10-CHAPTER COURSE

Advanced SQL

Professional-level MySQL — joins, subqueries, CTEs, window functions, views, stored procedures, triggers, events, and query optimisation.

1 Database Design and Normalisation

2 INNER JOIN and LEFT JOIN

3 RIGHT JOIN, CROSS JOIN, Self-Joins, and Multi-Table Joins

4 Subqueries

5 Common Table Expressions

6 Window Functions

7 Views

8 Stored Procedures and Functions

9 Triggers and Scheduled Events

10 Query Optimisation

ADVANCED SQL — CHAPTER 1

Database Design and Normalisation

Chapter 1 — Database Design and Normalisation

Good SQL starts before you write a single query. A poorly designed schema creates problems that no amount of clever querying can fully fix: redundant data that drifts out of sync, updates that break things silently, and queries that are far slower than they need to be. This chapter covers normalisation — the formal rules for designing tables that avoid these problems — and ends by building the extended bookshop schema used throughout this entire course.

What this chapter covers: the three anomalies that bad design causes (update, insert, delete). First, Second, and Third Normal Form — what each requires, why it matters, and how to recognise a violation. Denormalisation: when breaking the rules is deliberate and correct. ER relationship types and foreign key behaviour options. The full extended bookshop schema: `publishers`, `genres`, `book_authors`, `book_genres`, `order_items`, and `reviews` — with all CREATE TABLE statements and the design decisions behind them.

The Problem — Three Anomalies Bad Design Creates

Normalisation solves three specific problems that arise when related data is stored in the same table without proper structure. Here's a poorly designed books table that stores the publisher's details directly alongside the book:

✗ Denormalised — publisher embedded in books

id	title	pub_name	pub_country	price
1	Dune	Hodder	UK	9.99
2	Foundation	Hodder	UK	8.99
3	Neuromancer	Hodder	UK	7.99
4	Hyperion	Gollancz	UK	9.99

pub_name and pub_country duplicated across rows 1–3

✓ Normalised — publishers in own table

id	title	pub_id	price
1	Dune	1	9.99
2	Foundation	1	8.99
3	Neuromancer	1	7.99
4	Hyperion	2	9.99

Publisher details live in publishers(id, name, country) — one row each

The denormalised version creates three anomalies:

- **Update anomaly:** if Hodder rebrands to "Hodder & Stoughton", you must update three rows — and if you miss one, the database now contradicts itself. With a `publishers` table, you update exactly one row.
- **Insert anomaly:** you can't record a new publisher until they have at least one book in the table. There's nowhere to store Penguin unless Penguin has a book row.
- **Delete anomaly:** if you delete Hyperion (the only Gollancz book), you lose the fact that Gollancz exists as a publisher. Publisher data is coupled to book data and they disappear together.

The Three Normal Forms

1NF

First Normal Form — atomic values, no repeating groups

Rule: every column must hold a single, indivisible value. No lists, no comma-separated values, no repeating column groups (genre1, genre2, genre3...). Every row must be uniquely identifiable by a primary key.

Test: can you split a column value and still get meaningful data? If yes, it violates 1NF.

2NF

Second Normal Form — no partial dependencies

Rule: must be in 1NF, and every non-key column must depend on the *whole* primary key — not just part of it. This only applies when the primary key is composite (two or more columns). If your primary key is a single column, you get 2NF for free.

Test: take each non-key column. Does it describe the whole PK, or just one part of it? If just part → partial dependency → 2NF violation.
3NF
Third Normal Form — no transitive dependencies
Rule: must be in 2NF, and no non-key column may depend on another non-key column. Non-key columns must depend only on the primary key — not on each other.
Test: take each non-key column. Does it describe the primary key, or does it describe another non-key column? If the latter → transitive dependency → 3NF violation.

1NF — Fixing Atomic Values

✗ Violates 1NF — genres in one column

id	title	genres
1	Dune	Science Fiction, Adventure
2	Gone Girl	Mystery, Thriller, Crime
3	Foundation	Science Fiction

Can't query "all science fiction books" without string parsing
✓ Satisfies 1NF — junction table

book_id	genre_id
1	1
1	2
2	3
2	4
2	5
3	1

book_genres table: each genre per book is its own row
-- The query that was impossible before is now simple SELECT b.title FROM books b JOIN book_genres bg ON bg.book_id = b.id JOIN genres g ON g.id = bg.genre_id WHERE g.name = 'Science Fiction';

2NF — Fixing Partial Dependencies

This only matters with composite primary keys. Consider an `order_items` table where the PK is `(order_id, book_id)`:

✗ Violates 2NF — book_title in order_items

order_id	book_id	book_title	qty
101	1	Dune	2
101	3	Foundation	1
102	1	Dune	1

book_title depends only on book_id, not on (order_id, book_id)
✓ Satisfies 2NF — title stays in books

order_id	book_id	qty	unit_price
101	1	2	9.99
101	3	1	8.99
102	1	1	9.99

title comes from books.title via JOIN; unit_price is order-specific (price may change)
Why store unit_price in order_items? The book's current price can change — a sale, a price increase. The order captures the price *at the time of purchase*. This is a deliberate decision: `unit_price` in `order_items` is not redundant with `books.price` — they represent different things. This pattern is correct even though it looks like duplication.

3NF — Fixing Transitive Dependencies

✗ Violates 3NF — publisher details in books

id	title	pub_id	pub_name	pub_country

1	Dune	1	Hodder	UK
2	Foundation	1	Hodder	UK
4	Hyperion	2	Gollancz	UK

pub_name and pub_country depend on pub_id, not on the book's id
✓ Satisfies 3NF — publishers table

id	name	country
1	Hodder	UK
2	Gollancz	UK

books only stores publisher_id as a FK; publisher details live in publishers

The chain `book.id → pub_id → pub_name` is the transitive dependency: `pub_name` doesn't describe the book — it describes the publisher. Moving it to a `publishers` table breaks the chain and eliminates the update, insert, and delete anomalies from the opening example.

Denormalisation — When Breaking the Rules is Correct

Normalisation is the right starting point. But databases serving read-heavy workloads — reporting dashboards, product listings, analytics — sometimes deliberately denormalise to avoid expensive joins at query time.

Denormalise intentionally, not accidentally. A denormalised table created on purpose, with a documented reason and a process to keep it in sync, is a valid engineering choice. A denormalised table that happened because nobody thought about design is a maintenance problem. The distinction is intention and control.

- **Derived/cached columns:** storing a pre-calculated `order_total` on the `orders` row avoids summing `order_items` on every page load. Acceptable if you update it on every insert/update to `order_items` (a trigger or application logic).
- **Redundant columns for speed:** adding `author_name` to `books` to avoid a join on the book listing page. Acceptable if update paths are controlled.
- **Separate reporting tables:** a daily snapshot table that copies and flattens live data for reporting queries. The snapshot is explicitly denormalised and rebuilt on schedule — not the source of truth.
- **The rule:** the normalised schema is always the authoritative source of truth. Denormalised copies are derived. They should be clearly marked as such in table names (e.g. `report_daily_sales`) or schema documentation.

ER Relationships — One-to-One, One-to-Many, Many-to-Many

One-to-One (1:1)
A ——— B
One row in A relates to exactly one row in B and vice versa. Rare in practice — if the data always appears together, put it in one table. Use when splitting a wide table for permissions (users vs user_secrets) or optional data (books vs book_cover_art).
Example: users ↔ user_preferences
One-to-Many (1:N)
A ——— B B B B
One row in A relates to many rows in B. The most common relationship. Implemented with a foreign key in B pointing to A's primary key. One publisher has many books; one customer has many orders.
Example: publishers ↔ books, customers ↔ orders
Many-to-Many (M:N)
A A A ——— B B B
Many rows in A relate to many rows in B. Implemented with a *junction table* (also called a bridge or associative table) that holds foreign keys to both A and B. One book has many genres; one genre has many books.
Example: books ↔ book_genres ↔ genres

Foreign Keys and ON DELETE Behaviour

A foreign key constraint enforces referential integrity: a row in the child table cannot reference a parent row that doesn't exist. When a parent row is deleted, MySQL needs to know what to do with the orphaned children.

ON DELETE option	What happens to child rows	Best for
RESTRICT	MySQL blocks the DELETE if any child rows reference this parent. The parent cannot be deleted until all children are deleted first.	Safety-first. Use when accidental deletion would be catastrophic (e.g. deleting a publisher that still has books).
CASCADE	Child rows are automatically deleted when the parent is deleted.	Owned data where children have no meaning without the parent. Deleting an order cascades to order_items. Deleting a user cascades to their sessions.
SET NULL	The FK column in child rows is set to NULL when the parent is deleted. The FK column must be nullable.	Optional references. A review's customer_id could be SET NULL if the customer deletes their account — the review stays, attributed to a deleted user.
NO ACTION	Identical to RESTRICT in MySQL InnoDB. Checks the constraint at the end of the statement.	Rarely chosen explicitly — defaults vary by engine and SQL standard.

The Extended Bookshop Schema

The SQL Foundations course used a simplified bookshop schema. The Advanced course extends it into a properly normalised database that can model real-world scenarios. All JOIN, subquery, CTE, window function, and optimisation examples in subsequent chapters use this schema.

Tables at a glance

publishers
id · name · country · founded_year
authors
id · first_name · last_name · nationality · bio
genres
id · name · slug
books
id · title · publisher_id · isbn · price · year_published · stock
book_authors
book_id · author_id · author_order
junction — M:N books↔authors
book_genres
book_id · genre_id
junction — M:N books↔genres
customers
id · first_name · last_name · email · created_at
orders
id · customer_id · status · created_at
no total_amount — it's derived
order_items
id · order_id · book_id · quantity · unit_price
reviews
id · book_id · customer_id · rating · body · created_at

ER diagram — entity relationships

publishers ──1:N──> books <──M:N──> authors (via book_authors) │ ├──M:N──> genres (via book_genres) │ ├──1:N──> order_items <──N:1──> orders <──N:1──> customers │ └──1:N──> reviews <──N:1──> customers Key relationships: publishers 1:N books one publisher has many books books M:N authors a book can have multiple authors (co-authored) books M:N genres a book can belong to multiple genres customers 1:N orders one customer places many orders orders 1:N order_items one order contains many line items books 1:N order_items one book appears in many orders books 1:N reviews one book has many reviews customers 1:N reviews one customer writes many reviews

CREATE TABLE — the full schema

```
-- ── Create and use the bookshop database ─────────────────────────── CREATE DATABASE IF NOT EXISTS bookshop CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci; USE bookshop; -- ── Reference tables (no foreign keys — no dependencies) ──────────── CREATE TABLE publishers ( id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY, name VARCHAR(200) NOT NULL, country VARCHAR(100) NOT NULL DEFAULT '', founded_year SMALLINT UNSIGNED DEFAULT NULL, UNIQUE KEY uq_publisher_name (name) ) ENGINE=InnoDB; CREATE TABLE authors ( id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY, first_name VARCHAR(100) NOT NULL, last_name VARCHAR(100) NOT NULL, nationality VARCHAR(100) DEFAULT NULL, bio TEXT DEFAULT NULL, INDEX idx_author_name (last_name, first_name) ) ENGINE=InnoDB; CREATE TABLE genres ( id TINYINT UNSIGNED AUTO_INCREMENT PRIMARY KEY, name VARCHAR(80) NOT NULL, slug VARCHAR(80) NOT NULL, UNIQUE KEY uq_genre_slug (slug) ) ENGINE=InnoDB; -- ── Core content table ──────────────────────────────────────────────── CREATE TABLE books ( id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY, title VARCHAR(300) NOT NULL, publisher_id INT UNSIGNED DEFAULT NULL, -- nullable: unknown publisher isbn CHAR(13) DEFAULT NULL, price DECIMAL(8,2) NOT NULL, year_published SMALLINT UNSIGNED DEFAULT NULL, stock INT UNSIGNED NOT NULL DEFAULT 0, description TEXT DEFAULT NULL, UNIQUE KEY uq_isbn (isbn), INDEX idx_books_publisher (publisher_id), INDEX idx_books_year (year_published), CONSTRAINT fk_books_publisher FOREIGN KEY (publisher_id) REFERENCES publishers(id) ON DELETE SET NULL -- keep the book if publisher is deleted ON UPDATE CASCADE ) ENGINE=InnoDB; -- ── Junction tables ─────────────────────────────────────────────────── CREATE TABLE book_authors ( book_id INT UNSIGNED NOT NULL, author_id INT UNSIGNED NOT NULL, author_order TINYINT UNSIGNED NOT NULL DEFAULT 1, -- 1 = primary author PRIMARY KEY (book_id, author_id), CONSTRAINT fk_ba_book FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE CASCADE, CONSTRAINT fk_ba_author FOREIGN KEY (author_id) REFERENCES authors(id) ON DELETE CASCADE ) ENGINE=InnoDB; CREATE TABLE book_genres ( book_id INT UNSIGNED NOT NULL, genre_id TINYINT UNSIGNED NOT NULL, PRIMARY KEY (book_id, genre_id), CONSTRAINT fk_bg_book FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE CASCADE, CONSTRAINT fk_bg_genre FOREIGN KEY (genre_id) REFERENCES genres(id) ON DELETE CASCADE ) ENGINE=InnoDB; -- ── Customer and order tables ───────────────────────────────────────── CREATE TABLE customers ( id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY, first_name VARCHAR(100) NOT NULL, last_name VARCHAR(100) NOT NULL, email VARCHAR(254) NOT NULL, created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP, UNIQUE KEY uq_customer_email (email), INDEX idx_customer_name (last_name, first_name) ) ENGINE=InnoDB; CREATE TABLE orders ( id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY, customer_id INT UNSIGNED NOT NULL, status ENUM('pending', 'paid', 'shipped', 'delivered', 'cancelled') NOT NULL DEFAULT 'pending', created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP, INDEX idx_orders_customer (customer_id), INDEX idx_orders_status (status), INDEX idx_orders_created (created_at), CONSTRAINT fk_orders_customer FOREIGN KEY (customer_id) REFERENCES customers(id) ON DELETE RESTRICT -- never silently delete a customer with orders ON UPDATE CASCADE ) ENGINE=InnoDB; CREATE TABLE order_items ( id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY, order_id INT UNSIGNED NOT NULL, book_id INT UNSIGNED NOT NULL, quantity SMALLINT UNSIGNED NOT NULL DEFAULT 1, unit_price DECIMAL(8,2) NOT NULL, -- price at time of purchase INDEX idx_oi_order (order_id), INDEX idx_oi_book (book_id), CONSTRAINT fk_oi_order FOREIGN KEY (order_id) REFERENCES orders(id) ON DELETE CASCADE, CONSTRAINT fk_oi_book FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE RESTRICT ) ENGINE=InnoDB; CREATE TABLE reviews ( id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY, book_id INT UNSIGNED NOT NULL, customer_id INT UNSIGNED DEFAULT NULL, -- nullable: guest review rating TINYINT UNSIGNED NOT NULL, body TEXT DEFAULT NULL, created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP, INDEX idx_reviews_book (book_id), INDEX idx_reviews_customer (customer_id), CONSTRAINT chk_rating CHECK (rating BETWEEN 1 AND 5), CONSTRAINT fk_rev_book FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE CASCADE, CONSTRAINT fk_rev_customer FOREIGN KEY (customer_id) REFERENCES customers(id) ON DELETE SET NULL ) ENGINE=InnoDB;
```

Sample data to follow along

```
-- Publishers INSERT INTO publishers (name, country, founded_year) VALUES ('Hodder & Stoughton', 'UK', 1868), ('Victor Gollancz', 'UK', 1927), ('Doubleday', 'USA', 1897), ('Ace Books', 'USA', 1952); -- Authors INSERT INTO authors (first_name, last_name, nationality) VALUES ('Frank', 'Herbert', 'American'), ('Isaac', 'Asimov', 'American'), ('William', 'Gibson', 'American'), ('Dan', 'Simmons', 'American'), ('Gillian', 'Flynn', 'American'), ('Agatha', 'Christie', 'British'); -- Genres INSERT INTO genres (name, slug) VALUES ('Science Fiction', 'science-fiction'), ('Thriller', 'thriller'), ('Mystery', 'mystery'), ('Cyberpunk', 'cyberpunk'), ('Adventure', 'adventure'), ('Crime', 'crime'); -- Books (publisher_id matches INSERT order above) INSERT INTO books (title, publisher_id, isbn, price, year_published, stock) VALUES ('Dune', 4, '9780441013593', 9.99, 1965, 50), ('Foundation', 3, '9780553293357', 8.99, 1951, 35), ('Neuromancer', 4, '9780441569595', 7.99, 1984, 28), ('Hyperion', 3, '9780385249492', 9.99, 1989, 40), ('Gone Girl', 3, '9780307588364', 8.49, 2012, 22), ('Murder on the Orient Express', 1, '9780007119318', 7.49, 1934, 60); -- book_authors (author_order: 1 = primary) INSERT INTO book_authors VALUES (1, 1, 1), -- Dune → Herbert (2, 2, 1), -- Foundation → Asimov (3, 3, 1), -- Neuromancer → Gibson (4, 4, 1), -- Hyperion → Simmons (5, 5, 1), -- Gone Girl → Flynn (6, 6, 1); -- Orient Express → Christie -- book_genres INSERT INTO book_genres VALUES (1, 1), (1, 5), -- Dune: Sci-Fi, Adventure (2, 1), -- Foundation: Sci-Fi (3, 1), (3, 4), -- Neuromancer: Sci-Fi, Cyberpunk (4, 1), (4, 5), -- Hyperion: Sci-Fi, Adventure (5, 2), (5, 3), (5, 6), -- Gone Girl: Thriller, Mystery, Crime (6, 3), (6, 6); -- Orient Express: Mystery, Crime -- Customers INSERT INTO customers (first_name, last_name, email) VALUES ('Alice', 'Martin', 'alice@example.com'), ('Bob', 'Nguyen', 'bob@example.com'), ('Carol', 'Patel', 'carol@example.com'); -- Orders INSERT INTO orders (customer_id, status, created_at) VALUES (1, 'delivered', '2026-04-10 09:00:00'), (1, 'shipped', '2026-05-20 11:30:00'), (2, 'delivered', '2026-03-05 14:00:00'), (3, 'pending', '2026-06-01 08:45:00'); -- Order items INSERT INTO order_items (order_id, book_id, quantity, unit_price) VALUES (1, 1, 2, 9.99), -- Alice: 2× Dune (1, 3, 1, 7.99), -- Alice: 1× Neuromancer (2, 4, 1, 9.99), -- Alice: 1× Hyperion (3, 2, 1, 8.99), -- Bob: 1× Foundation (3, 6, 1, 7.49), -- Bob: 1× Orient Express (4, 5, 1, 8.49); -- Carol: 1× Gone Girl -- Reviews INSERT INTO reviews (book_id, customer_id, rating, body, created_at) VALUES (1, 1, 5, 'A masterpiece. Read it twice.', '2026-04-15'), (1, 2, 4, 'Dense but
```

```
rewarding.', '2026-03-20'), (2, 2, 5, 'Asimov at his best.', '2026-03-10'), (3, 1, 4, 'Invented cyberpunk. Essential.', '2026-04-20'), (6, 2, 5, 'The plot twist still holds up 90 years on.', '2026-03-25'); -- Verify all tables loaded SELECT table_name, table_rows FROM information_schema.tables WHERE table_schema = 'bookshop' ORDER BY table_name;
```

Quick Reference — Chapter 1

Concept	The rule
1NF	Atomic values only — no lists, no comma-separated values, no repeating column groups. PK required.
2NF	No partial dependencies — every non-key column depends on the *whole* composite PK. Irrelevant if PK is a single column.
3NF	No transitive dependencies — non-key columns describe the PK, not each other.
M:N relationship	Always needs a junction table with FKs to both sides. PK is the composite of both FKs.
ON DELETE CASCADE	Child rows deleted automatically. Use for owned data (order_items when order deleted).
ON DELETE RESTRICT	Blocks parent deletion while children exist. Use for safety (can't delete customer with orders).
ON DELETE SET NULL	Child FK set to NULL. Use for optional references (review stays if customer is deleted).
unit_price in order_items	Not a 3NF violation — it records the price *at time of purchase*, distinct from current books.price.
Denormalisation	Deliberate only. Normalised schema = source of truth. Derived copies are clearly separate.

