workaround using UNION

MySQL doesn't support FULL OUTER JOIN directly, but you can emulate it by combining LEFT and RIGHT joins with UNION.

```
-- MySQL equivalent
SELECT c.customer_id, c.name, o.order_id, o.total_amount
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id

UNION

SELECT c.customer_id, c.name, o.order_id, o.total_amount
FROM customers c
RIGHT JOIN orders o ON c.customer_id = o.customer_id
WHERE c.customer_id IS NULL;
```

### When to use FULL OUTER JOIN

FULL OUTER JOIN is valuable for data quality audits and reconciliation tasks. Use it to identify orphaned records on either side of a relationship, detect referential integrity issues, or create comprehensive reports that must account for all data regardless of relationships.

# CROSS JOIN: Cartesian products

CROSS JOIN produces a Cartesian product—every row from the first table paired with every row from the second table. This creates a result set with rows equal to the product of both table sizes. Whilst rarely used unintentionally, it has legitimate applications.

## CROSS JOIN syntax and behaviour

```
SELECT
    p.name AS product,
    c.name AS customer
FROM products p
CROSS JOIN customers c
LIMIT 10;
```

| 4 | 3 | 12 |
| --- | --- | --- |
| **Products** | **Customers** | **Result rows** |
| Number of rows in products table | Number of rows in customers table | Total combinations (4 × 3) |

## Legitimate uses for CROSS JOIN

### Generating test data

Create comprehensive test scenarios by combining every configuration option

### Building reporting matrices

Generate rows for every date × product combination, then LEFT JOIN actual sales data

### Calendar and scheduling queries

Pair employees with time slots to build shift schedules

**Performance warning**

CROSS JOIN result sets grow multiplicatively and can quickly become enormous. A CROSS JOIN between two 1,000-row tables produces 1,000,000 rows. Always use LIMIT during development and ensure you genuinely need a Cartesian product.

# SELF-JOIN: Tables relating to themselves

A self-join joins a table to itself, enabling you to model hierarchical or networked relationships within a single table. This powerful technique is essential for representing organisational structures, category trees, and social network connections.

## Employee-manager relationships

Our employees table uses `manager_id` to reference another employee's `emp_id`. A self-join reveals these relationships by treating the table as two separate entities: employees and managers.

```
SELECT
    e.name AS employee,
    m.name AS manager,
    e.salary AS employee_salary
FROM employees e
LEFT JOIN employees m
    ON e.manager_id = m.emp_id
ORDER BY m.name, e.name;
```

| Employee | Manager | Salary |
| --- | --- | --- |
| CEO | NULL (top of hierarchy) | £200,000 |
| Alice Mgr | CEO | £150,000 |
| Developer | Alice Mgr | £90,000 |
| Tester | Alice Mgr | £70,000 |

Note the LEFT JOIN ensures the CEO appears even though they have no manager (manager_id = NULL). An INNER JOIN would exclude top-level records.

# SELF-JOIN: Advanced patterns

Self-joins enable sophisticated queries that reveal relationships within hierarchical data. Beyond basic manager lookups, you can find peers, detect patterns, and analyse network structures.

## Finding employees with the same manager

This query identifies colleagues—employees who share the same manager. It's useful for team analysis, peer grouping, and organisational planning.

```
SELECT
    a.name AS employee_a,
    b.name AS employee_b,
    m.name AS shared_manager
FROM employees a
JOIN employees b
    ON a.manager_id = b.manager_id
JOIN employees m
    ON a.manager_id = m.emp_id
WHERE a.emp_id < b.emp_id  -- Avoid duplicate pairs
ORDER BY m.name, a.name;
```

## Result analysis

This query returns pairs of employees under the same manager. The condition `a.emp_id < b.emp_id` ensures each pair appears only once (preventing both "Developer, Tester" and "Tester, Developer" from appearing).