ADVANCED SQL — CHAPTER 2

INNER JOIN and LEFT JOIN

Chapter 2 — INNER JOIN and LEFT JOIN

Chapter 1 split the bookshop into ten normalised tables. Now comes the payoff: JOINs reunite that data at query time, combining rows from multiple tables into a single result set. INNER JOIN and LEFT JOIN cover 90% of real-world multi-table queries. This chapter works through both in depth — what they return, when to use each, and the patterns that will appear in almost every query you write.

This chapter uses the bookshop schema from Chapter 1. Run the CREATE TABLE and INSERT statements from Chapter 1 before following along if you haven't already. All queries are tested against that exact data set.

Why JOINs Exist

Normalisation splits data across tables to eliminate redundancy. The publisher's name lives in `publishers`, not in every `books` row. That's correct design — but when a user asks "what books does Doubleday publish?", we need both tables. A JOIN tells MySQL: *for each row in books, find the matching row in publishers and glue them together.*

The result of a JOIN looks like a single wide table. It isn't stored anywhere — MySQL builds it on the fly for this query, then discards it. The underlying tables stay exactly as they are.

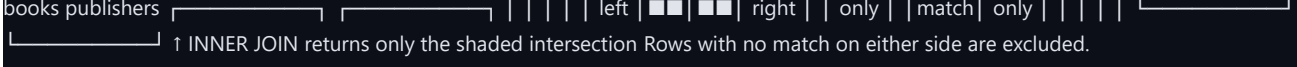
Explicit JOIN vs implicit JOIN: SQL has two syntaxes for joining. Implicit joins list tables in FROM separated by commas and filter in WHERE. Explicit joins use the JOIN keyword. Always use explicit JOIN — it makes the join condition visible and separate from the filter condition, and it prevents accidental cartesian products.
✗ Implicit join — never write this

```
SELECT b.title, p.name
FROM   books b, publishers p
WHERE  b.publisher_id = p.id;
-- Mix of join condition and filter in WHERE
-- If you forget the WHERE: 6 books × 4 publishers = 24 rows
```

✓ Explicit JOIN — always write this

```
SELECT b.title, p.name
FROM   books b
JOIN   publishers p ON p.id = b.publisher_id;
-- Join condition is ON, filter conditions go in WHERE
-- Clear, intentional, impossible to forget the ON
```

INNER JOIN



↑ INNER JOIN returns only the shaded intersection Rows with no match on either side are excluded.

INNER JOIN returns only rows where the ON condition is satisfied in *both* tables. If a book has a NULL `publisher_id`, it won't appear. If a publisher has no books, it won't appear. Only matched pairs make it into the result.

Basic syntax

SELECT b.title, p.name AS publisher, b.price, b.year_published FROM books b JOIN publishers p ON p.id = b.publisher_id ORDER BY b.year_published;
Result — 6 books, all with a matched publisher

title	publisher	price	year_published
Murder on the Orient Express	Hodder & Stoughton	7.49	1934
Foundation	Doubleday	8.99	1951
Dune	Ace Books	9.99	1965

Neuromancer	Ace Books	7.99	1984
Hyperion	Doubleday	9.99	1989
Gone Girl	Doubleday	8.49	2012

JOIN is shorthand for INNER JOIN. MySQL treats `JOIN` and `INNER JOIN` identically. Most developers write `JOIN` for inner joins and spell out `LEFT JOIN` / `RIGHT JOIN` / `CROSS JOIN` in full — the asymmetry makes the type visible at a glance.

Adding a WHERE clause — filtering after joining

-- Books from UK publishers only SELECT b.title, p.name AS publisher, p.country FROM books b JOIN publishers p ON p.id = b.publisher_id WHERE p.country = 'UK' ORDER BY b.title;
Result — only Murder on the Orient Express matches (Hodder = UK)

title	publisher	country
Murder on the Orient Express	Hodder & Stoughton	UK

Joining three tables — books + authors via book_authors

Books and authors have a many-to-many relationship through `book_authors`. To get author names on books, we join through the junction table:

-- Books with their primary author's full name SELECT b.title, CONCAT(a.first_name, ' ', a.last_name) AS author, b.price FROM books b JOIN book_authors ba ON ba.book_id = b.id AND ba.author_order = 1 -- primary author only JOIN authors a ON a.id = ba.author_id ORDER BY a.last_name;
Result

title	author	price
Foundation	Isaac Asimov	8.99
Murder on the Orient Express	Agatha Christie	7.49
Gone Girl	Gillian Flynn	8.49
Neuromancer	William Gibson	7.99
Dune	Frank Herbert	9.99
Hyperion	Dan Simmons	9.99

1
books b — start with the books table (aliased as b)
2
JOIN book_authors ba ON ba.book_id = b.id AND ba.author_order = 1 — join to the junction table, keeping only the primary author entry. The additional `AND` condition on the ON clause is applied during the join itself — it's more efficient than filtering in WHERE.
3
JOIN authors a ON a.id = ba.author_id — now join to authors using the author_id from the junction row. We now have one result row per book with all columns from all three tables available.
4
SELECT / ORDER BY — pick the columns we want and sort. MySQL has already done the hard work.

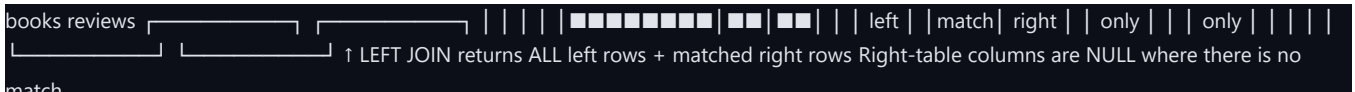
Aggregating across a join — average review rating per book

-- Books that have at least one review, with their average rating SELECT b.title, ROUND(AVG(r.rating), 1) AS avg_rating, COUNT(r.id) AS review_count FROM books b JOIN reviews r ON r.book_id = b.id GROUP BY b.id, b.title ORDER BY avg_rating DESC;
Result — only books with reviews appear (INNER JOIN excludes unreviewed books)

title	avg_rating	review_count
Foundation	5.0	1
Murder on the Orient Express	5.0	1
Dune	4.5	2
Neuromancer	4.0	1

Hyperion and Gone Girl are missing — they have no reviews yet. That's INNER JOIN's defining behaviour. To include them, we need LEFT JOIN.

LEFT JOIN



LEFT JOIN returns every row from the left table, regardless of whether a match exists in the right table. Where there is no match, all columns from the right table appear as NULL. This is exactly what you want when the relationship is optional: a book may or may not have reviews; you still want the book in the result.

All books with their average rating — including unreviewed books

```
-- All books, showing NULL avg_rating and 0 reviews for unreviewed books SELECT b.title, ROUND(AVG(r.rating), 1) AS avg_rating, COUNT(r.id) AS review_count FROM books b LEFT JOIN reviews r ON r.book_id = b.id GROUP BY b.id, b.title ORDER BY review_count DESC, b.title;
```

Result — all 6 books present; Hyperion and Gone Girl have no reviews

title	avg_rating	review_count
Dune	4.5	2
Foundation	5.0	1
Murder on the Orient Express	5.0	1
Neuromancer	4.0	1
Gone Girl	NULL	0
Hyperion	NULL	0

COUNT(r.id) vs COUNT(*): when using LEFT JOIN, `COUNT(*)` counts the NULL row for unmatched left-side entries — giving 1 instead of 0 for books with no reviews. `COUNT(r.id)` counts only non-NULL values in the right table's column — correctly giving 0 when there are no matching reviews. Always use `COUNT(column_from_right_table)` after a LEFT JOIN aggregation.

COALESCE — replacing NULLs in output

```
-- Show "No reviews yet" instead of NULL, and "0.0" instead of NULL avg SELECT b.title, COALESCE(CAST(ROUND(AVG(r.rating), 1) AS CHAR), 'No reviews yet') AS avg_rating, COUNT(r.id) AS review_count FROM books b LEFT JOIN reviews r ON r.book_id = b.id GROUP BY b.id, b.title ORDER BY review_count DESC, b.title;
```

All customers with their order count — including customers who never ordered

```
SELECT CONCAT(c.first_name, ' ', c.last_name) AS customer, COUNT(o.id) AS order_count, COALESCE(SUM(o.id), 0) AS placeholder FROM customers c LEFT JOIN orders o ON o.customer_id = c.id GROUP BY c.id ORDER BY order_count DESC;
```

Result — Alice has 2 orders, Bob has 1, Carol has 1 (all pending)

customer	order_count
Alice Martin	2
Bob Nguyen	1
Carol Patel	1

LEFT JOIN with a publisher that has books with nullable publisher_id

```
-- Add a book with no publisher to see LEFT JOIN behaviour INSERT INTO books (title, publisher_id, price, year_published, stock) VALUES ('Self-Published Manifesto', NULL, 4.99, 2025, 10); -- INNER JOIN: book with NULL publisher_id is excluded SELECT b.title, p.name AS publisher FROM books b JOIN publishers p ON p.id = b.publisher_id; -- 6 rows — Self-Published Manifesto not in result -- LEFT JOIN: all books, NULL publisher for the unmatched one SELECT b.title, COALESCE(p.name, 'Independent') AS publisher FROM books b LEFT JOIN publishers p ON p.id = b.publisher_id ORDER BY b.title; -- 7 rows — Self-Published Manifesto shows "Independent"
```

The Anti-Join Pattern — LEFT JOIN + WHERE IS NULL

A LEFT JOIN where you filter for NULL on the right table's column finds rows in the left table that have *no match* at all in the right table. This is called an anti-join — it's how you answer "which X has no Y?"

```
-- Which books have NO reviews yet? SELECT b.title, b.price FROM books b LEFT JOIN reviews r ON r.book_id = b.id WHERE r.id IS NULL -- NULL means no review row matched ORDER BY b.title;
```

Result

title	price
Gone Girl	8.49
Hyperion	9.99
Self-Published Manifesto	4.99

-- Which customers have NEVER placed an order? SELECT c.first_name, c.last_name, c.email FROM customers c LEFT JOIN orders o ON o.customer_id = c.id WHERE o.id IS NULL; -- Which authors have no books in the catalogue? SELECT a.first_name, a.last_name FROM authors a LEFT JOIN book_authors ba ON ba.author_id = a.id WHERE ba.book_id IS NULL;

The IS NULL filter must reference a NOT NULL column from the right table. Filter on the primary key or a column defined NOT NULL. If you accidentally filter on a nullable column, you can't tell whether NULL means "no match" or "the column value is NULL". `r.id`, `o.id`, and `ba.book_id` above are all primary or not-null keys — safe choices for the IS NULL test.

Joining on Non-Primary-Key Columns

The ON clause can match any column — not just primary and foreign keys. Joins on natural keys (ISBN, email, slug) and even on non-unique values are valid.

-- Join two hypothetical catalogue systems on ISBN (a natural key, not a PK) SELECT b.title, b.price AS our_price, ext.price AS competitor_price FROM books b JOIN external_catalogue ext ON ext.isbn = b.isbn; -- Join on a non-unique column: find all orders placed the same day as order #1 SELECT o2.id AS same_day_order, o2.customer_id, o2.status FROM orders o1 JOIN orders o2 ON DATE(o2.created_at) = DATE(o1.created_at) AND o2.id != o1.id -- exclude the original order itself WHERE o1.id = 1; -- Joining on a genre slug (string comparison) — no integer key involved SELECT b.title FROM books b JOIN book_genres bg ON bg.book_id = b.id JOIN genres g ON g.id = bg.genre_id WHERE g.slug = 'cyberpunk';

Index the join column for performance. MySQL uses the ON column to look up matching rows. If the column isn't indexed, MySQL scans the entire right-table for every left-table row — catastrophic on large tables. Primary keys are automatically indexed. Foreign key columns (like `book_id` in `reviews`) should have an index added explicitly, as we did in Chapter 1. For joins on natural keys like ISBN, add an index or UNIQUE constraint.

NULL Handling in JOINs

-- NULL != NULL — a NULL FK never matches any row -- This is why INNER JOIN excludes NULLs automatically -- Demonstrate: ON p.id = b.publisher_id where publisher_id IS NULL SELECT NULL = NULL; +-----------+ | NULL = NULL | +-----------+ | NULL | -- not TRUE — the comparison itself is NULL (unknown) +-----------+ -- MySQL evaluates ON conditions as true/false/null -- A NULL result is treated as false → the row is excluded from INNER JOIN -- Left JOIN includes it but sets all right-table columns to NULL -- Safe NULL check: use IS NULL / IS NOT NULL, never = NULL SELECT title FROM books WHERE publisher_id IS NULL; -- correct SELECT title FROM books WHERE publisher_id = NULL; -- always returns 0 rows

Common Mistakes

Missing the ON clause — accidental CROSS JOIN: `JOIN publishers p` without `ON p.id = b.publisher_id` is a CROSS JOIN — every book row is combined with every publisher row. 6 books × 4 publishers = 24 rows of garbage. MySQL 8.0 will usually throw a syntax error for a missing ON, but MySQL 5.x may not. Always write the ON.

-- Mistake 1: filtering on the wrong column SELECT b.title, p.name FROM books b JOIN publishers p ON p.id = b.id; -- wrong: should be b.publisher_id -- Joins book.id to publisher.id — nonsensical match -- Mistake 2: using LEFT JOIN but forgetting IS NULL in anti-join SELECT b.title FROM books b LEFT JOIN reviews r ON r.book_id = b.id; -- Returns ALL books including reviewed ones — forgot WHERE r.id IS NULL -- Mistake 3: COUNT(*) after LEFT JOIN SELECT b.title, COUNT(*) AS review_count -- returns 1 for unreviewed books FROM books b LEFT JOIN reviews r ON r.book_id = b.id GROUP BY b.id; -- Fix: COUNT(r.id) — counts only non-NULL values from the right table -- Mistake 4: ambiguous column name (both tables have "id") SELECT id, title, name -- ERROR: Column 'id' in field list is ambiguous FROM books b JOIN publishers p ON p.id = b.publisher_id; -- Fix: always qualify with alias when both tables share a column name SELECT b.id, b.title, p.name -- explicit table alias qualifiers FROM books b JOIN publishers p ON p.id = b.publisher_id;

Practical Example — Order Summary Report

A real-world query combining multiple JOINs and LEFT JOINs to produce an order line-item report:

-- Order summary: customer name, order date, book title, qty, line total -- Include orders with no items (pending with nothing added) via LEFT JOIN on items SELECT CONCAT(c.first_name, ' ', c.last_name) AS customer, o.id AS order_id, o.status, DATE(o.created_at) AS order_date, b.title AS

book, oi.quantity, oi.quantity * oi.unit_price AS line_total FROM orders o JOIN customers c ON c.id = o.customer_id LEFT JOIN order_items oi ON oi.order_id = o.id LEFT JOIN books b ON b.id = oi.book_id ORDER BY o.created_at, oi.id;

Result — all orders, with line items. An order with no items would show NULL for book/qty/total.

customer	order_id	status	order_date	book	qty	line_total
Bob Nguyen	3	delivered	2026-03-05	Foundation	1	8.99
Bob Nguyen	3	delivered	2026-03-05	Murder on the Orient Express	1	7.49
Alice Martin	1	delivered	2026-04-10	Dune	2	19.98
Alice Martin	1	delivered	2026-04-10	Neuromancer	1	7.99
Alice Martin	2	shipped	2026-05-20	Hyperion	1	9.99
Carol Patel	4	pending	2026-06-01	Gone Girl	1	8.49

Quick Reference — Chapter 2

Pattern	What it does
JOIN … ON	INNER JOIN — returns only rows with a match in both tables. Excludes NULLs and unmatched rows from either side.
LEFT JOIN … ON	Returns all rows from the left table + matched rows from the right. Right-table columns are NULL where there is no match.
LEFT JOIN … WHERE right.id IS NULL	Anti-join — rows in the left table that have no matching row in the right table. Always filter on a NOT NULL column.
COUNT(right_table.id)	After a LEFT JOIN GROUP BY, counts only matched rows (0 for unmatched). `COUNT(*)` wrongly gives 1.
COALESCE(col, 'default')	Replaces NULL in output with a fallback value. Use on right-table columns in LEFT JOIN results.
table alias	`FROM books b` — alias b. Use b.column everywhere. Short aliases (b, p, a, c, o, r, oi) improve readability and remove ambiguity when tables share column names.
ON clause condition	The join condition. Can include AND for extra filters applied during the join (more efficient than WHERE for LEFT JOINs). Use WHERE for post-join filters on the result.
Three-table join	`FROM a JOIN b ON … JOIN c ON …` — chain as many JOINs as needed. Each adds one more table to the working result set.
NULL = NULL	Evaluates to NULL (unknown), not TRUE. A NULL FK never matches any row in an INNER JOIN. Use IS NULL / IS NOT NULL for null checks — never = NULL.

ADVANCED SQL — CHAPTER 3

RIGHT JOIN, CROSS JOIN, Self-Joins, and Multi-Table Joins

Chapter 3 — RIGHT JOIN, CROSS JOIN, Self-Joins, and Multi-Table Joins

The previous chapter covered the two joins you'll write 90% of the time — INNER JOIN and LEFT JOIN. This chapter handles the rest of the toolkit: RIGHT JOIN (rarely needed, but good to recognise), CROSS JOIN (combinatorial queries and test data), self-joins (comparing rows in the same table), and the practical mechanics of chaining three or more tables together without losing your mind.

RIGHT JOIN
All rows from the *right* table; matching rows from the left. NULLs where the left has no match. Usually rewritable as a LEFT JOIN by swapping table order.
CROSS JOIN
Every row in table A paired with every row in table B — no join condition. Produces *m* × *n* rows. Use deliberately: test data grids, pricing matrices, calendar generation.
Self-Join
A table joined to itself using two aliases. Classic uses: employees and their managers, comparing pairs of rows, finding records that relate to each other.
3+ Table Joins
Chain joins one at a time. Give every table a short alias. Think of each JOIN as "add another column source." MySQL plans the order for you.
Schema reminder — all examples use the extended bookshop schema from Chapter 1: `books`, `authors`, `book_authors`, `publishers`, `customers`, `orders`, `order_items`, `reviews`. Setup scripts are in Chapter 1 if you need to recreate the tables.

1. RIGHT JOIN

A RIGHT JOIN is the mirror image of a LEFT JOIN. It keeps *all rows from the right table* and fills in NULLs for columns from the left table wherever there is no match.

RIGHT JOIN — right side highlighted

Syntax

SELECT columns FROM left_table RIGHT JOIN right_table ON left_table.id = right_table.left_id;

Example — find publishers with no books yet

We want every publisher in the `publishers` table, even those that haven't had any books added to the `books` table yet.

-- Publishers that have no books in the catalogue SELECT p.publisher_id, p.name AS publisher, b.title AS book_title FROM books b RIGHT JOIN publishers p ON b.publisher_id = p.publisher_id WHERE b.book_id IS NULL ORDER BY p.name;

publisher_id	publisher	book_title
7	Horizon Press	NULL
11	Velvet Pages	NULL

The rewrite rule — prefer LEFT JOIN

Any RIGHT JOIN can be rewritten as a LEFT JOIN by swapping the tables. Most developers use LEFT JOIN exclusively because it reads left-to-right the way you think about it: "start with *this* table, optionally attach *that* table."

-- Identical result — LEFT JOIN version (preferred style) SELECT p.publisher_id, p.name AS publisher, b.title AS book_title FROM publishers p LEFT JOIN books b ON b.publisher_id = p.publisher_id WHERE b.book_id IS NULL ORDER BY p.name;
Style rule: Always rewrite RIGHT JOINs as LEFT JOINs by swapping the table order. Mixing LEFT and RIGHT JOIN in the same query is legal but extremely hard to read. Pick one direction and stick to it — LEFT JOIN convention is almost universal.

When RIGHT JOIN is genuinely useful

You might see RIGHT JOIN in legacy code or when someone finds it easier to express a query that way. Accept it, understand it, but in new code always convert to LEFT JOIN.

2. CROSS JOIN — Cartesian Products

A CROSS JOIN produces every combination of rows from two tables. If table A has 5 rows and table B has 4 rows, the result has 5 × 4 = 20 rows. There is no ON condition — every row is paired with every other row.

Size warning: On large tables CROSS JOIN is catastrophic. 1,000 rows × 1,000 rows = 1,000,000 rows. Always know the cardinality of both sides before writing one, and add a WHERE clause if you only want a subset of combinations.

Syntax

SELECT a.col, b.col FROM table_a a CROSS JOIN table_b b;

Use case 1 — pricing matrix

Generate every combination of book format (hardcover, paperback, ebook) and discount tier (0%, 10%, 20%) to build a pricing grid:

-- Small helper tables — create temporarily for this example CREATE TEMPORARY TABLE formats (format VARCHAR(20)); INSERT INTO formats VALUES ('Hardcover'), ('Paperback'), ('Ebook'); CREATE TEMPORARY TABLE discounts (pct DECIMAL(4,2)); INSERT INTO discounts VALUES (0.00), (0.10), (0.20); -- All 9 combinations SELECT f.format, d.pct AS discount, ROUND(14.99 * (1 - d.pct), 2) AS sale_price FROM formats f CROSS JOIN discounts d ORDER BY f.format, d.pct;

format	discount	sale_price
Ebook	0.00	14.99
Ebook	0.10	13.49
Ebook	0.20	11.99
Hardcover	0.00	14.99
Hardcover	0.10	13.49
Hardcover	0.20	11.99
Paperback	0.00	14.99
Paperback	0.10	13.49
Paperback	0.20	11.99

Use case 2 — generate test data

Cross joining a list of first names with last names quickly builds a large customer test set:

-- 4 first names × 5 last names = 20 synthetic customers SELECT CONCAT(fn.first_name, ' ', ln.last_name) AS full_name, CONCAT(LOWER(fn.first_name), '.', LOWER(ln.last_name), '@example.com') AS email FROM (SELECT 'Alice' AS first_name UNION ALL SELECT 'Bob' UNION ALL SELECT 'Clara' UNION ALL SELECT 'Diego') fn CROSS JOIN (SELECT 'Smith' AS last_name UNION ALL SELECT 'Jones' UNION ALL SELECT 'Brown' UNION ALL SELECT 'Davis' UNION ALL SELECT 'Wilson') ln;

Implicit CROSS JOIN — a common accident

If you list two tables in FROM without a join condition, MySQL performs a CROSS JOIN silently. This is how beginners accidentally create million-row result sets:

-- ACCIDENTAL cross join — missing ON clause SELECT * FROM books, authors; -- ← every book × every author -- What was intended SELECT * FROM books b JOIN book_authors ba ON ba.book_id = b.book_id JOIN authors a ON a.author_id = ba.author_id;

Always write explicit JOIN syntax. The comma-separated FROM style (old SQL-89 syntax) hides cross joins and makes it easy to forget a condition. Modern SQL uses explicit JOIN … ON everywhere.

3. Self-Join — a Table Joined to Itself

A self-join is not a special JOIN keyword — it's just joining a table to itself using two different aliases. It's the standard solution whenever rows in the same table relate to each other.

Classic example — employee/manager hierarchy

Add a staff table to the bookshop schema to demonstrate:

CREATE TABLE IF NOT EXISTS staff (staff_id INT PRIMARY KEY AUTO_INCREMENT, name VARCHAR(100) NOT NULL, role VARCHAR(60), manager_id INT DEFAULT NULL, -- FK → staff.staff_id FOREIGN KEY (manager_id) REFERENCES staff(staff_id)); INSERT INTO staff (name, role, manager_id) VALUES ('Sandra Holt', 'General Manager', NULL), ('Tom Briggs', 'Buying Manager', 1), ('Rita Cho', 'Events Manager', 1), ('Kenji Mori', 'Buyer', 2), ('Amara Osei', 'Buyer', 2), ('Lena Fischer', 'Events Coordinator',3);

Self-join — show each staff member with their manager

SELECT emp.name AS employee, emp.role AS employee_role, mgr.name AS manager, mgr.role AS manager_role FROM staff emp LEFT JOIN staff mgr ON emp.manager_id = mgr.staff_id ORDER BY mgr.name NULLS FIRST, emp.name;

employee	employee_role	manager	manager_role
Sandra Holt	General Manager	NULL	NULL
Kenji Mori	Buyer	Tom Briggs	Buying Manager
Amara Osei	Buyer	Tom Briggs	Buying Manager
Lena Fischer	Events Coordinator	Rita Cho	Events Manager
Tom Briggs	Buying Manager	Sandra Holt	General Manager
Rita Cho	Events Manager	Sandra Holt	General Manager

Why LEFT JOIN? Sandra Holt has no manager (manager_id IS NULL). An INNER JOIN would drop her from the results — we'd silently lose the top of the hierarchy. Always use LEFT JOIN in self-joins unless you are certain every row has a parent.

Self-join — find pairs of books by the same publisher

Self-joins also work for comparing sibling rows — finding books that share something:

-- Pairs of books published by the same publisher (no duplicates, no self-pairs) SELECT a.title AS book_a, b.title AS book_b, p.name AS publisher FROM books a JOIN books b ON a.publisher_id = b.publisher_id AND a.book_id < b.book_id -- prevents (A,B) and (B,A) both appearing JOIN publishers p ON a.publisher_id = p.publisher_id ORDER BY p.name, a.title;

The < trick: When pairing rows in a self-join, `a.id < b.id` ensures each pair appears exactly once and a row is never paired with itself. Use <= only if you want self-pairs (a row matching itself).

4. Joining Three or More Tables

Multi-table joins are nothing more than chaining — each JOIN adds another "column source" to the growing result set. MySQL evaluates them left to right unless you explicitly hint otherwise. The key to readable multi-table queries is consistent, short table aliases.

Three-table join — books with their authors and publisher

-- books → book_authors (bridge) → authors → publishers SELECT b.title, CONCAT(a.first_name, ' ', a.last_name) AS author, p.name AS publisher FROM books b JOIN book_authors ba ON ba.book_id = b.book_id JOIN authors a ON a.author_id = ba.author_id JOIN publishers p ON p.publisher_id = b.publisher_id ORDER BY b.title;

Five-table join — full order line detail

A realistic reporting query: every order line with customer name, book title, author, and price paid.

SELECT o.order_id, o.order_date, CONCAT(c.first_name, ' ', c.last_name) AS customer, b.title AS book, CONCAT(a.first_name, ' ', a.last_name) AS author, oi.quantity, oi.unit_price, (oi.quantity * oi.unit_price) AS line_total FROM orders o JOIN customers c ON c.customer_id = o.customer_id JOIN order_items oi ON oi.order_id = o.order_id JOIN books b ON b.book_id = oi.book_id JOIN book_authors ba ON ba.book_id = b.book_id JOIN authors a ON a.author_id = ba.author_id ORDER BY o.order_date DESC, o.order_id, b.title;

Beware fan-out with many-to-many: If a book has two authors, each order line for that book will appear *twice* in this result — once per author. That's correct for listing authors, but it will double-count `line_total` in a SUM. To aggregate safely, either pick one author per book (MAX/MIN on author name), or aggregate before joining authors.

Mixing INNER and LEFT JOIN in the same query

You can — and often should — mix join types. Inner joins for required relationships, left joins for optional ones:

-- All books: required author + publisher, optional review score SELECT b.title, CONCAT(a.first_name, ' ', a.last_name) AS author, p.name AS publisher, ROUND(AVG(r.rating), 1) AS avg_rating, COUNT(r.review_id) AS review_count FROM books b JOIN book_authors ba ON ba.book_id = b.book_id -- required JOIN authors a ON a.author_id = ba.author_id -- required JOIN publishers p ON p.publisher_id = b.publisher_id -- required LEFT JOIN reviews r ON r.book_id = b.book_id -- optional GROUP BY b.book_id, b.title, a.first_name, a.last_name, p.name ORDER BY avg_rating DESC NULLS LAST;

Alias discipline — a worked example

With five tables in a query, ambiguous column names become a real problem. Always qualify every column with its alias. A consistent naming convention helps:

-- Convention: 1-2 letter alias = table initial(s) -- o = orders c = customers -- oi = order_items b = books -- p = publishers a = authors -- r = reviews ba = book_authors -- s = staff -- Never write: SELECT title, name, first_name ... -- Always write: SELECT b.title, p.name, a.first_name ...
Join order doesn't change correctness — MySQL's query planner reorders joins for performance. But write them in a logical sequence (start from the "central" table, radiate outward) so the query is easy to read and debug.

Putting It Together — A Worked Exercise

Build a query that answers: "*Which customers have never left a review for any book they purchased?*"

This requires four tables: customers, orders, order_items, and reviews.

-- Step 1: identify all books each customer has purchased -- Step 2: left-join reviews so un-reviewed purchases show NULL -- Step 3: filter to customers where ALL their purchases have NULL reviews SELECT DISTINCT c.customer_id, CONCAT(c.first_name, ' ', c.last_name) AS customer, c.email FROM customers c JOIN orders o ON o.customer_id = c.customer_id JOIN order_items oi ON oi.order_id = o.order_id LEFT JOIN reviews r ON r.book_id = oi.book_id AND r.customer_id = c.customer_id WHERE r.review_id IS NULL ORDER BY customer;
Why put the customer condition inside ON, not WHERE? If we move `r.customer_id = c.customer_id` to the WHERE clause, it effectively turns the LEFT JOIN into an INNER JOIN — any row with a non-matching review would be filtered out. Keeping it in ON means "match only reviews that belong to *this* customer; if none exist, keep the row anyway." Chapter 2 covered this pattern in detail.

Quick Reference

Join type	What it returns	Typical use	Watch out for
RIGHT JOIN	All rows from right table + matching rows from left (NULLs where no match)	"Find right-table rows with no left-table partner" — though usually rewritten as LEFT JOIN	Mixing LEFT and RIGHT in one query; just rewrite as LEFT JOIN
CROSS JOIN	Every row in A × every row in B — no condition	Pricing matrices, test data generation, calendar grids	Accidental cross join via comma-FROM; exponential row count on large tables
Self-join	Rows from the same table related to each other via an FK pointing back to the same table	Manager/employee hierarchies, comparing sibling rows, finding related records in one table	Forgetting LEFT JOIN at the top of a hierarchy; duplicate pairs without `a.id < b.id`
3+ table join	Progressively wider result set — each JOIN adds column sources	Any real reporting query that spans bridge/junction tables	Fan-out from many-to-many: rows multiply per join leg; inflates SUM/COUNT aggregates
`a.id < b.id` in self-join	Ensures each pair appears once; no self-pairs	Finding books by the same publisher, duplicate-detection	Use <= only if you explicitly want self-pairs
Condition in ON vs WHERE	ON condition applied during join (keeps LEFT JOIN soft); WHERE applied after (can harden to INNER)	Filtering on the optional table's columns — must go in ON	Moving optional-table filter to WHERE silently converts LEFT to INNER JOIN

Next: Chapter 4 — Subqueries: scalar, row, and table subqueries; correlated vs non-correlated; IN, EXISTS, ANY, ALL; when a subquery beats a join and when it doesn't.

ADVANCED SQL — CHAPTER 4

Subqueries

Chapter 4 — Subqueries

A subquery is a SELECT statement nested inside another SQL statement. It can appear almost anywhere an expression or table name is expected — in a SELECT list, a WHERE clause, a FROM clause, or even the target of an INSERT or UPDATE. The outer query sees the subquery's result as if it were a value, a row, or a temporary table.

Subqueries solve problems that joins can't easily express: "find customers who have *never* ordered," "find books priced above the average," "delete rows that match a condition on a different table." Understanding when to reach for a subquery — and when a join or CTE is better — is one of the most valuable judgement calls in SQL.

Chapter context: Chapter 3 covered joins; Chapter 5 covers CTEs. This chapter sits between them — subqueries pre-date both and are still the right tool in several specific situations. By the end you'll know exactly when that is.

1. Three Shapes of Subquery

Scalar subquery
Returns exactly one value (one row, one column). Can be used anywhere a single value is expected: SELECT list, WHERE, HAVING, ORDER BY.
Row subquery
Returns one row with multiple columns. Compared to a tuple of values using `= (val1, val2)` or row constructors. Rare in practice.
Table subquery
Returns multiple rows and columns. Used in FROM (derived table), with IN / NOT IN, EXISTS / NOT EXISTS, or ANY / ALL.

2. Scalar Subqueries

In the SELECT list — add a comparison column

-- Show each book's price alongside the overall average price SELECT title, price, (SELECT ROUND(AVG(price), 2) FROM books) AS avg_price, price - (SELECT AVG(price) FROM books) AS diff_from_avg FROM books ORDER BY diff_from_avg DESC;

title	price	avg_price	diff_from_avg
Dune	19.99	12.84	7.15
Foundation	16.50	12.84	3.66
Neuromancer	11.99	12.84	-0.85
1984	9.99	12.84	-2.85
The Hobbit	8.49	12.84	-4.35

Repeated scalar subqueries are evaluated once by the optimiser in most cases when they are not correlated. The two `(SELECT AVG(price) FROM books)` calls above are typically computed once and reused — but to be safe and explicit, a CTE (Chapter 5) or user variable is cleaner for complex queries.

In WHERE — filter against a computed value

-- Books priced above the catalogue average SELECT title, price FROM books WHERE price > (SELECT AVG(price) FROM books) ORDER BY price DESC;

In HAVING — filter groups against a computed value

-- Genres with more books than the average number of books per genre SELECT g.name AS genre, COUNT(bg.book_id) AS book_count FROM genres g JOIN book_genres bg ON bg.genre_id = g.genre_id GROUP BY g.genre_id, g.name HAVING COUNT(bg.book_id) > (SELECT AVG(cnt) FROM (SELECT COUNT(book_id) AS cnt FROM book_genres GROUP BY genre_id) counts) ORDER BY book_count DESC;
Scalar subqueries must return exactly one row and one column. If the subquery returns more than one row, MySQL raises error 1242: *Subquery returns more than 1 row*. If it returns zero rows, the result is NULL — which can silently break comparisons. Always be sure the subquery is truly scalar before using it this way.

3. IN and NOT IN

`IN (subquery)` tests whether a value appears in the set returned by the subquery. It's the most natural way to express "rows in table A whose key appears in table B."

IN — customers who have placed at least one order

SELECT customer_id, first_name, last_name, email FROM customers WHERE customer_id IN (SELECT DISTINCT customer_id FROM orders) ORDER BY last_name;

NOT IN — customers who have never ordered

SELECT customer_id, first_name, last_name, email FROM customers WHERE customer_id NOT IN (SELECT DISTINCT customer_id FROM orders) ORDER BY last_name;
NOT IN is broken when the subquery can return NULL. If `orders.customer_id` is nullable and even one row has NULL, `NOT IN` returns zero rows — the entire result set disappears silently. This is the most common subquery bug in production SQL.

The fix: add `WHERE customer_id IS NOT NULL` inside the subquery, or use `NOT EXISTS` instead (see below) — it handles NULLs correctly.
-- Safe version: guard against NULLs in the subquery SELECT customer_id, first_name, last_name FROM customers WHERE customer_id NOT IN (SELECT customer_id FROM orders WHERE customer_id IS NOT NULL -- ← always add this);

4. EXISTS and NOT EXISTS

`EXISTS (subquery)` returns TRUE if the subquery produces at least one row — it doesn't matter what the columns are. MySQL stops as soon as it finds the first matching row, making EXISTS very efficient for existence checks.

EXISTS is always correlated — the subquery references columns from the outer query. This is the key difference from IN: IN computes its full set first, then filters; EXISTS short-circuits per outer row.

EXISTS — customers with at least one order

SELECT c.customer_id, c.first_name, c.last_name FROM customers c WHERE EXISTS (SELECT 1 -- SELECT 1 is convention — the column doesn't matter FROM orders o WHERE o.customer_id = c.customer_id -- ← correlated reference) ORDER BY c.last_name;

NOT EXISTS — customers who have never ordered (NULL-safe)

SELECT c.customer_id, c.first_name, c.last_name, c.email FROM customers c WHERE NOT EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.customer_id) ORDER BY c.last_name;
NOT EXISTS vs LEFT JOIN … IS NULL: Both express "rows with no match." They produce identical results. In modern MySQL the optimiser often rewrites them to the same plan. Prefer whichever reads more naturally — NOT EXISTS is explicit about intent; LEFT JOIN IS NULL is familiar to join-oriented thinkers.

EXISTS with a condition — customers who spent over £100 in a single order

SELECT c.customer_id, c.first_name, c.last_name FROM customers c WHERE EXISTS (SELECT 1 FROM orders o JOIN order_items oi ON oi.order_id = o.order_id WHERE o.customer_id = c.customer_id GROUP BY o.order_id HAVING SUM(oi.quantity * oi.unit_price) > 100)

5. ANY and ALL

`ANY` and `ALL` compare a value against every value returned by a subquery using a comparison operator.

- `val > ANY (subquery)` — true if val is greater than *at least one* value in the set (equivalent to `val > MIN(subquery)`)
- `val > ALL (subquery)` — true if val is greater than *every* value in the set (equivalent to `val > MAX(subquery)`)
- `= ANY` is equivalent to `IN`
- `<> ALL` is equivalent to `NOT IN` (with the same NULL caveats)

ANY — books more expensive than at least one Penguin title

SELECT b.title, b.price FROM books b JOIN publishers p ON p.publisher_id = b.publisher_id WHERE p.name <> 'Penguin' AND b.price > ANY (SELECT price FROM books b2 JOIN publishers p2 ON p2.publisher_id = b2.publisher_id WHERE p2.name = 'Penguin') ORDER BY b.price DESC;

ALL — books more expensive than every Penguin title

SELECT b.title, b.price FROM books b JOIN publishers p ON p.publisher_id = b.publisher_id WHERE p.name <> 'Penguin' AND b.price > ALL (SELECT price FROM books b2 JOIN publishers p2 ON p2.publisher_id = b2.publisher_id WHERE p2.name = 'Penguin') ORDER BY b.price DESC;
In practice, most developers rewrite ANY/ALL with their aggregate equivalents (`> ANY` → `> (SELECT MIN(…))`, `> ALL` → `> (SELECT MAX(…))`) because the aggregate versions are more readable and unambiguously express intent. Know ANY/ALL for reading code; write the aggregate form.

6. Correlated vs Non-Correlated Subqueries

A **non-correlated** subquery is self-contained — it doesn't reference any column from the outer query. MySQL can evaluate it once and cache the result.

A **correlated** subquery references a column from the outer query. MySQL must re-evaluate it once for each row of the outer query. This makes them potentially slow on large tables.

Non-correlated — evaluated once

-- The inner SELECT has no reference to the outer query SELECT title, price FROM books WHERE price > (SELECT AVG(price) FROM books); -- ↑ no reference to outer "books" alias — runs once

Correlated — runs once per outer row

-- For each book, count how many reviews it has received SELECT b.title, b.price, (SELECT COUNT(*) FROM reviews r WHERE r.book_id = b.book_id -- ← b.book_id ties this to the outer row) AS review_count FROM books b ORDER BY review_count DESC;
Correlated subquery in SELECT list → rewrite as LEFT JOIN. The correlated subquery above runs once per book row. The equivalent LEFT JOIN runs one pass over both tables — much faster at scale:
-- Equivalent — and faster for large tables SELECT b.title, b.price, COUNT(r.review_id) AS review_count FROM books b LEFT JOIN reviews r ON r.book_id = b.book_id GROUP BY b.book_id, b.title, b.price ORDER BY review_count DESC;

When correlated subqueries are the right tool

They shine when the logic genuinely requires row-by-row comparison and a join would produce fan-out. A classic case: "find the most recent order for each customer" without using a window function:

-- Each customer's most recent order — correlated approach SELECT c.first_name, c.last_name, o.order_id, o.order_date FROM customers c JOIN orders o ON o.customer_id = c.customer_id WHERE o.order_date = (SELECT MAX(order_date) FROM orders o2 WHERE o2.customer_id = c.customer_id -- ← correlated) ORDER BY c.last_name;

7. Derived Tables — Subqueries in FROM

A subquery in the FROM clause acts as a temporary, unnamed table — often called a *derived table*. It must be given an alias. Every column you want to use from it must have a name (alias expressions if needed).