### Team identification

Quickly identify who works together under the same leadership

### Peer relationships

Analyse reporting structures and team dynamics

### Hierarchy analysis

Understand organisational depth and span of control

**Self-join use cases**

Self-joins appear in many domains: product recommendations (customers who bought X also bought Y), forum threads (posts and replies), category hierarchies (parent/child categories), transportation routes (connecting cities), and social networks (followers and following).

# Aggregate Functions and GROUP BY

Aggregate functions summarise data across multiple rows, transforming detailed records into meaningful statistics. Combined with GROUP BY, they power business intelligence queries, reports, and analytics dashboards.

## Core aggregate functions

### COUNT()

Counts rows or non-NULL values

```
COUNT(*)
COUNT(column_name)
```

### SUM()

Totals numeric values

```
SUM(total_amount)
SUM(qty * price)
```

### AVG()

Calculates arithmetic mean

```
AVG(salary)
AVG(order_value)
```

### MIN() / MAX()

Finds smallest/largest values

```
MIN(signup_date)
MAX(price)
```

## Practical example: Customer spending analysis

This query reveals purchasing patterns by calculating order count and total spend per customer. Notice how GROUP BY enables per-customer aggregation whilst LEFT JOIN includes customers with zero orders.

```
SELECT
    c.customer_id,
    c.name,
    COUNT(o.order_id) AS orders_count,
    COALESCE(SUM(o.total_amount), 0) AS total_spent
FROM customers c
LEFT JOIN orders o
    ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.name
ORDER BY total_spent DESC;
```

### Understanding COALESCE

COALESCE returns the first non-NULL value in its arguments. Here, `COALESCE(SUM(o.total_amount), 0)` converts NULL (for customers with no orders) to 0, preventing NULL in your report totals.

# Filtering Aggregates: HAVING vs WHERE

Understanding the distinction between WHERE and HAVING is crucial for writing correct aggregate queries. They filter at different stages of query execution, serving complementary purposes.

## WHERE vs HAVING: Execution order


### WHERE clause

**Filters individual rows** before grouping occurs. Use WHERE to limit which rows participate in aggregation.

```
-- Only aggregate completed orders
SELECT customer_id, SUM(total_amount)
FROM orders
WHERE status = 'completed'
GROUP BY customer_id;
```

### HAVING clause

**Filters aggregated groups** after grouping. Use HAVING to filter based on aggregate results.

```
-- Only show customers with 2+ orders
SELECT customer_id, COUNT(*) as order_count
FROM orders
GROUP BY customer_id
HAVING COUNT(*) >= 2;
```

## Practical example: High-value customers

This query identifies customers who have placed multiple orders by filtering groups with HAVING. You cannot use WHERE for this because order count only exists after grouping.

```
SELECT
    c.name,
    COUNT(o.order_id) AS orders_count,
    SUM(o.total_amount) AS total_spent
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.name
HAVING COUNT(o.order_id) > 1
ORDER BY total_spent DESC;
```

### Combining WHERE and HAVING

You can use both in the same query: WHERE filters rows before aggregation (improving performance), whilst HAVING filters the aggregated results. For example, calculate total completed orders per customer, then show only customers with totals exceeding £100.

# Window Functions: Advanced Analytics

Window functions perform calculations across sets of rows related to the current row, but unlike aggregate functions with GROUP BY, they don't collapse rows. This preserves individual row details whilst adding analytical context—the best of both worlds.

## Window functions vs GROUP BY

### Traditional GROUP BY

```
SELECT
    category,
    AVG(price) as avg_price
FROM products
GROUP BY category;
```

**Result:** One row per category with the average

### Window function

```
SELECT
    name,
    category,
    price,
    AVG(price) OVER (PARTITION BY category) as avg_price
FROM products;
```

**Result:** Every product row, plus its category's average

## Ranking customers by spending

This query assigns rankings to customers based on total spending. Notice how the subquery aggregates spending first, then the outer query applies RANK() as a window function.

```
SELECT
    customer_id,
    name,
    total_spent,
    RANK() OVER (ORDER BY total_spent DESC) AS spending_rank,
    DENSE_RANK() OVER (ORDER BY total_spent DESC) AS dense_rank
FROM (
    SELECT
        c.customer_id,
        c.name,
        COALESCE(SUM(o.total_amount), 0) AS total_spent
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.customer_id, c.name
) t
ORDER BY spending_rank;
```