SELECT outer_col FROM (SELECT ... -- inner query FROM ...) AS alias_required -- alias is mandatory in MySQL WHERE ...

Derived table — filter on an aggregate without a CTE

-- Authors with an average book price above £15 SELECT sub.author, sub.book_count, sub.avg_price FROM (SELECT CONCAT(a.first_name, ' ', a.last_name) AS author, COUNT(b.book_id) AS book_count, ROUND(AVG(b.price), 2) AS avg_price FROM authors a JOIN book_authors ba ON ba.author_id = a.author_id JOIN books b ON b.book_id = ba.book_id GROUP BY a.author_id, a.first_name, a.last_name) sub WHERE sub.avg_price > 15 ORDER BY sub.avg_price DESC;
Derived table vs CTE: Both do the same job. A CTE (Chapter 5) names the block at the top with `WITH` and is reusable. A derived table is inline and can only be referenced once. Prefer CTEs for anything beyond a single, simple subquery — they are easier to read and debug.

Lateral derived tables (MySQL 8.0.14+)

A LATERAL derived table can reference columns from tables listed earlier in the FROM clause — removing the need for correlated subqueries in many cases:

-- For each customer, the value of their single largest order SELECT c.first_name, c.last_name, top_order.order_id, top_order.order_total FROM customers c JOIN LATERAL (SELECT o.order_id, SUM(oi.quantity * oi.unit_price) AS order_total FROM orders o JOIN order_items oi ON oi.order_id = o.order_id WHERE o.customer_id = c.customer_id -- ← reference outer table GROUP BY o.order_id ORDER BY order_total DESC LIMIT 1) top_order ON TRUE ORDER BY top_order.order_total DESC;

8. Subqueries in INSERT, UPDATE, and DELETE

INSERT ... SELECT — copy rows from one table to another

-- Archive orders older than 2 years into an orders_archive table INSERT INTO orders_archive (order_id, customer_id, order_date, total_amount) SELECT order_id, customer_id, order_date, total_amount FROM orders WHERE order_date < CURDATE() - INTERVAL 2 YEAR;

UPDATE with a subquery in SET

-- Set each book's price to the average for its publisher UPDATE books b SET b.price = (SELECT AVG(b2.price) FROM books b2 WHERE b2.publisher_id = b.publisher_id) WHERE b.publisher_id = 4;

UPDATE with a subquery in WHERE

-- Mark orders as 'priority' if the customer is a VIP (spent > £500 lifetime) UPDATE orders SET status = 'priority' WHERE customer_id IN (SELECT customer_id FROM v_customer_ltv WHERE lifetime_value > 500) AND status = 'pending';

DELETE with a subquery

-- Remove reviews left by customers who no longer exist -- (orphan cleanup after a non-cascading delete) DELETE FROM reviews WHERE customer_id NOT IN (SELECT customer_id FROM customers WHERE customer_id IS NOT NULL);
MySQL restriction: You cannot SELECT from the same table you are modifying in a DELETE or UPDATE in older MySQL versions. Wrap the subquery in a derived table to work around it:
-- Delete duplicate reviews — keep the one with the lowest review_id DELETE FROM reviews WHERE review_id NOT IN (SELECT min_id FROM (SELECT MIN(review_id) AS min_id FROM reviews GROUP BY book_id, customer_id) keeper -- ← derived table breaks the self-reference);

9. Subquery vs JOIN vs CTE — Decision Guide

Situation	Reach for	Why
"Does a related row exist?" with no columns needed from it	EXISTS / NOT EXISTS	Short-circuits. NULL-safe. Expresses intent clearly.
Filter on a single aggregate (avg price, max date)	Scalar subquery in WHERE	Concise. Non-correlated = evaluated once.
Need columns from both tables in the output	JOIN	Subqueries can't easily return data from both sides.
Complex multi-step logic, result reused more than once	CTE (WITH)	Named, readable, reusable. Replaces nested derived tables.
"Not in this list" — the list has no NULLs	NOT IN	Readable. Just ensure the subquery excludes NULLs.
"Not in this list" — NULLs possible	NOT EXISTS or LEFT JOIN IS NULL	NOT IN breaks silently on NULLs.
One result per outer row from a subquery with ORDER BY + LIMIT	LATERAL JOIN (MySQL 8.0.14+)	Only lateral derived tables can correlate and limit at the same time.
Add an aggregate alongside detail rows without collapsing them	Window function (Chapter 6)	Correlated subquery in SELECT is a common anti-pattern; window functions do it in one pass.

Quick Reference

Form	Returns	Used in	Watch out for
Scalar subquery	One value	SELECT list, WHERE, HAVING, ORDER BY, SET	Error if >1 row returned; NULL if 0 rows returned.
IN (subquery)	Set of values	WHERE	NULL in list makes IN always FALSE for that value.
NOT IN (subquery)	Set of values	WHERE	Any NULL in subquery returns zero rows. Always add `WHERE col IS NOT NULL`.
EXISTS (subquery)	Boolean	WHERE	Always correlated. Short-circuits on first match — fast.
NOT EXISTS (subquery)	Boolean	WHERE	NULL-safe alternative to NOT IN. Preferred for "no match" checks.
val > ANY (subquery)	Boolean	WHERE	True if val > at least one value. Equivalent to `val > (SELECT MIN(…))`.
val > ALL (subquery)	Boolean	WHERE	True if val > every value. Equivalent to `val > (SELECT MAX(…))`.
Derived table (FROM)	Virtual table	FROM clause	Must have an alias. Cannot be referenced more than once — use CTE instead.
LATERAL JOIN	Virtual table per outer row	JOIN … ON TRUE	MySQL 8.0.14+. Can reference outer columns. Useful with LIMIT inside the lateral.
Correlated subquery	Any of the above	WHERE, SELECT list	Runs once per outer row — expensive on large tables. Rewrite as JOIN or window function when possible.

Next: Chapter 5 — Common Table Expressions (CTEs): the WITH clause, chaining multiple CTEs, and recursive CTEs for hierarchical data.

ADVANCED SQL — CHAPTER 5

Common Table Expressions

Chapter 5 — Common Table Expressions (CTEs)

A Common Table Expression — written with the `WITH` keyword — is a named, temporary result set that exists only for the duration of a single SQL statement. Think of it as giving a subquery a name so you can refer to it clearly, reuse it, and chain multiple steps together without nesting queries five levels deep.

MySQL added full CTE support in version 8.0, including the powerful **recursive** variant that can walk tree structures and graphs. If you're still on MySQL 5.7 you won't have CTEs — one more reason to upgrade.

Availability: CTEs (WITH clause, including recursive) require MySQL 8.0+. MariaDB added them in 10.2. Check your version: `SELECT VERSION();`

1. Basic Syntax — WITH ... AS

WITH cte_name AS (← name you choose SELECT ... ← any valid SELECT FROM ... WHERE ...) SELECT * FROM cte_name ← reference the CTE here WHERE ... ;

The CTE name is scoped to the statement that defines it. You cannot reference it from other queries or sessions. It disappears the moment the statement finishes.

Why bother? — CTE vs inline subquery

Both queries below produce identical results. The CTE version is dramatically easier to read, especially once queries grow:

-- ❌ Inline subquery — hard to parse, inner query buried SELECT customer, total_spent FROM (SELECT CONCAT(c.first_name, ' ', c.last_name) AS customer, SUM(oi.quantity * oi.unit_price) AS total_spent FROM customers c JOIN orders o ON o.customer_id = c.customer_id JOIN order_items oi ON oi.order_id = o.order_id GROUP BY c.customer_id) sub WHERE total_spent > 100 ORDER BY total_spent DESC;
-- ✅ CTE version — logic named up front, main query reads cleanly WITH customer_spend AS (SELECT CONCAT(c.first_name, ' ', c.last_name) AS customer, SUM(oi.quantity * oi.unit_price) AS total_spent FROM customers c JOIN orders o ON o.customer_id = c.customer_id JOIN order_items oi ON oi.order_id = o.order_id GROUP BY c.customer_id) SELECT customer, total_spent FROM customer_spend WHERE total_spent > 100 ORDER BY total_spent DESC;
Good CTE names are nouns, not verbs. Name the result set, not what you're doing: `customer_spend`, `top_books`, `unpaid_orders` — not `get_customers` or `filter_orders`. The name should describe what the data *is*.

2. Chaining Multiple CTEs

Multiple CTEs are separated by commas — only one `WITH` keyword, multiple named blocks. Each CTE can reference *any CTE defined before it* in the chain.

WITH first_cte AS (SELECT ...), second_cte AS (SELECT ... FROM first_cte -- can reference first_cte), third_cte AS (SELECT ... FROM second_cte -- can reference first or second) SELECT * FROM third_cte;

Example — monthly bestsellers with rank

A three-step chain: aggregate sales → rank by month → filter to top 3 per month:

WITH -- Step 1: total units sold per book per month monthly_sales AS (SELECT b.book_id, b.title, DATE_FORMAT(o.order_date, '%Y-%m') AS month, SUM(oi.quantity) AS units_sold FROM order_items oi JOIN orders o ON o.order_id = oi.order_id JOIN books b ON b.book_id = oi.book_id GROUP BY b.book_id, b.title, month), -- Step 2: rank books within each month ranked_sales AS (SELECT month, title, units_sold, RANK() OVER (PARTITION BY month ORDER BY units_sold DESC) AS rnk FROM monthly_sales) -- Final: keep only top 3 per month SELECT month, rnk, title, units_sold FROM ranked_sales WHERE rnk <= 3 ORDER BY month DESC, rnk;
RANK() is a window function — covered in full in Chapter 6. For now, just know it numbers rows within each partition (month) by the ORDER BY expression. Ranked CTEs like this are a very common pattern you'll write regularly.

Example — two-stage customer segmentation

```
WITH spend_by_customer AS ( SELECT c.customer_id, CONCAT(c.first_name, ' ', c.last_name) AS customer, COUNT(DISTINCT o.order_id) AS order_count, SUM(oi.quantity * oi.unit_price) AS total_spent FROM customers c JOIN orders o ON o.customer_id = c.customer_id JOIN order_items oi ON oi.order_id = o.order_id GROUP BY c.customer_id, customer ), segmented AS ( SELECT customer, order_count, total_spent, CASE WHEN total_spent >= 200 THEN 'VIP' WHEN total_spent >= 75 THEN 'Regular' ELSE 'Casual' END AS segment FROM spend_by_customer ) SELECT segment, COUNT(*) AS customers, ROUND(AVG(total_spent), 2) AS avg_spend FROM segmented GROUP BY segment ORDER BY FIELD(segment, 'VIP', 'Regular', 'Casual');
```

segment	customers	avg_spend
VIP	3	284.50
Regular	8	112.75
Casual	14	38.20

CTEs are not materialised by default in MySQL 8. The optimiser may evaluate each CTE inline (as if it were a subquery) or materialise it into a temp table depending on cost. You can force materialisation with the `MATERIALIZE` optimiser hint if you want the CTE evaluated once and reused — useful when a CTE is referenced multiple times in the final query.

3. Recursive CTEs

A recursive CTE references itself. It's the standard SQL way to walk tree structures — category hierarchies, org charts, threaded comments, bill-of-materials — without application-side loops or multiple round-trips to the database.

Anchor member

The starting point — a normal SELECT that returns the root row(s). Runs once.

Recursive member

References the CTE by name to join the *previous iteration's output* back to the table. Runs repeatedly until it returns zero rows.

```
WITH RECURSIVE cte_name AS ( -- 1. Anchor: starting rows SELECT id, parent_id, name, 1 AS depth FROM tree_table WHERE parent_id IS NULL UNION ALL -- 2. Recursive: join previous result back to the table SELECT t.id, t.parent_id, t.name, c.depth + 1 FROM tree_table t JOIN cte_name c ON t.parent_id = c.id ) SELECT * FROM cte_name ORDER BY depth, name;
```

UNION ALL not UNION: Recursive CTEs must use `UNION ALL` (not `UNION DISTINCT`). Deduplication during recursion is not supported and would prevent the recursion from working correctly.

Setup — add a genre category tree

Add a self-referential `genres` table to the bookshop to model a category hierarchy:

```
CREATE TABLE IF NOT EXISTS genres ( genre_id INT PRIMARY KEY AUTO_INCREMENT, name VARCHAR(80) NOT NULL, parent_id INT DEFAULT NULL, FOREIGN KEY (parent_id) REFERENCES genres(genre_id) ); INSERT INTO genres (genre_id, name, parent_id) VALUES -- Root categories (1, 'Fiction', NULL), (2, 'Non-Fiction', NULL), (3, 'Children', NULL), -- Fiction sub-genres (4, 'Literary Fiction', 1), (5, 'Science Fiction', 1), (6, 'Fantasy', 1), (7, 'Crime & Thriller', 1), -- Non-Fiction sub-genres (8, 'History', 2), (9, 'Science', 2), (10, 'Self-Help', 2), -- Science Fiction sub-sub-genres (11, 'Space Opera', 5), (12, 'Cyberpunk', 5), (13, 'Hard SF', 5), -- Fantasy sub-sub-genres (14, 'Epic Fantasy', 6), (15, 'Urban Fantasy', 6);
```

Walk the entire tree — with indented display

```
WITH RECURSIVE genre_tree AS ( -- Anchor: root genres (no parent) SELECT genre_id, name, parent_id, 0 AS depth, CAST(name AS CHAR(500)) AS path -- breadcrumb path FROM genres WHERE parent_id IS NULL UNION ALL -- Recursive: join children of the current level SELECT g.genre_id, g.name, g.parent_id, gt.depth + 1, CONCAT(gt.path, ' > ', g.name) FROM genres g JOIN genre_tree gt ON g.parent_id = gt.genre_id ) SELECT genre_id, CONCAT(REPEAT(' ', depth), name) AS name_indented, depth, path FROM genre_tree ORDER BY path;
```

genre_id	name_indented	depth	path
3	Children	0	Children
1	Fiction	0	Fiction
7	Crime & Thriller	1	Fiction > Crime & Thriller
6	Fantasy	1	Fiction > Fantasy
14	Epic Fantasy	2	Fiction > Fantasy > Epic Fantasy
15	Urban Fantasy	2	Fiction > Fantasy > Urban Fantasy
5	Science Fiction	1	Fiction > Science Fiction

genre_id	name_indented	depth	path
13	Hard SF	2	Fiction > Science Fiction > Hard SF
12	Cyberpunk	2	Fiction > Science Fiction > Cyberpunk
11	Space Opera	2	Fiction > Science Fiction > Space Opera
2	Non-Fiction	0	Non-Fiction
8	History	1	Non-Fiction > History
9	Science	1	Non-Fiction > Science
10	Self-Help	1	Non-Fiction > Self-Help

Find all descendants of a single genre

Change the anchor to start at a specific node — the recursion will collect everything below it:

-- All sub-genres under "Fiction" (genre_id = 1) WITH RECURSIVE subtree AS (SELECT genre_id, name, parent_id, 0 AS depth FROM genres WHERE genre_id = 1 -- ← start here UNION ALL SELECT g.genre_id, g.name, g.parent_id, s.depth + 1 FROM genres g JOIN subtree s ON g.parent_id = s.genre_id) SELECT genre_id, name, depth FROM subtree WHERE depth > 0 -- exclude the root itself ORDER BY depth, name;

Find the ancestors of a leaf node (walk upward)

Flip the direction: start at a leaf and JOIN to the *parent* on each iteration:

-- Breadcrumb path from "Space Opera" up to root WITH RECURSIVE ancestors AS (SELECT genre_id, name, parent_id, 0 AS depth FROM genres WHERE name = 'Space Opera' UNION ALL SELECT g.genre_id, g.name, g.parent_id, a.depth - 1 FROM genres g JOIN ancestors a ON g.genre_id = a.parent_id -- walk UP via parent_id) SELECT name, depth FROM ancestors ORDER BY depth;

name	depth
Fiction	-2
Science Fiction	-1
Space Opera	0

4. Depth Limiting and Cycle Detection

Recursive CTEs will loop forever if your data contains a cycle (A is the parent of B, B is the parent of A) or if the recursion has a bug. MySQL enforces a hard limit via the `cte_max_recursion_depth` system variable (default: 1000 iterations) — when hit, the query errors rather than running forever. For production code, add explicit guards.

Limit depth with a WHERE condition

WITH RECURSIVE genre_tree AS (SELECT genre_id, name, parent_id, 0 AS depth FROM genres WHERE parent_id IS NULL UNION ALL SELECT g.genre_id, g.name, g.parent_id, gt.depth + 1 FROM genres g JOIN genre_tree gt ON g.parent_id = gt.genre_id WHERE gt.depth < 5 -- ← hard stop at depth 5) SELECT * FROM genre_tree ORDER BY depth, name;

Detect cycles with a path string

Track visited IDs in a path string. Before continuing, check that the next node's ID isn't already in the path:

WITH RECURSIVE safe_walk AS (SELECT genre_id, name, parent_id, 0 AS depth, CAST(genre_id AS CHAR(1000)) AS visited_ids, FALSE AS is_cycle FROM genres WHERE parent_id IS NULL UNION ALL SELECT g.genre_id, g.name, g.parent_id, sw.depth + 1, CONCAT(sw.visited_ids, ',', g.genre_id), FIND_IN_SET(g.genre_id, sw.visited_ids) > 0 AS is_cycle FROM genres g JOIN safe_walk sw ON g.parent_id = sw.genre_id WHERE sw.is_cycle = FALSE -- stop if cycle found) SELECT * FROM safe_walk WHERE is_cycle = FALSE;

Change the recursion depth limit for deep trees: `SET SESSION cte_max_recursion_depth = 5000;` Only increase this for legitimate deep hierarchies — not to paper over an infinite loop bug. Always add a depth guard in your recursive member too.

5. Practical CTE Patterns

Generate a date series (no date table needed)

-- Generate every day in June 2026 WITH RECURSIVE date_series AS (SELECT DATE('2026-06-01') AS dt UNION ALL SELECT DATE_ADD(dt, INTERVAL 1 DAY) FROM date_series WHERE dt < '2026-06-30') -- Left join to orders so days with zero orders still appear SELECT ds.dt, COALESCE(COUNT(o.order_id), 0) AS orders_placed FROM date_series ds LEFT JOIN orders o ON DATE(o.order_date) = ds.dt GROUP BY ds.dt ORDER BY ds.dt;

Date series are one of the most useful recursive CTE patterns. Generating a spine of dates and left-joining to transactional data ensures your report includes every day — even ones with no activity — without needing a pre-populated calendar table.

Generate a number series

-- Numbers 1 to 100 — useful for cross joins, test data, pagination WITH RECURSIVE nums AS (SELECT 1 AS n UNION ALL SELECT n + 1 FROM nums WHERE n < 100) SELECT n FROM nums;

CTE in an UPDATE or DELETE

CTEs can prefix UPDATE and DELETE statements too — useful for complex filtering:

-- Mark reviews as 'verified' where the customer actually purchased the book WITH verified_purchases AS (SELECT DISTINCT oi.book_id, o.customer_id FROM order_items oi JOIN orders o ON o.order_id = oi.order_id) UPDATE reviews r JOIN verified_purchases vp ON vp.book_id = r.book_id AND vp.customer_id = r.customer_id SET r.verified = TRUE WHERE r.verified = FALSE;

Quick Reference

Concept	Syntax / rule	Notes
Basic CTE	`WITH name AS ( SELECT … ) SELECT … FROM name`	Scoped to the single statement. Vanishes after execution.
Multiple CTEs	`WITH a AS (…), b AS (…) SELECT …`	One `WITH`, comma-separated. Later CTEs can reference earlier ones.
Recursive CTE	`WITH RECURSIVE name AS ( anchor UNION ALL recursive ) SELECT …`	MySQL 8.0+ only. Must use `UNION ALL`. Runs until recursive member returns zero rows.
Depth limit	`WHERE depth < N` in the recursive member	Always add this for production queries on real (potentially dirty) data.
System depth cap	`SET SESSION cte_max_recursion_depth = N;`	Default 1000. Increase for legitimately deep trees, not to hide bugs.
Path / cycle tracking	Build a `visited_ids` string; check with `FIND_IN_SET()`	Guards against cycles in data that should be acyclic but isn't.
Date/number series	Recursive CTE starting at a seed value, adding 1 or 1 day each iteration	Left-join against transactions to include zero-activity periods in reports.
CTE + UPDATE/DELETE	`WITH … UPDATE table JOIN cte …`	Useful for complex filter logic before modifying rows. MySQL syntax — no `FROM` needed.
Materialisation hint	`WITH name AS ( SELECT /*+ MATERIALIZE */ … )`	Forces MySQL to evaluate the CTE into a temp table once. Useful if referenced multiple times.

Next: Chapter 6 — Window functions: OVER, PARTITION BY, ROW_NUMBER, RANK, DENSE_RANK, LAG, LEAD, and running totals with SUM OVER.

ADVANCED SQL — CHAPTER 6

Window Functions

Chapter 6 — Window Functions

Window functions are the single most powerful upgrade between "knows SQL" and "writes professional SQL." They let you perform calculations *across a set of related rows* without collapsing them into a single output row the way GROUP BY does.

With a window function you can answer questions like: "What is each order's value *and* the running total up to that order?" or "Which book ranked #1 in sales *within each genre* this month?" — in a single query, without self-joins or subqueries.

MySQL version: Window functions require MySQL 8.0+. They are evaluated *after* WHERE and GROUP BY but *before* ORDER BY and LIMIT — which is why you cannot filter on a window function result in a WHERE clause directly. Use a CTE or subquery to wrap and filter.

1. The OVER() Clause — Anatomy

Every window function is a regular function followed by `OVER()`. What's inside OVER defines the "window" — which rows the function looks at when computing its result for the current row.

RANK() OVER (PARTITION BY genre_id ← reset the window per group (optional) ORDER BY units_sold DESC ← sort within the window (often required) ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ← frame: which rows count (optional))

- **PARTITION BY** — splits rows into independent groups (like GROUP BY, but rows are kept). Omit to treat the whole result set as one window.
- **ORDER BY** — defines row order inside each partition. Required for ranking and navigation functions; optional for pure aggregates.
- **Frame clause** — narrows which rows within the partition are included. Defaults to `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` when ORDER BY is present.

GROUP BY vs window function — the key difference

-- GROUP BY: one row per genre, detail lost SELECT genre_id, SUM(units_sold) AS total FROM monthly_sales GROUP BY genre_id; -- Window function: every row kept, total added as extra column SELECT genre_id, title, units_sold, SUM(units_sold) OVER (PARTITION BY genre_id) AS genre_total FROM monthly_sales;

2. Ranking Functions

ROW_NUMBER()
Unique sequential integer per row within the partition. Ties get different numbers — which tie wins depends on row order.
RANK()
Same as ROW_NUMBER but ties share the same rank and the next rank skips. 1, 2, 2, 4 …
DENSE_RANK()
Like RANK but no gaps. Ties share rank, next rank is always +1. 1, 2, 2, 3 …
NTILE(n)
Divides rows into n roughly equal buckets. Good for quartiles, deciles, percentile bands.
PERCENT_RANK()
Relative rank 0–1. (rank - 1) / (rows - 1). First row = 0.0, last = 1.0.
CUME_DIST()
Cumulative distribution: fraction of rows with value ≤ current row's value.

ROW_NUMBER, RANK, DENSE_RANK side by side

-- Rank books by units sold — see how ties behave differently SELECT title, units_sold, ROW_NUMBER() OVER (ORDER BY units_sold DESC) AS row_num, RANK() OVER (ORDER BY units_sold DESC) AS rnk, DENSE_RANK() OVER (ORDER BY units_sold DESC) AS dense_rnk FROM monthly_sales ORDER BY units_sold DESC;

title	units_sold	row_num	rnk	dense_rnk
Dune	42	1	1	1

title	units_sold	row_num	rnk	dense_rnk
Foundation	38	2	2	2
Neuromancer	35	3	3	3
The Hobbit	35	4	3	3
1984	29	5	5	4
Middlemarch	21	6	6	5

Neuromancer and The Hobbit both sold 35 copies — both get rank 3 under RANK() and DENSE_RANK(). RANK() then skips to 5 (a gap); DENSE_RANK() continues to 4 (no gap).

Top-N per group — the most common window pattern

Find the top 2 best-selling books within each genre this month. You cannot put a window function in WHERE — wrap it in a CTE:

WITH ranked_books AS (SELECT b.title, g.name AS genre, SUM(oi.quantity) AS units_sold, RANK() OVER (PARTITION BY bg.genre_id ORDER BY SUM(oi.quantity) DESC) AS rnk FROM order_items oi JOIN books b ON b.book_id = oi.book_id JOIN book_genres bg ON bg.book_id = b.book_id JOIN genres g ON g.genre_id = bg.genre_id GROUP BY b.book_id, b.title, bg.genre_id, g.name) SELECT genre, rnk, title, units_sold FROM ranked_books WHERE rnk <= 2 ORDER BY genre, rnk;

RANK vs ROW_NUMBER for top-N: Use `RANK()` when you want all tied rows at position N included (e.g. both books that tied for 2nd). Use `ROW_NUMBER()` when you want exactly N rows per partition, ties broken arbitrarily.

NTILE — bucket customers into quartiles

WITH spend AS (SELECT CONCAT(c.first_name, ' ', c.last_name) AS customer, SUM(oi.quantity * oi.unit_price) AS total_spent FROM customers c JOIN orders o ON o.customer_id = c.customer_id JOIN order_items oi ON oi.order_id = o.order_id GROUP BY c.customer_id, customer) SELECT customer, total_spent, NTILE(4) OVER (ORDER BY total_spent DESC) AS quartile FROM spend ORDER BY total_spent DESC;

3. Navigation Functions — LAG, LEAD, FIRST_VALUE, LAST_VALUE

LAG(col, n)
Value from n rows *before* the current row in the window. Default n=1. Returns NULL at the start.
LEAD(col, n)
Value from n rows *after* the current row. Returns NULL at the end.
FIRST_VALUE(col)
Value from the first row of the window frame. Every row in the partition can see the partition's first value.
LAST_VALUE(col)
Value from the last row of the current frame. Needs an explicit frame clause to reach the partition end.
NTH_VALUE(col, n)
Value from the nth row of the window frame. 1-indexed.

LAG and LEAD — month-over-month sales change

WITH monthly_totals AS (SELECT DATE_FORMAT(o.order_date, '%Y-%m') AS month, SUM(oi.quantity * oi.unit_price) AS revenue FROM orders o JOIN order_items oi ON oi.order_id = o.order_id GROUP BY month) SELECT month, revenue, LAG(revenue) OVER (ORDER BY month) AS prev_month_revenue, revenue - LAG(revenue) OVER (ORDER BY month) AS change, ROUND((revenue - LAG(revenue) OVER (ORDER BY month)) / LAG(revenue) OVER (ORDER BY month) * 100, 1) AS pct_change FROM monthly_totals ORDER BY month;

month	revenue	prev_month_revenue	change	pct_change
2026-01	1240.00	NULL	NULL	NULL
2026-02	980.50	1240.00	-259.50	-20.9
2026-03	1560.75	980.50	+580.25	+59.2
2026-04	1820.00	1560.75	+259.25	+16.6
2026-05	1390.25	1820.00	-429.75	-23.6

LAG with a default value

`LAG(col, n, default)` — the third argument replaces NULL at the boundary:

-- Replace NULL with 0 for the first row so arithmetic doesn't break LAG(revenue, 1, 0) OVER (ORDER BY month)

FIRST_VALUE — benchmark each month against January

SELECT month, revenue, FIRST_VALUE(revenue) OVER (ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS jan_revenue, ROUND(revenue / FIRST_VALUE(revenue) OVER (ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) * 100, 1) AS vs_jan_pct FROM monthly_totals ORDER BY month;

LAST_VALUE gotcha: By default the frame is `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, so LAST_VALUE returns the *current row's* value — not the partition's last value. Always add `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` when you want the true last value of the partition.

4. Aggregate Window Functions — Running Totals and Moving Averages

Any aggregate function (SUM, COUNT, AVG, MIN, MAX) can be used as a window function by adding OVER(). The frame clause controls exactly which rows are included in the aggregate.

Frame clause reference

Frame syntax	Meaning
`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`	All rows from partition start up to and including current row → running total
`ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`	All rows in the partition → same as no frame (partition total on every row)
`ROWS BETWEEN 2 PRECEDING AND CURRENT ROW`	Current row plus the 2 rows before it → 3-row moving window
`ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING`	One row before, current row, one row after → centred 3-row window
`RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`	Default when ORDER BY is present. Like ROWS but includes all peers (tied ORDER BY values).

Running total and cumulative percentage

WITH monthly_totals AS (SELECT DATE_FORMAT(o.order_date, '%Y-%m') AS month, SUM(oi.quantity * oi.unit_price) AS revenue FROM orders o JOIN order_items oi ON oi.order_id = o.order_id GROUP BY month) SELECT month, revenue, SUM(revenue) OVER (ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS running_total, ROUND(SUM(revenue) OVER (ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) / SUM(revenue) OVER () * 100 , 1) AS cumulative_pct FROM monthly_totals ORDER BY month;

month	revenue	running_total	cumulative_pct
2026-01	1240.00	1240.00	18.5%
2026-02	980.50	2220.50	33.1%
2026-03	1560.75	3781.25	56.4%
2026-04	1820.00	5601.25	83.5%
2026-05	1390.25	6991.50	100.0%

3-month moving average

SELECT month, revenue, ROUND(AVG(revenue) OVER (ORDER BY month ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) , 2) AS moving_avg_3m FROM monthly_totals ORDER BY month;

SUM() OVER () — no ORDER BY, no PARTITION BY — means "the total of the entire result set." This is the simplest window aggregate: every row sees the same grand total. Useful for computing percentages: `revenue / SUM(revenue) OVER () * 100`.

Running COUNT — orders placed so far this year

SELECT order_date, order_id, COUNT(*) OVER (ORDER BY order_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS orders_ytd FROM orders WHERE YEAR(order_date) = 2026 ORDER BY order_date;

5. Named Windows — the WINDOW Clause

When the same OVER() definition is repeated several times, define it once with `WINDOW … AS (…)` at the end of the query and reference it by name. This keeps queries DRY and reduces errors from copy-paste drift.

SELECT month, revenue, SUM(revenue) OVER w AS running_total, AVG(revenue) OVER w AS running_avg, COUNT(*) OVER w AS months_so_far, MIN(revenue) OVER w AS lowest_so_far, MAX(revenue) OVER w AS highest_so_far FROM monthly_totals WINDOW w AS (ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) ORDER BY month;

6. Practical Patterns

Deduplicate — keep only the most recent order per customer

ROW_NUMBER is the standard way to deduplicate or pick one row per group:

WITH latest_orders AS (SELECT *, ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date DESC, order_id DESC) AS rn FROM orders) SELECT customer_id, order_id, order_date, total_amount FROM latest_orders WHERE rn = 1 ORDER BY customer_id;

Detect gaps — find orders where the previous order was more than 90 days ago

WITH order_gaps AS (SELECT customer_id, order_id, order_date, LAG(order_date) OVER (PARTITION BY customer_id ORDER BY order_date) AS prev_order_date, DATEDIFF(order_date, LAG(order_date) OVER (PARTITION BY customer_id ORDER BY order_date)) AS days_since_last FROM orders) SELECT customer_id, order_date, prev_order_date, days_since_last FROM order_gaps WHERE days_since_last > 90 ORDER BY days_since_last DESC;

Pareto / cumulative share — find which books make up 80% of revenue

WITH book_revenue AS (SELECT b.title, SUM(oi.quantity * oi.unit_price) AS revenue FROM order_items oi JOIN books b ON b.book_id = oi.book_id GROUP BY b.book_id, b.title), cumulative AS (SELECT title, revenue, SUM(revenue) OVER (ORDER BY revenue DESC ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cumulative_revenue, SUM(revenue) OVER () AS grand_total FROM book_revenue) SELECT title, revenue, ROUND(cumulative_revenue / grand_total * 100, 1) AS cumulative_pct FROM cumulative WHERE (cumulative_revenue - revenue) / grand_total < 0.80 -- books needed to reach 80% ORDER BY revenue DESC;
Cannot filter on a window function in WHERE. MySQL evaluates window functions after WHERE. The pattern is always: compute in a CTE → filter in the outer SELECT's WHERE clause.

Quick Reference

Function / clause	What it does	Common use
ROW_NUMBER()	Unique integer per row within partition, no tie-sharing	Deduplication, exactly-N-per-group selection
RANK()	Ties share rank; next rank skips (gaps)	Top-N per group where ties should all be included
DENSE_RANK()	Ties share rank; no gaps in sequence	Medal-style ranking; consecutive rank bands
NTILE(n)	Divides rows into n buckets	Quartiles, deciles, percentile segmentation
LAG(col, n, default)	Value from n rows earlier in the window	Period-over-period comparison, gap detection
LEAD(col, n, default)	Value from n rows later in the window	Forward-looking metrics, time-to-next-event
FIRST_VALUE(col)	Value from first row of the frame	Benchmark every row against partition start
LAST_VALUE(col)	Value from last row of the frame (extend frame!)	Benchmark against partition end — needs `ROWS BETWEEN … UNBOUNDED FOLLOWING`
SUM/AVG/COUNT OVER (ORDER BY … ROWS BETWEEN …)	Running or moving aggregate	Running totals, moving averages, YTD counts
SUM(x) OVER ()	Grand total across all rows	Percentage-of-total: `x / SUM(x) OVER ()`
PARTITION BY	Resets window per group	Per-category ranks, per-customer running totals
WINDOW w AS (…)	Named window reused across multiple functions	Avoid repeating identical OVER() clauses
Cannot use in WHERE	Window functions evaluated after WHERE/GROUP BY	Always wrap in a CTE, then filter in outer WHERE

Next: Chapter 7 — Views: CREATE VIEW, querying views, updatable views, using views to simplify complex queries and control column-level access.

ADVANCED SQL — CHAPTER 7

Views

Chapter 7 — Views

A view is a saved SELECT statement stored in the database under a name. From the outside it looks and behaves exactly like a table — you can SELECT from it, JOIN it, and in many cases INSERT, UPDATE, and DELETE through it. The SELECT itself is stored; the data is not (unless you use a materialised view, which MySQL doesn't support natively).

Views serve three main purposes in production databases:

- **Simplify complex queries** — give a multi-join report query a short, stable name so application code doesn't repeat it.
- **Abstract schema changes** — rename a column in the underlying table; update the view definition; application code sees no change.
- **Control access** — expose only certain columns or rows to a database user without granting access to the base tables directly.

Views store definitions, not data. Every time you query a view, MySQL re-executes the underlying SELECT. There is no automatic caching. For expensive queries run repeatedly, consider a scheduled job that writes to a real summary table instead.

1. CREATE VIEW — Basic Syntax

CREATE [OR REPLACE] [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] VIEW view_name [(column_list)] AS select_statement [WITH [CASCADED | LOCAL] CHECK OPTION];

A simple view — full book catalogue with author and publisher

This is a query you'd otherwise paste into every report. Give it a name:

CREATE OR REPLACE VIEW v_book_catalogue AS SELECT b.book_id, b.title, b.isbn, b.price, CONCAT(a.first_name, ' ', a.last_name) AS author, p.name AS publisher, b.published_year FROM books b JOIN book_authors ba ON ba.book_id = b.book_id JOIN authors a ON a.author_id = ba.author_id JOIN publishers p ON p.publisher_id = b.publisher_id;

From this point on, application code just writes:

-- Looks exactly like querying a table SELECT * FROM v_book_catalogue WHERE publisher = 'Penguin'; SELECT * FROM v_book_catalogue WHERE price < 10.00 ORDER BY title; -- Join a view to a real table SELECT vc.title, vc.author, COUNT(r.review_id) AS reviews, ROUND(AVG(r.rating), 1) AS avg_rating FROM v_book_catalogue vc LEFT JOIN reviews r ON r.book_id = vc.book_id GROUP BY vc.book_id, vc.title, vc.author ORDER BY avg_rating DESC;

OR REPLACE vs DROP + CREATE

`CREATE OR REPLACE VIEW` atomically swaps the definition — no downtime, no gap where the view doesn't exist. Always prefer it over the two-step DROP then CREATE pattern.

-- Safe update of a view definition CREATE OR REPLACE VIEW v_book_catalogue AS SELECT b.book_id, b.title, b.isbn, b.price, b.stock_quantity, -- ← new column added CONCAT(a.first_name, ' ', a.last_name) AS author, p.name AS publisher, b.published_year FROM books b JOIN book_authors ba ON ba.book_id = b.book_id JOIN authors a ON a.author_id = ba.author_id JOIN publishers p ON p.publisher_id = b.publisher_id;

Named column list

You can rename columns in the view definition rather than using aliases inside the SELECT:

CREATE OR REPLACE VIEW v_customer_summary (customer_id, full_name, email, total_orders, lifetime_value) AS SELECT c.customer_id, CONCAT(c.first_name, ' ', c.last_name), c.email, COUNT(DISTINCT o.order_id), SUM(oi.quantity * oi.unit_price) FROM customers c JOIN orders o ON o.customer_id = c.customer_id JOIN order_items oi ON oi.order_id = o.order_id GROUP BY c.customer_id, c.first_name, c.last_name, c.email;

2. Updatable vs Non-Updatable Views

MySQL can execute INSERT, UPDATE, and DELETE statements *through* a view — as long as the view is simple enough that MySQL can unambiguously map the operation back to a single base table row.

✓ Updatable — view can be modified

- SELECT from a single base table
- No DISTINCT
- No aggregate functions (SUM, COUNT, AVG ...)
- No GROUP BY or HAVING
- No UNION or UNION ALL
- No subqueries in SELECT list
- No window functions
- All NOT NULL columns in base table either included or have defaults

✗ Non-updatable — read-only

- JOIN across multiple tables
- DISTINCT keyword present
- Aggregate functions in SELECT
- GROUP BY or HAVING clause
- UNION or UNION ALL
- Subquery in SELECT list or WHERE referencing the same table
- Window functions (OVER clause)
- Derived columns with no mapping to a base column

Example — an updatable view for book stock management

CREATE OR REPLACE VIEW v_book_stock AS SELECT book_id, title, stock_quantity, price FROM books; -- UPDATE through the view — affects the books table directly UPDATE v_book_stock SET stock_quantity = stock_quantity - 1 WHERE book_id = 42; -- INSERT through the view — inserts into books INSERT INTO v_book_stock (title, stock_quantity, price) VALUES ('A New Title', 50, 14.99); -- DELETE through the view DELETE FROM v_book_stock WHERE book_id = 99;

Check whether a view is updatable: `SELECT table_name, is_updatable FROM information_schema.views WHERE table_schema = DATABASE();`

WITH CHECK OPTION — enforce the view's WHERE on writes

Without CHECK OPTION, you can UPDATE a row through a view so that it no longer satisfies the view's WHERE clause — the row silently "disappears" from the view after the update. WITH CHECK OPTION blocks that:

-- View that only shows in-stock books CREATE OR REPLACE VIEW v_in_stock_books AS SELECT book_id, title, stock_quantity FROM books WHERE stock_quantity > 0 WITH CHECK OPTION; -- This UPDATE is blocked — would make the row invisible to this view UPDATE v_in_stock_books SET stock_quantity = 0 WHERE book_id = 5; -- ERROR 1369: CHECK OPTION failed 'bookshop.v_in_stock_books' -- This UPDATE is fine — row stays visible UPDATE v_in_stock_books SET stock_quantity = 10 WHERE book_id = 5;

CASCADED vs LOCAL check option: When a view is built on another view, `WITH CASCADED CHECK OPTION` (the default) enforces the WHERE clause of the current view *and all underlying views*. `WITH LOCAL CHECK OPTION` enforces only the current view's WHERE. Use CASCADED unless you have a specific reason not to — it is safer.

3. Column-Level Access Control via Views

MySQL's GRANT system works at the table level by default. Views are the standard way to expose only specific columns to a database user — the equivalent of column-level permissions without needing to grant them explicitly.