## Common window functions

### ROW_NUMBER()

Assigns unique sequential integers. Ties get different numbers

### RANK()

Assigns ranks with gaps after ties. Two #1 ranks followed by #3

### DENSE_RANK()

Assigns ranks without gaps. Two #1 ranks followed by #2

### NTILE(n)

Divides rows into n equal buckets for quartile/percentile analysis

# String, Date, and Numeric Functions

SQL provides rich libraries of built-in functions for transforming and manipulating data. These functions enable data cleaning, formatting, and calculation without requiring application-side processing.

## String manipulation functions

### UPPER() / LOWER()

Convert text case

```
SELECT UPPER(name)
FROM customers;
```

### TRIM() / LTRIM() / RTRIM()

Remove whitespace

```
SELECT TRIM(email)
FROM customers;
```

### SUBSTRING() / SUBSTR()

Extract portions of strings

```
SELECT SUBSTRING(name, 1, 3)
FROM products;
```

### CONCAT() / ||

Join strings together

```
SELECT name || ' - ' || category
FROM products;
```

## Date and time functions

Date functions vary significantly between database systems. Here are examples for PostgreSQL and MySQL.

### PostgreSQL date functions

```
-- Extract components
SELECT EXTRACT(YEAR FROM order_date)
FROM orders;

-- Truncate to start of period
SELECT DATE_TRUNC('month', order_date)
FROM orders;

-- Date arithmetic
SELECT order_date + INTERVAL '30 days'
FROM orders;
```

### MySQL date functions

```
-- Format dates
SELECT DATE_FORMAT(order_date, '%Y-%m')
FROM orders;

-- Extract components
SELECT YEAR(order_date), MONTH(order_date)
FROM orders;

-- Date arithmetic
SELECT DATE_ADD(order_date, INTERVAL 30 DAY)
FROM orders;
```

## Numeric functions

### ROUND(value, decimals)

Rounds to specified decimal places: `ROUND(19.996, 2)` returns 20.00

### CEIL() / CEILING()

Rounds up to nearest integer: `CEIL(19.1)` returns 20

### FLOOR()

Rounds down to nearest integer: `FLOOR(19.9)` returns 19

### ABS()

Returns absolute value: `ABS(-15.5)` returns 15.5

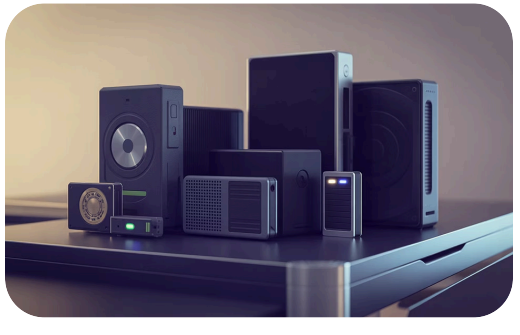
## Practical example: Monthly sales aggregation

```
SELECT
 EXTRACT(MONTH FROM order_date) AS month,
 EXTRACT(YEAR FROM order_date) AS year,
 COUNT(*) AS order_count,
 SUM(total_amount) AS monthly_total,
 ROUND(AVG(total_amount), 2) AS avg_order_value
FROM orders
GROUP BY EXTRACT(YEAR FROM order_date), EXTRACT(MONTH FROM order_date)
ORDER BY year, month;
```

# Data Import and Export Overview

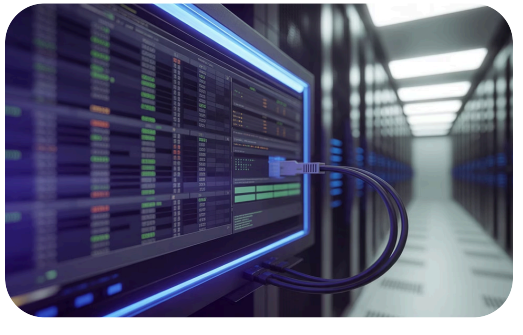
Moving data in and out of databases is a fundamental operational task. Whether you're migrating data, creating backups, loading initial datasets, or integrating with external systems, understanding import and export mechanisms is essential.