Scenario — a reporting user who must not see customer PII

-- Base table has sensitive columns: email, phone, address -- Create a view that omits them CREATE OR REPLACE VIEW v_customers_public AS SELECT customer_id, first_name, last_name, city, country, created_at FROM customers; -- email, phone, address deliberately excluded -- Grant the reporting user access to the VIEW only, not the base table GRANT SELECT ON bookshop.v_customers_public TO 'reporter'@'%'; -- Do NOT grant: GRANT SELECT ON bookshop.customers TO 'reporter'@'%';

Row-level security — filter rows per user

Views can also restrict which *rows* a user sees using `CURRENT_USER()` or a lookup table:

-- Each customer can only see their own orders CREATE OR REPLACE VIEW v_my_orders AS SELECT o.order_id, o.order_date, oi.book_id, b.title, oi.quantity, oi.unit_price FROM orders o JOIN order_items oi ON oi.order_id = o.order_id JOIN books b ON b.book_id = oi.book_id JOIN customers c ON c.customer_id = o.customer_id WHERE c.email = CURRENT_USER(); -- maps DB login to customer email

SQL SECURITY — DEFINER vs INVOKER: By default MySQL views run with `SQL SECURITY DEFINER` — the view executes with the privileges of the user who created it, not the user querying it. This means the querying user doesn't need access to the base tables at all, which is exactly what you want for access control views. Change to `SQL SECURITY INVOKER` if you want the caller's own privileges to apply.

-- Explicit DEFINER (default behaviour, but make it visible) CREATE OR REPLACE DEFINER = 'admin'@'localhost' SQL SECURITY DEFINER VIEW v_customers_public AS SELECT customer_id, first_name, last_name, city, country FROM customers;

4. View Algorithm — MERGE, TEMPTABLE, UNDEFINED

The ALGORITHM clause tells MySQL how to execute the view internally. In most cases you leave it as UNDEFINED and trust the optimiser — but knowing the options matters when performance is a concern.

MERGE

MySQL merges the view's SELECT with the outer query into a single query. The optimiser can use indexes on the base tables. This is what you want for most views.

TEMPTABLE

The view's SELECT runs first and its result is stored in a temporary table. The outer query runs against that. Indexes on base tables cannot help the outer WHERE — can be slow. Required for views with GROUP BY, DISTINCT, aggregate functions, or UNION.

UNDEFINED

Default. MySQL chooses MERGE if possible, falls back to TEMPTABLE when the view requires it. Almost always the right choice.

-- Force MERGE for a simple view (optimiser usually picks this anyway) CREATE OR REPLACE ALGORITHM = MERGE VIEW v_recent_orders AS SELECT order_id, customer_id, order_date, total_amount FROM orders WHERE order_date >= CURDATE() - INTERVAL 30 DAY; -- TEMPTABLE required here (GROUP BY makes it non-updatable too) CREATE OR REPLACE ALGORITHM = TEMPTABLE VIEW v_genre_revenue AS SELECT g.name AS genre, SUM(oi.quantity * oi.unit_price) AS revenue, COUNT(DISTINCT oi.order_id) AS orders FROM order_items oi JOIN books b ON b.book_id = oi.book_id JOIN book_genres bg ON bg.book_id = b.book_id JOIN genres g ON g.genre_id = bg.genre_id GROUP BY g.genre_id, g.name;

TEMPTABLE views cannot be updated and the outer WHERE clause cannot push predicates into the temp table scan — it always scans the full temp table. If you query a TEMPTABLE view with `WHERE genre = 'Fiction'`, MySQL cannot use an index; it materialises all rows first. For large aggregation views accessed frequently, consider writing results to a real summary table on a schedule.

5. Inspecting and Managing Views

List all views in the current database

SELECT table_name AS view_name, is_updatable, security_type, definer FROM information_schema.views WHERE table_schema = DATABASE() ORDER BY table_name;

view_name	is_updatable	security_type	definer
v_book_catalogue	NO	DEFINER	admin@localhost
v_book_stock	YES	DEFINER	admin@localhost
v_customer_summary	NO	DEFINER	admin@localhost
v_customers_public	YES	DEFINER	admin@localhost
v_genre_revenue	NO	DEFINER	admin@localhost
v_in_stock_books	YES	DEFINER	admin@localhost
v_my_orders	NO	DEFINER	admin@localhost
v_recent_orders	YES	DEFINER	admin@localhost

Show a view's full definition

-- Method 1: SHOW CREATE VIEW SHOW CREATE VIEW v_book_catalogue\G -- Method 2: information_schema (gives the clean SELECT without grants) SELECT view_definition FROM information_schema.views WHERE table_schema = DATABASE() AND table_name = 'v_book_catalogue';

DESCRIBE a view — see its columns

DESCRIBE v_book_catalogue; -- or SHOW COLUMNS FROM v_book_catalogue;

Drop a view

DROP VIEW IF EXISTS v_book_catalogue; -- Drop multiple views at once DROP VIEW IF EXISTS v_book_catalogue, v_book_stock, v_customer_summary;
Always use IF EXISTS when dropping views in scripts — prevents a hard error if the view was already removed or never existed.

6. Putting It Together — A Reporting View Layer

A well-designed application often has a thin view layer that sits between the raw schema and the application code. Here's a practical set for the bookshop:

-- ── 1. Full book catalogue (read-only — multi-table join) CREATE OR REPLACE VIEW v_book_catalogue AS SELECT b.book_id, b.title, b.isbn, b.price, b.stock_quantity, CONCAT(a.first_name, ' ', a.last_name) AS author, p.name AS publisher, b.published_year FROM books b JOIN book_authors ba ON ba.book_id = b.book_id JOIN authors a ON a.author_id = ba.author_id JOIN publishers p ON p.publisher_id = b.publisher_id; -- ── 2. In-stock books only (updatable + check option) CREATE OR REPLACE VIEW v_in_stock AS SELECT book_id, title, stock_quantity, price FROM books WHERE stock_quantity > 0 WITH CHECK OPTION; -- ── 3. Customer lifetime value (read-only — aggregate) CREATE OR REPLACE VIEW v_customer_ltv AS SELECT c.customer_id, CONCAT(c.first_name, ' ', c.last_name) AS customer, c.email, COUNT(DISTINCT o.order_id) AS total_orders, SUM(oi.quantity * oi.unit_price) AS lifetime_value, MAX(o.order_date) AS last_order_date FROM customers c JOIN orders o ON o.customer_id = c.customer_id JOIN order_items oi ON oi.order_id = o.order_id GROUP BY c.customer_id, customer, c.email; -- ── 4. Revenue by genre (read-only — aggregate) CREATE OR REPLACE VIEW v_genre_revenue AS SELECT g.genre_id, g.name AS genre, COUNT(DISTINCT oi.order_id) AS orders, SUM(oi.quantity) AS units_sold, SUM(oi.quantity * oi.unit_price) AS revenue FROM order_items oi JOIN books b ON b.book_id = oi.book_id JOIN book_genres bg ON bg.book_id = b.book_id JOIN genres g ON g.genre_id = bg.genre_id GROUP BY g.genre_id, g.name; -- ── 5. Low-stock alert (updatable — single table, with check) CREATE OR REPLACE VIEW v_low_stock AS SELECT book_id, title, stock_quantity FROM books WHERE stock_quantity < 5 WITH CHECK OPTION;

Now the application layer is clean

-- Homepage: featured in-stock books SELECT title, author, price FROM v_book_catalogue WHERE stock_quantity > 0 ORDER BY published_year DESC LIMIT 12; -- Restock report: books needing reorder SELECT * FROM v_low_stock ORDER BY stock_quantity; -- Marketing: top spenders this year SELECT customer, lifetime_value, total_orders, last_order_date FROM v_customer_ltv WHERE last_order_date >= '2026-01-01' ORDER BY lifetime_value DESC LIMIT 20;

Quick Reference

Concept	Syntax / rule	Notes
Create / replace	`CREATE OR REPLACE VIEW name AS SELECT …`	Atomic swap — prefer over DROP + CREATE.
Named columns	`CREATE VIEW name (col1, col2) AS SELECT …`	Alternative to aliases inside the SELECT list.
Updatable view	Single base table, no DISTINCT / GROUP BY / UNION / aggregates / window functions	INSERT, UPDATE, DELETE work through the view.
WITH CHECK OPTION	`… WHERE condition WITH CHECK OPTION`	Blocks writes that would make the row invisible to the view's WHERE clause.
CASCADED vs LOCAL	`WITH CASCADED CHECK OPTION` (default) / `WITH LOCAL CHECK OPTION`	CASCADED enforces check on all underlying views too. Safer default.
SQL SECURITY	`SQL SECURITY DEFINER` (default) / `SQL SECURITY INVOKER`	DEFINER runs with creator's privileges — caller needs no access to base tables. Ideal for access-control views.
ALGORITHM MERGE	`CREATE ALGORITHM = MERGE VIEW …`	Outer WHERE is pushed into base-table scan — indexes are used. Best performance.
ALGORITHM TEMPTABLE	`CREATE ALGORITHM = TEMPTABLE VIEW …`	View materialised to temp table first. Required for aggregates/UNION. Outer WHERE cannot use base-table indexes.
List views	`SELECT table_name, is_updatable FROM information_schema.views WHERE table_schema = DATABASE();`	Also shows security_type and definer.
Show definition	`SHOW CREATE VIEW name\G`	Returns the original CREATE VIEW statement.
Drop	`DROP VIEW IF EXISTS name;`	Safe drop — no error if view doesn't exist.

Concept	Syntax / rule	Notes
No stored data	Views execute their SELECT on every query	For expensive aggregations queried frequently, materialise to a real table on a schedule instead.

Next: Chapter 8 — Stored procedures and functions: CREATE PROCEDURE, IN/OUT/INOUT parameters, CREATE FUNCTION, flow control, calling from application code, and when to use them vs application logic.

ADVANCED SQL — CHAPTER 8

Stored Procedures and Functions

Chapter 8 — Stored Procedures and Functions

Stored procedures and functions let you save a block of SQL — with variables, flow control, and error handling — directly in the database under a name. Any authorised client can call them without knowing the SQL inside. They are the closest thing MySQL has to general-purpose programming.

Stored Procedure

- Called with `CALL name(...)`
- Can return zero, one, or many result sets
- Can have IN, OUT, and INOUT parameters
- Cannot be used inside a SELECT or expression
- Can execute DDL (CREATE TABLE, ALTER ...)
- Best for: multi-step workflows, batch jobs, encapsulating business logic

Stored Function

- Called like a built-in function: `SELECT name(...)`
- Returns exactly one scalar value via RETURN
- Only IN parameters (no OUT/INOUT)
- Can be used inside SELECT, WHERE, ORDER BY
- Cannot execute DDL or use CALL
- Best for: reusable calculations, custom formatting, derived values

1. The DELIMITER Problem

MySQL uses `;` to end statements. Inside a procedure body you also use `;` to end each sub-statement — which would confuse the client into thinking the procedure definition ended too early. The fix: temporarily change the client's statement delimiter to something that won't appear in your code, define the procedure, then restore `;`.

mysql> DELIMITER $$ -- change delimiter to $$ mysql> CREATE PROCEDURE example() BEGIN SELECT 'hello'; -- semicolons inside are fine now SELECT 'world'; END$$ -- $$ ends the CREATE PROCEDURE mysql> DELIMITER ; -- restore normal semicolons
DELIMITER is a client command, not SQL. It only works in the mysql CLI and MySQL Workbench. Application code (PHP, Python, Node ...) sends one statement at a time, so the DELIMITER trick is never needed there — you pass the entire CREATE PROCEDURE body as a single string to the driver.

2. CREATE PROCEDURE

Simplest possible procedure

DELIMITER $$ CREATE PROCEDURE sp_all_books() BEGIN SELECT book_id, title, price FROM books ORDER BY title; END$$ DELIMITER ; -- Call it CALL sp_all_books();

Parameter modes

IN (default)
Caller passes a value in. The procedure can read it but any changes are local — the caller's variable is unchanged.
OUT
Procedure writes a value back to the caller's variable. The initial value is NULL inside the procedure regardless of what the caller passed.
INOUT
Both directions: caller passes a value in AND the procedure can write a new value back. Use when the procedure modifies the caller's value in place.

IN parameter — filter books by genre

DELIMITER $$ CREATE PROCEDURE sp_books_by_genre(IN p_genre_name VARCHAR(80)) BEGIN SELECT b.title, b.price, b.stock_quantity FROM books b JOIN book_genres bg ON bg.book_id = b.book_id JOIN genres g ON g.genre_id = bg.genre_id WHERE g.name = p_genre_name ORDER BY b.title; END$$ DELIMITER ; CALL sp_books_by_genre('Science Fiction');

OUT parameter — return a computed value

DELIMITER $$ CREATE PROCEDURE sp_customer_total(IN p_customer_id INT, OUT p_order_count INT, OUT p_total_spent DECIMAL(10,2)) BEGIN SELECT COUNT(DISTINCT o.order_id), SUM(oi.quantity * oi.unit_price) INTO p_order_count, p_total_spent FROM orders o JOIN order_items oi ON oi.order_id = o.order_id WHERE o.customer_id = p_customer_id; END$$ DELIMITER ; -- Call: pass user variables to receive OUT values CALL sp_customer_total(7, @orders, @spent); SELECT @orders AS order_count, @spent AS total_spent;

INOUT parameter — apply a discount and return the new price

DELIMITER $$ CREATE PROCEDURE sp_apply_discount(INOUT p_price DECIMAL(10,2), IN p_discount_pct DECIMAL(5,2)) BEGIN SET p_price = p_price * (1 - p_discount_pct / 100); END$$ DELIMITER ; SET @price = 29.99; CALL sp_apply_discount(@price, 15); -- 15% off SELECT @price; -- 25.49

3. Flow Control

DECLARE — local variables

Local variables must be declared at the top of the BEGIN…END block, before any statements. They are scoped to the block and initialised to NULL.

DECLARE v_count INT DEFAULT 0; DECLARE v_name VARCHAR(100); DECLARE v_total DECIMAL(10,2) DEFAULT 0.00; DECLARE v_done BOOLEAN DEFAULT FALSE;

IF / ELSEIF / ELSE

DELIMITER $$ CREATE PROCEDURE sp_stock_status(IN p_book_id INT) BEGIN DECLARE v_qty INT; SELECT stock_quantity INTO v_qty FROM books WHERE book_id = p_book_id; IF v_qty IS NULL THEN SELECT 'Book not found' AS status; ELSEIF v_qty = 0 THEN SELECT 'Out of stock' AS status; ELSEIF v_qty < 5 THEN SELECT 'Low stock' AS status, v_qty AS remaining; ELSE SELECT 'In stock' AS status, v_qty AS remaining; END IF; END$$ DELIMITER ;

CASE statement

CASE v_segment WHEN 'VIP' THEN SET v_discount = 20; WHEN 'Regular' THEN SET v_discount = 10; ELSE SET v_discount = 0; END CASE;

WHILE loop

DECLARE v_i INT DEFAULT 1; WHILE v_i <= 10 DO INSERT INTO log_table (msg) VALUES (CONCAT('iteration ', v_i)); SET v_i = v_i + 1; END WHILE;

REPEAT … UNTIL (do-while equivalent)

REPEAT SET v_i = v_i + 1; UNTIL v_i > 10 END REPEAT;

LOOP with LEAVE (break)

my_loop: LOOP SET v_i = v_i + 1; IF v_i > 10 THEN LEAVE my_loop; -- break out of the loop END IF; IF v_i = 5 THEN ITERATE my_loop; -- continue (skip rest of body) END IF; INSERT INTO log_table (msg) VALUES (v_i); END LOOP my_loop;

4. A Realistic Procedure — Place an Order

This procedure handles the full order-placement workflow: validate stock, insert the order header, insert line items, decrement stock, and return the new order ID — all in one atomic transaction.

DELIMITER $$ CREATE PROCEDURE sp_place_order(IN p_customer_id INT, IN p_book_id INT, IN p_quantity INT, OUT p_order_id INT, OUT p_error_msg VARCHAR(255)) BEGIN DECLARE v_stock INT; DECLARE v_price DECIMAL(10,2); -- Default outputs SET p_order_id = NULL; SET p_error_msg = NULL; -- 1. Check the book exists and has stock SELECT stock_quantity, price INTO v_stock, v_price FROM books WHERE book_id = p_book_id; IF v_stock IS NULL THEN SET p_error_msg = 'Book not found'; LEAVE sp_place_order; -- MySQL allows LEAVE on the proc label END IF;

IF v_stock < p_quantity THEN SET p_error_msg = CONCAT('Only ', v_stock, ' copies in stock'); LEAVE sp_place_order; END IF; -- 2. Everything valid — run inside a transaction START TRANSACTION; INSERT INTO orders (customer_id, order_date, status) VALUES (p_customer_id, NOW(), 'pending'); SET p_order_id = LAST_INSERT_ID(); INSERT INTO order_items (order_id, book_id, quantity, unit_price) VALUES (p_order_id, p_book_id, p_quantity, v_price); UPDATE books SET stock_quantity = stock_quantity - p_quantity WHERE book_id = p_book_id; COMMIT; END$$ DELIMITER ; -- Usage CALL sp_place_order(3, 12, 2, @new_order_id, @err); SELECT @new_order_id, @err;

Always pair START TRANSACTION with error handling in real code. The example above is simplified — a production procedure should declare an `EXIT HANDLER FOR SQLEXCEPTION` that rolls back the transaction on any error. See the error handling section below.

5. Error Handling — DECLARE ... HANDLER

MySQL stored procedures have a structured error-handling mechanism. You declare a handler that fires when a specific condition occurs.

DECLARE {EXIT | CONTINUE} HANDLER FOR condition_value handler_statement;

- **EXIT HANDLER** — runs the statement then exits the current BEGIN...END block.
- **CONTINUE HANDLER** — runs the statement then continues from the line after the error.

Transaction rollback on any error

DELIMITER $$ CREATE PROCEDURE sp_safe_transfer(IN p_from_book INT, IN p_to_book INT, IN p_units INT) BEGIN DECLARE v_error BOOLEAN DEFAULT FALSE; -- Fire on any SQL error or warning-escalated-to-error DECLARE EXIT HANDLER FOR SQLEXCEPTION BEGIN ROLLBACK; SET v_error = TRUE; SELECT 'Transfer failed — rolled back' AS result; END; START TRANSACTION; UPDATE books SET stock_quantity = stock_quantity - p_units WHERE book_id = p_from_book; UPDATE books SET stock_quantity = stock_quantity + p_units WHERE book_id = p_to_book; COMMIT; SELECT 'Transfer complete' AS result; END$$ DELIMITER ;

CONTINUE handler — cursor NOT FOUND signal

DECLARE v_done INT DEFAULT FALSE; -- Fires when a SELECT INTO finds no rows, or a cursor runs out of rows DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done = TRUE;

Handle a specific error code

-- 1062 = Duplicate entry (unique constraint violation) DECLARE CONTINUE HANDLER FOR 1062 SET @dup_error = 'Duplicate key — record already exists';

6. Cursors — Row-by-Row Processing

A cursor lets you iterate over a result set one row at a time inside a procedure. Use them reluctantly — set-based SQL is almost always faster. But for logic that genuinely can't be expressed as a single query (e.g. each row triggers a different branching action), cursors are the right tool.

DELIMITER $$ CREATE PROCEDURE sp_apply_bulk_discount(IN p_genre_name VARCHAR(80), IN p_discount_pct DECIMAL(5,2), OUT p_books_updated INT) BEGIN DECLARE v_done BOOLEAN DEFAULT FALSE; DECLARE v_book_id INT; DECLARE v_price DECIMAL(10,2); SET p_books_updated = 0; -- 1. Declare the cursor DECLARE cur_books CURSOR FOR SELECT b.book_id, b.price FROM books b JOIN book_genres bg ON bg.book_id = b.book_id JOIN genres g ON g.genre_id = bg.genre_id WHERE g.name = p_genre_name; -- 2. Handler fires when the cursor has no more rows DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done = TRUE; -- 3. Open the cursor OPEN cur_books; read_loop: LOOP -- 4. Fetch one row FETCH cur_books INTO v_book_id, v_price; IF v_done THEN LEAVE read_loop; END IF; -- 5. Row-level logic UPDATE books SET price = ROUND(v_price * (1 - p_discount_pct / 100), 2) WHERE book_id = v_book_id; SET p_books_updated = p_books_updated + 1; END LOOP; -- 6. Close the cursor CLOSE cur_books; END$$ DELIMITER ; CALL sp_apply_bulk_discount('Fantasy', 15, @updated); SELECT @updated AS books_discounted;

Cursor order: DECLARE variables → DECLARE cursor → DECLARE handlers → statements. The order matters — MySQL enforces it. Handlers must be declared after cursors.