## Common scenarios for data transfer



### Backup and recovery

Regular database dumps protect against data loss from hardware failures, corruption, or human error

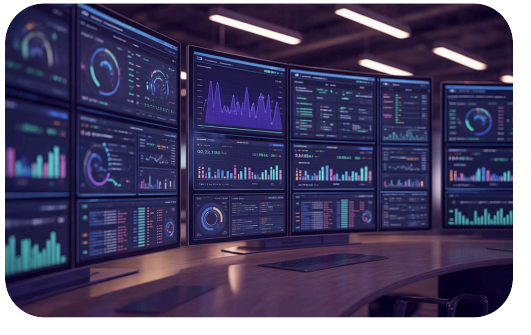


### Data migration

Moving data between systems during platform changes, upgrades, or consolidations

### Analysis and reporting

Exporting datasets for analysis in spreadsheets, BI tools, or data science platforms



### System integration

Exchanging data between applications via CSV, JSON, or other interchange formats

## Methods overview by database system

PostgreSQL and MySQL offer command-line utilities, SQL commands, and GUI tools for data transfer. The following cards explore each in detail.

# Exporting Data: PostgreSQL

PostgreSQL provides powerful export capabilities through `pg_dump` for complete database backups and `COPY` for selective data exports to CSV or other formats.

## Full database backup with pg_dump

The `pg_dump` utility creates complete, consistent snapshots of your database. This is the recommended method for backups and cross-system migrations.

```
# Full database backup in custom compressed format
pg_dump -U username -h hostname -F c -b -v -f backup_file.dump dbname

# Plain SQL format (readable, larger files)
pg_dump -U username -h hostname -f backup.sql dbname

# Backup specific tables only
pg_dump -U username -h hostname -t customers -t orders -f partial_backup.sql dbname
```

### pg_dump flags explained

- `-U username` — Database user to connect as
- `-h hostname` — Database server hostname or IP
- `-F c` — Output format (c=custom, p=plain, t=tar)
- `-b` — Include large objects (BLOBs)
- `-v` — Verbose mode (show progress)
- `-f filename` — Output file path

## Exporting tables to CSV with COPY

### Server-side COPY

```
-- Requires server filesystem access
COPY (SELECT * FROM orders)
TO '/tmp/orders.csv'
WITH CSV HEADER;
```

Writes directly on the database server. Fast but requires appropriate permissions and file system access.

### Client-side \copy (psql)

```
-- Writes to local machine
\copy (SELECT * FROM orders)
TO 'orders.csv'
WITH CSV HEADER
```

Executed through psql client. Writes to your local filesystem, more flexible for remote databases.

**COPY options**

Customise delimiters with `DELIMITER ','`, handle NULLs with `NULL 'NULL'`, and control quoting with `QUOTE '"'`. Include column headers with `HEADER` for easier imports into spreadsheets.

# Exporting Data: MySQL

MySQL provides `mysqldump` for comprehensive backups and `SELECT ... INTO OUTFILE` for CSV exports. These tools integrate well with automated backup scripts and ETL pipelines.

## Complete database backup with mysqldump

```
# Backup entire database
mysqldump -u username -p dbname > backup.sql

# Backup specific tables
mysqldump -u username -p dbname customers orders > partial_backup.sql

# Include routines, triggers, and events
mysqldump -u username -p --routines --triggers --events dbname > full_backup.sql

# Backup all databases
mysqldump -u username -p --all-databases > all_databases.sql
```

### mysqldump best practices

**Add timestamps to backup filenames**

Use `backup_$(date +%Y%m%d).sql` for easy chronological organisation

**Compress large backups**

Pipe through gzip: `mysqldump ... | gzip > backup.sql.gz`

**Use --single-transaction for InnoDB**

Ensures consistent backups without locking tables: `--single-transaction`

## Exporting tables to CSV

MySQL's `SELECT ... INTO OUTFILE` writes query results directly to CSV files on the server filesystem.

```
SELECT
    order_id,
    customer_id,
    order_date,
    total_amount
FROM orders
INTO OUTFILE '/tmp/orders.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```

**Permissions and secure_file_priv**

MySQL's `secure_file_priv` setting restricts where files can be written. Check this with `SHOW VARIABLES LIKE 'secure_file_priv';` and ensure your target directory is permitted, or use GUI tools like MySQL Workbench for exports to arbitrary locations.

# Importing Data: PostgreSQL and MySQL

Importing data efficiently requires understanding your database system's bulk loading capabilities. Both PostgreSQL and MySQL offer high-performance import mechanisms that bypass overhead of individual INSERT statements.

## PostgreSQL: COPY and \copy

### Server-side COPY

```
-- From server filesystem
COPY products (name, category, price)
FROM '/tmp/products.csv'
CSV HEADER;
```

Fast bulk loading directly on the server. Requires file access on database server.

### Client-side \copy (psql)

```
-- From local filesystem
\copy products (name, category, price)
FROM 'products.csv'
CSV HEADER;
```

Loads from your local machine through psql. More flexible for remote databases.

## MySQL: LOAD DATA INFILE

```
LOAD DATA INFILE '/tmp/products.csv'
INTO TABLE products
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES  -- Skip header row
(name, category, price);
```

### Local file loading in MySQL

For files on your client machine (not the server), use `LOAD DATA LOCAL INFILE`. This requires `local_infile=1` to be enabled in your MySQL configuration.

```
LOAD DATA LOCAL INFILE 'C:/data/products.csv'
INTO TABLE products
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
IGNORE 1 LINES;
```

## Alternative: SQL INSERT statements

For smaller datasets or when bulk loading tools are unavailable, generate INSERT statements from your CSV data. Many tools and scripts can automate this conversion.

```
INSERT INTO products (name, category, price) VALUES
    ('T-shirt', 'Apparel', 19.99),
    ('Running Shoe', 'Footwear', 89.95),
    ('Water Bottle', 'Accessories', 12.00);
```

## GUI and ETL tools



### Database clients

DBeaver, pgAdmin, MySQL Workbench offer visual import/export wizards with column mapping and validation



### ETL platforms

Apache Airflow, Talend, SSIS provide scheduled, monitored, and logged data pipelines with transformation capabilities

# Hands-On Examples: Real-World Queries

Let's apply our knowledge to practical business questions. These examples demonstrate techniques you'll use regularly in application development and data analysis.

## Example 1: Identify best-selling product by quantity

This query aggregates order items to find which product has sold the most units—critical for inventory planning and marketing focus.

```
SELECT
    p.product_id,
    p.name,
    p.category,
    SUM(oi.qty) AS total_qty_sold
FROM products p
JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY p.product_id, p.name, p.category
ORDER BY total_qty_sold DESC
LIMIT 1;
```

### Query breakdown

- JOIN connects products to order_items
- SUM aggregates quantities across all orders
- GROUP BY creates per-product totals
- ORDER BY DESC sorts highest first
- LIMIT 1 returns only the top result

### Extension ideas

**Top 5 products:** Change LIMIT 1 to LIMIT 5

**By category:** Add PARTITION BY category to find top seller per category

**By revenue:** Change SUM(oi.qty) to SUM(oi.qty * oi.unit_price)

# Finding Customers Without Orders

Identifying inactive customers helps target re-engagement campaigns and understand user retention. This query demonstrates the power of LEFT JOIN combined with WHERE to find missing relationships.

## Example 2: Customers who haven't placed orders