7. CREATE FUNCTION

A stored function returns exactly one value and can be used anywhere a built-in function can — in SELECT lists, WHERE clauses, ORDER BY, computed column defaults. It's the right choice for reusable logic that produces a single derived value.

CREATE [OR REPLACE] FUNCTION function_name ([param datatype [, ...]]) RETURNS datatype [DETERMINISTIC | NOT DETERMINISTIC] [READS SQL DATA | MODIFIES SQL DATA | NO SQL | CONTAINS SQL] BEGIN -- body RETURN value; END

DETERMINISTIC means the function always returns the same output for the same input. Mark it DETERMINISTIC when true — the optimiser can cache results and binary logging is safer. If your function reads from a table it is NOT DETERMINISTIC (data can change between calls). MySQL requires you to declare one or the other.

Function — customer segment label

DELIMITER $$ CREATE OR REPLACE FUNCTION fn_customer_segment(p_lifetime_value DECIMAL(10,2)) RETURNS VARCHAR(20) DETERMINISTIC NO SQL BEGIN DECLARE v_segment VARCHAR(20); IF p_lifetime_value >= 200 THEN SET v_segment = 'VIP'; ELSEIF p_lifetime_value >= 75 THEN SET v_segment = 'Regular'; ELSE SET v_segment = 'Casual'; END IF; RETURN v_segment; END$$ DELIMITER ;

Now it works anywhere an expression is valid:

-- Use in SELECT list SELECT customer, lifetime_value, fn_customer_segment(lifetime_value) AS segment FROM v_customer_ltv ORDER BY lifetime_value DESC; -- Use in WHERE SELECT customer, lifetime_value FROM v_customer_ltv WHERE fn_customer_segment(lifetime_value) = 'VIP'; -- Use in ORDER BY SELECT * FROM v_customer_ltv ORDER BY fn_customer_segment(lifetime_value), lifetime_value DESC;

Function — formatted price with currency symbol

DELIMITER $$ CREATE OR REPLACE FUNCTION fn_format_price(p_amount DECIMAL(10,2), p_currency CHAR(3)) RETURNS VARCHAR(20) DETERMINISTIC NO SQL BEGIN DECLARE v_symbol CHAR(3); SET v_symbol = CASE p_currency WHEN 'GBP' THEN '£' WHEN 'USD' THEN '$' WHEN 'EUR' THEN '€' ELSE p_currency END; RETURN CONCAT(v_symbol, FORMAT(p_amount, 2)); END$$ DELIMITER ; SELECT title, fn_format_price(price, 'GBP') AS display_price FROM books ORDER BY price DESC;

8. Inspecting and Dropping Stored Code

-- List all procedures in the current database SHOW PROCEDURE STATUS WHERE Db = DATABASE()\G -- List all functions SHOW FUNCTION STATUS WHERE Db = DATABASE()\G -- Show a procedure's full definition SHOW CREATE PROCEDURE sp_place_order\G -- Show a function's full definition SHOW CREATE FUNCTION fn_customer_segment\G -- information_schema alternative SELECT routine_name, routine_type, created, last_altered FROM information_schema.routines WHERE routine_schema = DATABASE() ORDER BY routine_type, routine_name; -- Drop DROP PROCEDURE IF EXISTS sp_place_order; DROP FUNCTION IF EXISTS fn_customer_segment;

9. Calling from Application Code

Python (mysql-connector-python)

import mysql.connector conn = mysql.connector.connect(host='localhost', database='bookshop', user='admin', password='secret') cursor = conn.cursor() # Call a procedure with IN parameters and a result set cursor.callproc('sp_books_by_genre', ('Science Fiction',)) for result in cursor.stored_results(): for row in result.fetchall(): print(row) # Call a procedure with OUT parameters # Pass None for OUT params; retrieve via the returned args tuple args = (7, None, None) # customer_id, order_count (OUT), total_spent (OUT) results = cursor.callproc('sp_customer_total', args) order_count, total_spent = results[1], results[2] cursor.close() conn.close()

PHP (PDO)

$pdo = new PDO('mysql:host=localhost;dbname=bookshop', 'admin', 'secret'); // Call a stored function — just use it in SQL $stmt = $pdo->prepare('SELECT fn_customer_segment(:ltv)'); $stmt->execute([':ltv' => 250.00]); $segment = $stmt->fetchColumn(); // Call a procedure with OUT params via user variables $pdo->exec('CALL sp_customer_total(7, @cnt, @total)'); $row = $pdo->query('SELECT @cnt AS order_count, @total AS total_spent')->fetch();

10. When to Use Stored Code vs Application Logic

Use stored procedures when: Multiple different applications (PHP web app, Python ETL, reporting tool) all need the same workflow — centralise it in the database so each caller doesn't re-implement it. Also good for batch jobs that run entirely inside the database, and for encapsulating privileged operations (a low-privilege app user can CALL a procedure without needing direct table access).
Use stored functions when: You have a reusable calculation or formatting rule that should work like a built-in MySQL function — customer segment labels, price formatting, age-from-date, business day calculation.
Avoid stored code when: Your team's primary language is not SQL (stored procedures are harder to test, version-control, and debug than application code). Complex business logic buried in procedures is difficult to unit test, impossible to profile with standard tools, and invisible to application-layer observability (tracing, logging). Prefer application code for logic that changes frequently or needs thorough test coverage.

Never put secrets or API calls in stored procedures. The database has no HTTP client, no secret manager, and no safe way to handle credentials. Business logic that needs to call external services belongs in application code, not the database.

Quick Reference

Concept	Syntax / rule	Notes
Delimiter	`DELIMITER $$` … `DELIMITER ;`	CLI / Workbench only. Not needed in application code.
Create procedure	`CREATE PROCEDURE name(IN p1 type, OUT p2 type) BEGIN … END`	Use `OR REPLACE` (MySQL 8.0.29+) or DROP first.
Call procedure	`CALL name(arg1, @out_var);`	OUT/INOUT params take user variables (prefixed @).
Read OUT result	`SELECT @out_var;`	User variables persist for the session after CALL returns.
Local variable	`DECLARE v_name type DEFAULT value;`	Must be first statements in BEGIN block. Scoped to block.
Assign variable	`SET v_name = expr;` or `SELECT col INTO v_name FROM …`	SELECT INTO assigns to local variables, not user variables.
Create function	`CREATE FUNCTION name(p type) RETURNS type DETERMINISTIC BEGIN … RETURN val; END`	Must declare DETERMINISTIC or NOT DETERMINISTIC.
Error handler	`DECLARE EXIT HANDLER FOR SQLEXCEPTION BEGIN ROLLBACK; END;`	EXIT exits the block; CONTINUE resumes after the error.
Cursor lifecycle	DECLARE → OPEN → FETCH loop (check NOT FOUND) → CLOSE	Declare NOT FOUND handler before opening cursor.
Flow control	`IF … ELSEIF … ELSE … END IF` · `CASE … END CASE` · `WHILE … END WHILE` · `REPEAT … UNTIL … END REPEAT` · labelled `LOOP … END LOOP`	`LEAVE label` = break · `ITERATE label` = continue
Inspect	`SHOW PROCEDURE STATUS WHERE Db = DATABASE();` · `SHOW CREATE PROCEDURE name;`	Also available via `information_schema.routines`.
Drop	`DROP PROCEDURE IF EXISTS name;` · `DROP FUNCTION IF EXISTS name;`	Always use IF EXISTS in scripts.

Next: Chapter 9 — Triggers and scheduled events: BEFORE/AFTER INSERT/UPDATE/DELETE triggers, CREATE EVENT, practical use cases and the dangers to watch for.

ADVANCED SQL — CHAPTER 9

Triggers and Scheduled Events

Chapter 9 — Triggers and Scheduled Events

Triggers are stored programs that MySQL runs *automatically* whenever a specific data-change event occurs on a table — no application code involved, no CALL needed. Scheduled events are stored programs that MySQL runs *on a clock* — once at a future time, or repeatedly on an interval.

Together they move certain kinds of logic directly into the database engine: audit trails, constraint enforcement beyond what CHECK can express, derived-column maintenance, and background maintenance tasks like expiring old rows or refreshing summary tables.

Power and responsibility: Triggers and events run invisibly to application code. A developer INSERT-ing a row has no obvious way to know a trigger is also writing to three other tables. Always document them, keep them focused, and treat the "magic invisible side effect" quality as a cost — not a feature — to be used sparingly.

Part 1 — Triggers

1. Trigger Syntax and Timing

CREATE [OR REPLACE] TRIGGER trigger_name { BEFORE | AFTER } { INSERT | UPDATE | DELETE } ON table_name FOR EACH ROW [{ FOLLOWS | PRECEDES } other_trigger_name] trigger_body

Each combination of timing (BEFORE/AFTER) and event (INSERT/UPDATE/DELETE) is a distinct trigger slot. MySQL 5.7+ allows multiple triggers per slot — ordered with FOLLOWS / PRECEDES.

	INSERT	UPDATE	DELETE
BEFORE	✓ Validate or modify data before the row is written	✓ Block invalid changes; normalise values	✓ Check business rules before the delete
AFTER	✓ Write to audit log; update summary tables	✓ Propagate changes to related tables	✓ Cascade soft-delete; archive removed rows

OLD and NEW pseudo-rows

INSERT
`NEW.col` — the value being inserted
`OLD` — does not exist
UPDATE
`NEW.col` — value after the update
`OLD.col` — value before the update
DELETE
`OLD.col` — the value being deleted
`NEW` — does not exist

In a BEFORE trigger you can assign to `NEW.col` to change the value that will actually be written. In an AFTER trigger `NEW` is read-only — the row is already committed.

2. Practical Trigger Patterns

Pattern 1 — Audit log (AFTER INSERT / UPDATE / DELETE)

First, create the audit table:

CREATE TABLE IF NOT EXISTS books_audit (audit_id INT PRIMARY KEY AUTO_INCREMENT, action ENUM('INSERT','UPDATE','DELETE') NOT NULL, book_id INT NOT NULL, old_price DECIMAL(10,2), new_price DECIMAL(10,2), old_stock INT, new_stock INT, changed_by VARCHAR(100) DEFAULT USER(), changed_at DATETIME DEFAULT NOW());

DELIMITER $$ -- AFTER INSERT CREATE OR REPLACE TRIGGER trg_books_after_insert AFTER INSERT ON books FOR EACH ROW BEGIN INSERT INTO books_audit (action, book_id, new_price, new_stock) VALUES ('INSERT', NEW.book_id, NEW.price, NEW.stock_quantity); END$$ -- AFTER UPDATE CREATE OR REPLACE TRIGGER trg_books_after_update AFTER UPDATE ON books FOR EACH ROW BEGIN -- Only log if price or stock actually changed IF NEW.price <> OLD.price OR NEW.stock_quantity <> OLD.stock_quantity THEN INSERT INTO books_audit (action, book_id, old_price, new_price, old_stock, new_stock) VALUES ('UPDATE', NEW.book_id, OLD.price, NEW.price, OLD.stock_quantity, NEW.stock_quantity); END IF; END$$ -- AFTER DELETE CREATE OR REPLACE TRIGGER trg_books_after_delete AFTER DELETE ON books FOR EACH ROW BEGIN INSERT INTO books_audit (action, book_id, old_price, old_stock) VALUES ('DELETE', OLD.book_id, OLD.price, OLD.stock_quantity); END$$ DELIMITER ;

Pattern 2 — Validate and normalise with BEFORE INSERT / UPDATE

BEFORE triggers can modify `NEW` values before they hit the table, or signal an error to reject the write entirely:

DELIMITER $$ CREATE OR REPLACE TRIGGER trg_books_before_insert BEFORE INSERT ON books FOR EACH ROW BEGIN -- Reject negative prices IF NEW.price < 0 THEN SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Price cannot be negative'; END IF; -- Clamp stock to zero if a negative value is supplied IF NEW.stock_quantity < 0 THEN SET NEW.stock_quantity = 0; END IF; -- Trim whitespace from title SET NEW.title = TRIM(NEW.title); END$$ CREATE OR REPLACE TRIGGER trg_books_before_update BEFORE UPDATE ON books FOR EACH ROW BEGIN IF NEW.price < 0 THEN SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Price cannot be negative'; END IF; IF NEW.stock_quantity < 0 THEN SET NEW.stock_quantity = 0; END IF; END$$ DELIMITER ;
SIGNAL SQLSTATE '45000' raises a user-defined error. `45000` is the reserved state for generic application errors. The error propagates back to the caller exactly like any other MySQL error — the INSERT or UPDATE is rolled back and the application receives an exception.

Pattern 3 — Maintain a derived column automatically

Keep a `full_name` column on the `authors` table in sync without relying on application code to remember:

-- Add the derived column (or use a generated column in MySQL 5.7+) ALTER TABLE authors ADD COLUMN full_name VARCHAR(200) AS (CONCAT(first_name, ' ', last_name)) STORED; -- MySQL's generated column syntax makes this specific example redundant, -- but the same BEFORE trigger pattern applies to any non-trivial derivation: DELIMITER $$ CREATE OR REPLACE TRIGGER trg_orders_before_insert BEFORE INSERT ON orders FOR EACH ROW BEGIN -- Auto-compute order total from line items already in a staging table -- (illustrative pattern — adapt to your schema) SET NEW.total_amount = (SELECT COALESCE(SUM(quantity * unit_price), 0) FROM order_items_staging WHERE session_id = NEW.session_id); END$$ DELIMITER ;

Pattern 4 — Cascade soft-delete (AFTER DELETE)

When a customer is deleted, archive their orders rather than hard-deleting them:

CREATE TABLE IF NOT EXISTS orders_archive LIKE orders; ALTER TABLE orders_archive ADD COLUMN archived_at DATETIME DEFAULT NOW(), ADD COLUMN archived_reason VARCHAR(200); DELIMITER $$ CREATE OR REPLACE TRIGGER trg_customers_after_delete AFTER DELETE ON customers FOR EACH ROW BEGIN -- Move the deleted customer's orders into the archive INSERT INTO orders_archive (order_id, customer_id, order_date, status, total_amount, archived_reason) SELECT order_id, customer_id, order_date, status, total_amount, CONCAT('Customer ', OLD.customer_id, ' deleted') FROM orders WHERE customer_id = OLD.customer_id; END$$ DELIMITER ;

Pattern 5 — Maintain a running stock count on order_items

DELIMITER $$ -- Decrement stock when an order line is inserted CREATE OR REPLACE TRIGGER trg_order_items_after_insert AFTER INSERT ON order_items FOR EACH ROW BEGIN UPDATE books SET stock_quantity = stock_quantity - NEW.quantity WHERE book_id = NEW.book_id; END$$ -- Restore stock when an order line is deleted (cancellation) CREATE OR REPLACE TRIGGER trg_order_items_after_delete AFTER DELETE ON order_items FOR EACH ROW BEGIN UPDATE books SET stock_quantity = stock_quantity + OLD.quantity WHERE book_id = OLD.book_id; END$$ -- Handle quantity changes on existing lines CREATE OR REPLACE TRIGGER trg_order_items_after_update AFTER UPDATE ON order_items FOR EACH ROW BEGIN IF NEW.quantity <> OLD.quantity THEN UPDATE books SET stock_quantity = stock_quantity - (NEW.quantity - OLD.quantity) WHERE book_id = NEW.book_id; END IF; END$$ DELIMITER ;

3. Trigger Dangers

Triggers are invisible to application developers. An INSERT that takes 50ms instead of 1ms, or a DELETE that inexplicably fails a foreign key constraint on a different table, is extremely difficult to debug if the developer doesn't know a trigger exists. Always document triggers prominently.

- **No triggers on triggers.** A trigger cannot directly fire another trigger on the same table. It can fire a trigger on a *different* table — creating cascade chains that are very hard to trace.

- **Triggers slow down writes.** Every affected row runs the trigger body. A BEFORE UPDATE trigger on a table that receives 10,000 updates per second is now 10,000 extra operations per second.
- **Bulk operations pay per row.** `UPDATE books SET price = price * 1.1` with an AFTER UPDATE trigger runs the trigger once per matching row — not once for the whole statement. A million-row update fires the trigger a million times.
- **SIGNAL is your friend for validation.** Without SIGNAL, a BEFORE trigger that silently discards an invalid value gives the caller no feedback. Always SIGNAL for true constraint violations.
- **Avoid long-running work in triggers.** Triggers run inside the originating transaction. Any work they do holds the same locks. A trigger that sends an HTTP request or sleeps will block other transactions for the duration.

Test triggers with SHOW TRIGGERS and manual testing. After writing a trigger, explicitly INSERT/UPDATE/DELETE a test row and verify the side effect. Triggers with a bug fail silently in surprising ways — the original write may still succeed while the trigger's secondary write silently errors (unless you have a handler or the error propagates).

4. Inspecting and Dropping Triggers

-- List all triggers on the current database SHOW TRIGGERS [FROM bookshop]\G -- Filter by table SHOW TRIGGERS WHERE `Table` = 'books'\G -- Full information_schema query SELECT trigger_name, event_manipulation AS event, action_timing AS timing, event_object_table AS `table`, action_order, created FROM information_schema.triggers WHERE trigger_schema = DATABASE() ORDER BY event_object_table, action_timing, event_manipulation, action_order; -- Show a trigger's body SHOW CREATE TRIGGER trg_books_after_update\G -- Drop a trigger DROP TRIGGER IF EXISTS trg_books_after_insert; DROP TRIGGER IF EXISTS trg_books_after_update; DROP TRIGGER IF EXISTS trg_books_after_delete;

Part 2 — Scheduled Events

5. The Event Scheduler

MySQL's event scheduler is a background thread that fires stored SQL on a clock. Think of it as a cron job that lives inside the database — no OS-level cron, no external script, no network connection needed.

-- Check scheduler status SHOW VARIABLES LIKE 'event_scheduler'; -- Enable it for this session (or add to my.cnf to persist across restarts) SET GLOBAL event_scheduler = ON; -- Persist across restarts — add to [mysqld] section in my.cnf: -- event_scheduler = ON
The scheduler is OFF by default. Events defined while it is off are stored but never fire. Always verify `event_scheduler = ON` in production. On managed databases (AWS RDS, Google Cloud SQL) the parameter group or flags control this — check your provider's docs.

One-time vs recurring events

One-time (AT)

Fires once at a specific date/time. MySQL can optionally drop it automatically afterwards with `ON COMPLETION NOT PRESERVE`.

Recurring (EVERY)

Fires on a repeating interval — seconds, minutes, hours, days, weeks, months. Can have an optional start time and end time.

CREATE [OR REPLACE] EVENT [IF NOT EXISTS] event_name ON SCHEDULE { AT timestamp | EVERY interval [STARTS timestamp] [ENDS timestamp] } [ON COMPLETION [NOT] PRESERVE] [ENABLE | DISABLE] [COMMENT 'description'] DO event_body;

6. Practical Event Patterns

Pattern 1 — Nightly sales summary table

Refresh a denormalised summary table every night at 02:00 so daytime reports hit a pre-computed table instead of joining millions of order rows:

CREATE TABLE IF NOT EXISTS daily_sales_summary (summary_date DATE PRIMARY KEY, orders_placed INT NOT NULL DEFAULT 0, units_sold INT NOT NULL DEFAULT 0, revenue DECIMAL(12,2) NOT NULL DEFAULT 0.00, refreshed_at DATETIME); DELIMITER $$ CREATE OR REPLACE EVENT evt_nightly_sales_summary ON SCHEDULE EVERY 1 DAY STARTS (DATE(NOW()) + INTERVAL 1 DAY + INTERVAL 2 HOUR) -- next 02:00 ON COMPLETION PRESERVE -- keep the event definition after it fires ENABLE COMMENT 'Rebuild daily_sales_summary for the previous day' DO BEGIN INSERT INTO daily_sales_summary (summary_date, orders_placed, units_sold, revenue, refreshed_at) SELECT DATE(o.order_date) AS summary_date, COUNT(DISTINCT o.order_id) AS orders_placed, SUM(oi.quantity) AS units_sold, SUM(oi.quantity * oi.unit_price) AS revenue, NOW() FROM orders o JOIN order_items oi ON oi.order_id = o.order_id WHERE DATE(o.order_date) = CURDATE() - INTERVAL 1 DAY ON

DUPLICATE KEY UPDATE orders_placed = VALUES(orders_placed), units_sold = VALUES(units_sold), revenue = VALUES(revenue), refreshed_at = VALUES(refreshed_at); END$$ DELIMITER ;

Pattern 2 — Hourly expired session cleanup

CREATE OR REPLACE EVENT evt_purge_expired_sessions ON SCHEDULE EVERY 1 HOUR ON COMPLETION PRESERVE COMMENT 'Delete login sessions older than 24 hours' DO DELETE FROM user_sessions WHERE created_at < NOW() - INTERVAL 24 HOUR;
Batch deletes in events. If the table is large, a single DELETE that wipes millions of rows holds a heavy lock. Use a loop procedure called from the event to delete in batches of a few thousand rows at a time, sleeping briefly between each batch.