```
SELECT
    c.customer_id,
    c.name,
    c.email,
    c.signup_date,
    CURRENT_DATE - c.signup_date AS days_since_signup
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE o.order_id IS NULL
ORDER BY c.signup_date;
```

## How this query works

1

### LEFT JOIN

Includes all customers even if they have no matching orders

2

### WHERE o.order_id IS NULL

Filters to only customers where the join found no matches

3

### Result

List of customers who registered but never purchased

## Business applications

### Marketing campaigns

Send targeted "complete your first order" promotions with incentives

### Onboarding analysis

Identify friction points in the registration-to-purchase funnel

### Account cleanup

Archive or remove inactive accounts after a threshold period

### Alternative approaches

You can also write this with NOT EXISTS: `SELECT * FROM customers c WHERE NOT EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.customer_id)`. Both approaches work; LEFT JOIN with IS NULL is generally more readable whilst NOT EXISTS can sometimes optimise better.

# Denormalised Order Details Report

Creating comprehensive reports often requires combining data from multiple tables into a single, readable format. This example demonstrates advanced aggregation and string manipulation.

## Example 3: Complete order details with line items

```
-- PostgreSQL version
SELECT
    o.order_id,
    o.order_date,
    c.name AS customer,
    c.email,
    o.status,
    o.total_amount,
    ARRAY_AGG(p.name || ' x' || oi.qty ORDER BY p.name) AS items,
    COUNT(oi.order_item_id) AS item_count
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN order_items oi ON oi.order_id = o.order_id
JOIN products p ON p.product_id = oi.product_id
GROUP BY o.order_id, o.order_date, c.name, c.email, o.status, o.total_amount
ORDER BY o.order_date DESC;
```

## MySQL equivalent

MySQL uses `GROUP_CONCAT` instead of `ARRAY_AGG`. The syntax differs slightly but achieves the same result.

```
-- MySQL version
SELECT
    o.order_id,
    o.order_date,
    c.name AS customer,
    o.total_amount,
    GROUP_CONCAT(
        CONCAT(p.name, ' x', oi.qty)
        ORDER BY p.name
        SEPARATOR '; '
    ) AS items
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN order_items oi ON oi.order_id = o.order_id
JOIN products p ON p.product_id = oi.product_id
GROUP BY o.order_id, o.order_date, c.name, o.total_amount;
```

## Understanding the aggregation

01

### Join all related tables

Connect orders → customers → order_items → products

02

### Group by order attributes

All non-aggregated columns must appear in GROUP BY

03

### Aggregate line items into text

ARRAY_AGG or GROUP_CONCAT combines product names and quantities

04

### Result format

One row per order with concatenated item list: "T-shirt x2; Socks x3"

# Efficient Pagination Strategies

Pagination is essential for displaying large datasets in web applications. However, the common OFFSET approach performs poorly at high page numbers. Let's explore both methods and understand their trade-offs.

## Traditional OFFSET pagination

Simple to implement but increasingly slow as you navigate to later pages because the database must scan and discard all previous rows.

```
-- Page 1: Records 1-10
SELECT * FROM orders
ORDER BY order_date DESC
LIMIT 10 OFFSET 0;

-- Page 6: Records 51-60
SELECT * FROM orders
ORDER BY order_date DESC
LIMIT 10 OFFSET 50;
```

### Advantages

- Simple implementation
- Works with any ORDER BY
- Can jump to arbitrary pages

### Disadvantages

- Performance degrades at high offsets
- Results shift if data changes between requests
- Database must scan discarded rows

## Keyset pagination (seek method)

Superior performance for large datasets by filtering using the last seen value rather than counting from the beginning.

```
-- Initial page
SELECT * FROM orders
ORDER BY order_date DESC, order_id DESC
LIMIT 10;

-- Next page: Use last row's values from previous page
SELECT * FROM orders
WHERE (order_date, order_id) < ('2023-05-02', 100)
ORDER BY order_date DESC, order_id DESC
LIMIT 10;
```

### Keyset pagination requirements

#### Unique ordering

Include unique column (usually ID) as secondary sort to break ties

#### Appropriate index

Create composite index on (order_date, order_id) for optimal performance

#### Client-side tracking

Store last seen values to build WHERE clause for next page

**When to use each method**

Use OFFSET for small datasets, admin interfaces, or when users need arbitrary page jumping. Use keyset pagination for infinite scroll interfaces, APIs serving mobile apps, or any high-volume public endpoints where users typically navigate sequentially.

# Database Transactions in Practice

Transactions ensure data consistency by grouping multiple operations into atomic units—they either all succeed together or all fail together. This is crucial for maintaining data integrity during complex operations.

## ACID properties

**Atomicity**

All operations succeed or all fail—no partial completion

**Consistency**

Database remains in valid state, respecting all constraints

**Isolation**

Concurrent transactions don't interfere with each other

**Durability**

Committed changes persist even after system failures

## Example 5: Order creation with transaction

Creating an order involves multiple tables and must maintain consistency. If any step fails—insufficient stock, database error, constraint violation—we must roll back all changes.

```
-- PostgreSQL transaction example
BEGIN;

-- Create the order record
INSERT INTO orders (customer_id, order_date, total_amount, status)
VALUES (1, CURRENT_DATE, 39.98, 'processing')
RETURNING order_id;  -- PostgreSQL returns the new ID

-- Assume we got order_id = 10 back

-- Add order line items
INSERT INTO order_items (order_id, product_id, qty, unit_price)
VALUES (10, 1, 2, 19.99);

-- Update product inventory (if you have a stock column)
UPDATE products
SET stock = stock - 2
WHERE product_id = 1;

-- Check if stock went negative (business rule violation)
-- If this SELECT returns a negative stock, we'll rollback
-- For automatic rollback, you can use CHECK constraints or triggers

COMMIT;  -- Only if everything succeeded
-- ROLLBACK;  -- Call this instead if any validations fail
```

## MySQL equivalent

```
START TRANSACTION;

INSERT INTO orders (customer_id, order_date, total_amount, status)
VALUES (1, CURDATE(), 39.98, 'processing');

SET @order_id = LAST_INSERT_ID();

INSERT INTO order_items (order_id, product_id, qty, unit_price)
VALUES (@order_id, 1, 2, 19.99);

COMMIT;
```

**Transaction best practices**

Keep transactions short—long-running transactions lock resources. Never include user input waits inside transactions. Use appropriate isolation levels for your use case. Always handle exceptions in application code to ensure ROLLBACK on errors.

# Common Pitfalls and Best Practices

Learning from common mistakes accelerates your database expertise. These pitfalls appear frequently in real-world applications—recognising and avoiding them saves time, prevents data loss, and improves performance.

## Critical mistakes to avoid

1

### Missing backups

Hardware fails. Humans make mistakes. Without recent backups, you risk catastrophic data loss. Schedule automated daily backups with pg_dump or mysqldump. Store backups off-site. Test restoration procedures regularly—untested backups are useless backups.

2

### Inadequate indexing

Missing indexes cause full table scans, destroying performance as data grows. Create indexes on foreign keys, frequently queried columns, and JOIN conditions. However, avoid index overload—too many indexes slow INSERT/UPDATE operations.

3

### Using SELECT * in production

Retrieving unnecessary columns wastes network bandwidth, increases memory usage, and slows query execution. Explicitly name required columns: `SELECT name, email FROM customers` instead of `SELECT * FROM customers`.

4

### SQL injection vulnerabilities

Never concatenate user input directly into SQL queries. Always use parameterised queries or prepared statements. Example: `cursor.execute("SELECT * FROM users WHERE email = %s", (user_email,))` instead of `f"SELECT * FROM users WHERE email = '{user_email}'"`.

5

### Over-normalisation for read-heavy workloads

Extreme normalisation creates complex joins that slow queries. For read-heavy applications, consider controlled denormalisation, materialised views, or caching strategies. Balance theoretical purity with practical performance needs.

## Performance best practices

### Monitor slow query logs

Enable and regularly review slow query logs to identify performance bottlenecks

### Use connection pooling

Reuse database connections instead of creating new ones for each request

### Implement appropriate caching

Cache frequently accessed, rarely changing data at application level with Redis or Memcached

# Quick Reference Cheat-Sheet

Keep these essential commands handy for daily database work. This reference covers common operations you'll use repeatedly.

## Index management

### Create index

```
-- Single column
CREATE INDEX idx_customers_email
    ON customers(email);

-- Composite index
CREATE INDEX idx_orders_lookup
    ON orders(customer_id, order_date);
```

### Remove index

```
-- PostgreSQL
DROP INDEX idx_customers_email;

-- MySQL
DROP INDEX idx_customers_email
    ON customers;
```

## Query analysis

### View execution plan

```
-- Basic plan
EXPLAIN
SELECT * FROM orders
WHERE customer_id = 1;

-- With actual execution times (PostgreSQL)
EXPLAIN ANALYZE
SELECT * FROM orders
WHERE customer_id = 1;
```

### Interpreting EXPLAIN output

Look for:

- **Seq Scan:** Full table scan (slow)
- **Index Scan:** Using index (fast)
- **Cost:** Estimated query expense
- **Rows:** Expected rows processed

## Quick statistics

```
-- Accurate row count (may be slow on large tables)
SELECT COUNT(*) FROM orders;

-- PostgreSQL: Approximate row count (very fast)
SELECT reltuples AS approximate_row_count
FROM pg_class
WHERE relname = 'orders';

-- MySQL: Approximate row count from metadata
SELECT table_rows
FROM information_schema.tables
WHERE table_schema = 'your_database'
  AND table_name = 'orders';
```

## Essential maintenance commands

### PostgreSQL: VACUUM

```
VACUUM ANALYZE orders;
```

Reclaim storage and update statistics

### MySQL: OPTIMIZE

```
OPTIMIZE TABLE orders;
```

Defragment and reclaim space

### Update statistics

```
-- PostgreSQL
ANALYZE orders;

-- MySQL
ANALYZE TABLE orders;
```

Refresh query optimiser statistics

# Self-Study Tasks: Practise and Master

Theory becomes expertise through practice. Complete these exercises to reinforce your learning and build confidence with real-world database scenarios.

## Hands-on exercises

### Task 1: Implement inventory management

Add a `stock` column to the products table. Write a transaction that decrements stock when creating an order. Include logic to rollback if stock would go negative. Test with various scenarios including insufficient inventory.

```
ALTER TABLE products ADD COLUMN stock INT DEFAULT 100;
```

### Task 2: Index optimisation experiment

Create an index for frequently queried columns: orders filtered by status and order_date. Use EXPLAIN ANALYZE to measure query times before and after index creation. Document the performance improvement percentage.

### Task 3: Master all join types

Write queries demonstrating each join type covered in this guide. For each query, document the result set and explain when you'd use that join type in a real application. Include INNER, LEFT, RIGHT, FULL OUTER, CROSS, and SELF joins.

### Task 4: Data import/export workflow

Export the orders table to CSV using your database's tools. Create a temporary table with identical structure. Import the CSV into the temp table. Verify row counts match and data integrity is preserved. Delete the temp table when finished.

### Task 5: Materialised views (PostgreSQL)

Create a materialised view storing monthly sales totals. Write a query that uses this view. Refresh the view after inserting new orders. Compare query performance: direct aggregation vs. querying the materialised view.

```
CREATE MATERIALIZED VIEW monthly_sales AS
SELECT
    DATE_TRUNC('month', order_date) AS month,
    SUM(total_amount) AS total_sales
FROM orders
GROUP BY DATE_TRUNC('month', order_date);

REFRESH MATERIALIZED VIEW monthly_sales;
```

## Next steps in your learning journey

### Advanced topics

Explore query optimisation, database replication, partitioning strategies, and full-text search capabilities

### Build a real project

Apply these skills by building a complete application with database backend—nothing teaches like hands-on experience

### Join the community

Participate in database forums, contribute to open-source projects, and learn from experienced developers

**Congratulations!**

You've completed a comprehensive journey through database fundamentals. Continue practising these concepts, explore your database system's documentation, and remember that expertise comes from consistent application of knowledge. Happy querying!