Pattern 3 — One-time future job

-- Run a specific task at exactly one future moment then discard the event CREATE EVENT evt_launch_day_price_reset ON SCHEDULE AT '2026-09-01 09:00:00' ON COMPLETION NOT PRESERVE -- auto-drop after it fires DO UPDATE books SET price = price * 1.05 -- 5% price rise on 1 Sept WHERE publisher_id = 3;

Pattern 4 — Weekly review request emails (via a log table)

Events can't send emails directly — but they can write to a queue table that an external process polls:

CREATE TABLE IF NOT EXISTS email_queue (queue_id INT PRIMARY KEY AUTO_INCREMENT, to_email VARCHAR(200) NOT NULL, subject VARCHAR(255) NOT NULL, body TEXT, queued_at DATETIME DEFAULT NOW(), sent_at DATETIME); CREATE OR REPLACE EVENT evt_weekly_review_requests ON SCHEDULE EVERY 1 WEEK STARTS '2026-06-16 08:00:00' -- first Monday ON COMPLETION PRESERVE DO -- Queue a nudge for customers who bought something in the last week -- but haven't left a review INSERT INTO email_queue (to_email, subject) SELECT DISTINCT c.email, 'How was your recent purchase from the Bookshop?' FROM customers c JOIN orders o ON o.customer_id = c.customer_id JOIN order_items oi ON oi.order_id = o.order_id LEFT JOIN reviews r ON r.book_id = oi.book_id AND r.customer_id = c.customer_id WHERE o.order_date >= CURDATE() - INTERVAL 7 DAY AND r.review_id IS NULL;

7. Managing Events

-- List all events SHOW EVENTS [FROM bookshop]\G -- Detailed query via information_schema SELECT event_name, status, execute_at, interval_value, interval_field, starts, ends, last_executed, event_comment FROM information_schema.events WHERE event_schema = DATABASE() ORDER BY event_name; -- See the full event body SHOW CREATE EVENT evt_nightly_sales_summary\G -- Disable an event (keep definition, stop firing) ALTER EVENT evt_nightly_sales_summary DISABLE; -- Re-enable ALTER EVENT evt_nightly_sales_summary ENABLE; -- Change schedule ALTER EVENT evt_purge_expired_sessions ON SCHEDULE EVERY 30 MINUTE; -- Drop DROP EVENT IF EXISTS evt_launch_day_price_reset;

Quick Reference

Concept	Syntax / rule	Notes
Create trigger	`CREATE OR REPLACE TRIGGER name {BEFORE|AFTER} {INSERT|UPDATE|DELETE} ON table FOR EACH ROW BEGIN … END`	One trigger per timing+event combination per table (multiple allowed with FOLLOWS/PRECEDES).
NEW / OLD	`NEW.col` (INSERT, UPDATE) · `OLD.col` (UPDATE, DELETE)	Assign to NEW in BEFORE trigger to change the written value. NEW is read-only in AFTER.
Raise an error	`SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = '…';`	Rolls back the triggering statement. Always SIGNAL for genuine constraint violations.
Trigger ordering	`FOLLOWS other_trigger` / `PRECEDES other_trigger`	Controls execution order when multiple triggers exist for the same event. MySQL 5.7+.
Inspect triggers	`SHOW TRIGGERS [FROM db];` · `SHOW CREATE TRIGGER name;`	Also available in `information_schema.triggers`.
Drop trigger	`DROP TRIGGER IF EXISTS name;`	Triggers are dropped with the table when the table is dropped.
Enable scheduler	`SET GLOBAL event_scheduler = ON;` · add `event_scheduler=ON` to my.cnf	Off by default. Events are stored but never fire while it is off.
One-time event	`ON SCHEDULE AT 'YYYY-MM-DD HH:MM:SS'`	Use `ON COMPLETION NOT PRESERVE` to auto-drop after firing.

Concept	Syntax / rule	Notes
Recurring event	`ON SCHEDULE EVERY n {SECOND|MINUTE|HOUR|DAY|WEEK|MONTH|YEAR} [STARTS …] [ENDS …]`	Use `ON COMPLETION PRESERVE` so the definition survives after each run.
Disable / enable event	`ALTER EVENT name DISABLE;` · `ALTER EVENT name ENABLE;`	Useful during maintenance windows or deployments.
Change schedule	`ALTER EVENT name ON SCHEDULE EVERY 30 MINUTE;`	ALTER EVENT can also change the body, comment, and status.
Drop event	`DROP EVENT IF EXISTS name;`	Always IF EXISTS in scripts.

Next: Chapter 10 — Query optimisation: EXPLAIN and EXPLAIN ANALYZE output, index strategy (composite, covering, prefix indexes), avoiding full table scans, and profiling with the slow query log.

ADVANCED SQL — CHAPTER 10

Query Optimisation

Chapter 10 — Query Optimisation

Every query you write goes through three stages: the parser checks syntax, the optimiser decides *how* to execute it (which indexes to use, what join order to pick, whether to use a temporary table), and the executor runs the chosen plan. The optimiser is very good — but it can only work with the indexes you give it and the SQL you write. This chapter teaches you to read its mind, feed it what it needs, and spot the patterns that defeat it.

The golden rule of optimisation: measure first, then fix. Gut-feel optimisation creates complexity without guaranteed gain. Use EXPLAIN to confirm there's a problem, the slow query log to find which queries need attention, and benchmark before and after any change.

1. EXPLAIN — Reading the Query Plan

Prefix any SELECT (or DML statement in MySQL 8.0+) with `EXPLAIN` to see the execution plan the optimiser chose — without actually running the query.

EXPLAIN SELECT b.title, CONCAT(a.first_name, ' ', a.last_name) AS author, COUNT(r.review_id) AS reviews FROM books b JOIN book_authors ba ON ba.book_id = b.book_id JOIN authors a ON a.author_id = ba.author_id LEFT JOIN reviews r ON r.book_id = b.book_id GROUP BY b.book_id, b.title, a.first_name, a.last_name ORDER BY reviews DESC\G

The key columns you need to read

Column	What it tells you	Red flags
type	The access method (see graded list below)	ALL (full table scan), index (full index scan)
key	The index actually chosen (NULL if none)	NULL on a large table
key_len	Bytes of the index used — tells you how many columns of a composite index were used	Much shorter than expected → composite index not fully used
rows	Estimated rows examined per outer row — the optimiser's cost estimate	Very high relative to table size
filtered	Estimated % of rows that pass the WHERE after index access	Low (e.g. 5%) means index is poor for this filter
Extra	Additional plan details	Using filesort, Using temporary — expensive operations
possible_keys	Indexes the optimiser considered	NULL means no usable index exists for this table

The type column — access method graded best to worst

const / eq_ref
Best. At most one row matched — primary key or unique index equality. As fast as a lookup can be.
ref
Good. Non-unique index equality — returns a small set of matching rows. Common on foreign key columns.
range
Acceptable. Index range scan — WHERE col BETWEEN, col > val, col IN (...). Reads a contiguous slice of the index.
index
Marginal. Scans the entire index tree (cheaper than a table scan, but still full scan). Often appears when a covering index is used.
ALL
Bad. Full table scan — reads every row. Acceptable only on tiny tables (<a few hundred rows). Investigate immediately on large tables.

Extra column — what the flags mean

Extra value	Meaning	Fix
Using index	Covering index — no table row read needed	Great. No action needed.
Using where	WHERE filter applied after index access	Normal. Fine unless rows is very high.
Using filesort	ORDER BY couldn't use an index — sort in memory/disk	Add an index that covers the ORDER BY columns.

Extra value	Meaning	Fix
Using temporary	Intermediate result stored in a temp table (GROUP BY, DISTINCT, some subqueries)	Add an index covering the GROUP BY columns, or rewrite the query.
Using filesort + Using temporary	Sort after a temp table — double penalty	High priority to fix. Redesign index or query structure.
Using index condition	Index condition pushdown — filter applied inside the storage engine	Good. Engine filters before returning rows.

2. EXPLAIN ANALYZE — Actual Execution Stats

`EXPLAIN ANALYZE` (MySQL 8.0.18+) actually runs the query and returns the plan annotated with real timing and row counts alongside the estimates. Use it to confirm whether the optimiser's estimates match reality.

EXPLAIN ANALYZE SELECT b.title, COUNT(r.review_id) AS reviews FROM books b LEFT JOIN reviews r ON r.book_id = b.book_id GROUP BY b.book_id ORDER BY reviews DESC\G
-- Typical EXPLAIN ANALYZE output (tree format) -> Sort: reviews DESC (actual time=12.4..12.5 rows=84 loops=1) -> Table scan on <temporary> (actual time=0.0..0.4 rows=84 loops=1) -> Aggregate using temporary table (actual time=11.1..11.8 rows=84 loops=1) -> Nested loop left join (actual time=0.2..8.4 rows=312 loops=1) -> Table scan on b (cost=9.3 rows=84) (actual time=0.1..0.5 rows=84 loops=1) -> Index lookup on r using idx_reviews_book_id (book_id=b.book_id) (cost=0.9 rows=3) (actual time=0.06..0.09 rows=3 loops=84)
Compare estimated vs actual rows. When the optimiser estimates 3 rows but actually processes 30,000, its cost model is wrong — usually because table statistics are stale. Run `ANALYZE TABLE tablename;` to refresh them.

3. Index Strategy

Single-column index
The baseline. Index one column used frequently in WHERE, JOIN, or ORDER BY. InnoDB primary key is always a clustered index — the table rows are physically ordered by it.
Composite index
Index on (col_a, col_b, col_c). Left-prefix rule: queries must use col_a to benefit, then optionally col_b, then col_c. Skipping a column in the middle breaks the chain.
Covering index
A composite index that includes all columns the query needs — SELECT list + WHERE + ORDER BY. The engine never touches the table rows — EXPLAIN shows "Using index."
Prefix index
Index on the first N characters of a TEXT/VARCHAR column: `INDEX (email(20))`. Saves space but cannot be a covering index and cannot be used for ORDER BY.

Adding indexes to the bookshop schema

-- Foreign key columns (if not indexed automatically) CREATE INDEX idx_orders_customer ON orders (customer_id); CREATE INDEX idx_order_items_order ON order_items (order_id); CREATE INDEX idx_order_items_book ON order_items (book_id); CREATE INDEX idx_reviews_book ON reviews (book_id); CREATE INDEX idx_reviews_customer ON reviews (customer_id); -- Composite: date range reports that also filter by customer CREATE INDEX idx_orders_customer_date ON orders (customer_id, order_date); -- Covering: the catalogue listing query touches only these columns CREATE INDEX idx_books_cover ON books (publisher_id, price, title); -- Prefix: email is long, only the first 20 chars are usually enough CREATE INDEX idx_customers_email_pfx ON customers (email(20));

The left-prefix rule — composite index column order matters

-- Composite index: (customer_id, order_date) -- ✅ Uses index — leads with customer_id SELECT * FROM orders WHERE customer_id = 7 AND order_date >= '2026-01-01'; -- ✅ Uses index (partial) — customer_id alone still benefits SELECT * FROM orders WHERE customer_id = 7; -- ❌ Cannot use index — skips the leading column SELECT * FROM orders WHERE order_date >= '2026-01-01';

Covering index — eliminate the table row lookup

-- Index: (publisher_id, price, title) -- Query only touches those three columns → "Using index" in EXPLAIN SELECT title, price FROM books WHERE publisher_id = 3 ORDER BY price;

Index selectivity — not all columns deserve an index

Selectivity = distinct values ÷ total rows. A column with low selectivity (e.g. a boolean, or a status column with 3 values) is a poor index candidate — the optimiser may prefer a full table scan anyway because too many rows match.

-- Check selectivity before creating an index SELECT COUNT(DISTINCT status) AS distinct_values, COUNT(*) AS total_rows, ROUND(COUNT(DISTINCT status) / COUNT(*), 4) AS selectivity FROM orders; -- selectivity of 0.0003 → poor index candidate alone -- combine with a high-selectivity column in a composite index instead

4. Anti-Patterns That Defeat Indexes

1 — Function on the indexed column

-- ❌ YEAR() wraps the column — index on order_date is useless SELECT * FROM orders WHERE YEAR(order_date) = 2026; -- ✅ Rewrite as a range — index is used SELECT * FROM orders WHERE order_date >= '2026-01-01' AND order_date < '2027-01-01';
-- ❌ LOWER() on the column defeats the index SELECT * FROM customers WHERE LOWER(email) = 'alice@example.com'; -- ✅ Store emails lowercased, or use a case-insensitive collation SELECT * FROM customers WHERE email = 'alice@example.com'; -- (works if email column uses utf8mb4_unicode_ci collation)

2 — Implicit type cast

-- ❌ isbn is VARCHAR — passing an integer forces a cast on every row SELECT * FROM books WHERE isbn = 9780142000670; -- ✅ Pass the correct type SELECT * FROM books WHERE isbn = '9780142000670';

3 — Leading wildcard LIKE

-- ❌ Leading % means MySQL cannot use a B-tree index SELECT * FROM books WHERE title LIKE '%foundation%'; -- ✅ Trailing wildcard only — index is used SELECT * FROM books WHERE title LIKE 'Foundation%'; -- For full-text search, use FULLTEXT index + MATCH ... AGAINST ALTER TABLE books ADD FULLTEXT INDEX ft_title (title); SELECT * FROM books WHERE MATCH(title) AGAINST ('foundation' IN BOOLEAN MODE);

4 — OR across different columns

-- ❌ OR on two columns — optimiser often falls back to full scan SELECT * FROM books WHERE title = 'Dune' OR isbn = '9780441013593'; -- ✅ Rewrite as UNION — each branch can use its own index SELECT * FROM books WHERE title = 'Dune' UNION SELECT * FROM books WHERE isbn = '9780441013593';

5 — Negation operators

-- ❌ != / NOT IN / NOT LIKE rarely use indexes efficiently SELECT * FROM orders WHERE status != 'pending'; -- ✅ Rewrite positively when the positive set is small SELECT * FROM orders WHERE status IN ('shipped', 'delivered', 'cancelled');

6 — Non-deterministic functions in WHERE

-- ❌ NOW() is evaluated per row in some contexts — use a parameter instead SELECT * FROM orders WHERE order_date >= NOW() - INTERVAL 30 DAY; -- This is actually fine — NOW() is a constant for the query duration -- but DATE(order_date) > ... wraps the column — avoid that form

5. The Slow Query Log

The slow query log records every query that takes longer than a threshold you set. It's the most reliable way to find real problems in production — not guessed ones.

Enable the slow query log

-- Check current status SHOW VARIABLES LIKE 'slow_query%'; SHOW VARIABLES LIKE 'long_query_time'; -- Enable for this session (doesn't persist across restart) SET GLOBAL slow_query_log = 'ON'; SET GLOBAL long_query_time = 1; -- log queries > 1 second SET GLOBAL slow_query_log_file = '/var/log/mysql/slow.log'; -- Also log queries that perform a full table scan (no index used) SET GLOBAL log_queries_not_using_indexes = 'ON'; # Add to /etc/mysql/mysql.conf.d/mysqld.cnf for persistence: # [mysqld] # slow_query_log = 1 # slow_query_log_file = /var/log/mysql/slow.log # long_query_time = 1 # log_queries_not_using_indexes = 1

Reading the slow query log with mysqldumpslow

Summarise the top 10 slowest query patterns mysqldumpslow -s t -t 10 /var/log/mysql/slow.log # -s t = sort by total time # -s c = sort by call count (finds frequently-run slow queries) # -t 10 = top 10

Typical slow query log entry # Time: 2026-06-15T09:14:22 # User@Host: admin[admin] @ localhost [] # Query_time: 3.842134 Lock_time: 0.000120 # Rows_sent: 1 Rows_examined: 487503 # SET timestamp=1750000462; # SELECT * FROM orders WHERE YEAR(order_date) = 2026;

Rows_examined vs Rows_sent is your key ratio. Examining 487,503 rows to send 1 means the query is doing an enormous amount of unnecessary work. A well-indexed query should examine close to the number of rows it returns.

6. Live Diagnosis

SHOW PROCESSLIST — what's running right now

SHOW FULL PROCESSLIST\G -- Kill a stuck query (replace 42 with the Id from PROCESSLIST) KILL QUERY 42; -- kills the query, keeps the connection KILL 42; -- kills query AND disconnects

Key SHOW STATUS counters

-- Reset counters then run your workload, then check FLUSH STATUS; -- After running queries: SHOW STATUS LIKE 'Handler_%'; -- Important counters: -- Handler_read_rnd_next — rows read by full table scan (should be low) -- Handler_read_key — index lookups (higher is better) -- Handler_read_next — index range scan rows (acceptable) -- Created_tmp_disk_tables — temp tables spilled to disk (should be 0) -- Select_full_join — joins with no index (should be 0) SHOW STATUS LIKE 'Select_full%'; SHOW STATUS LIKE 'Created_tmp%';

ANALYZE TABLE — refresh statistics for the optimiser

-- After large INSERT/DELETE/UPDATE batches, statistics can become stale ANALYZE TABLE orders, order_items, books; -- Check when a table was last analyzed SELECT table_name, update_time, table_rows FROM information_schema.tables WHERE table_schema = DATABASE() ORDER BY update_time DESC;

7. A Complete Optimisation Walk-Through

A slow report query has been flagged in the slow query log — 4.2 seconds, examining 620,000 rows, returning 28. Here's the full diagnosis and fix cycle:

Step 1: run EXPLAIN

EXPLAIN SELECT DATE_FORMAT(o.order_date, '%Y-%m') AS month, COUNT(DISTINCT o.order_id) AS orders, SUM(oi.quantity * oi.unit_price) AS revenue FROM orders o JOIN order_items oi ON oi.order_id = o.order_id WHERE YEAR(o.order_date) = 2026 GROUP BY month ORDER BY month\G

table	type	key	rows	Extra
o	ALL	NULL	420000	Using where; Using temporary; Using filesort
oi	ref	idx_order_items_order	3	NULL

Diagnoses: (1) YEAR() wraps the column — index on order_date is bypassed. (2) No index exists for ORDER BY month. (3) GROUP BY forces a temporary table.

Step 2: fix the query and add an index

-- Fix 1: rewrite the date filter as a range -- Fix 2: add a composite index for date + order_id to cover the GROUP BY CREATE INDEX idx_orders_date_id ON orders (order_date, order_id); -- Rewritten query SELECT DATE_FORMAT(o.order_date, '%Y-%m') AS month, COUNT(DISTINCT o.order_id) AS orders, SUM(oi.quantity * oi.unit_price) AS revenue FROM orders o JOIN order_items oi ON oi.order_id = o.order_id WHERE o.order_date >= '2026-01-01' AND o.order_date < '2027-01-01' GROUP BY month ORDER BY month;

Step 3: verify with EXPLAIN

table	type	key	rows	Extra
o	range	idx_orders_date_id	28	Using where; Using index
oi	ref	idx_order_items_order	3	NULL

Rows examined dropped from 420,000 to 28. "Using temporary" and "Using filesort" are gone. Query time: 12ms.

Quick Reference

Tool / concept	Command / rule	What to act on
EXPLAIN	`EXPLAIN SELECT …`	type = ALL, key = NULL, Extra = Using filesort / Using temporary
EXPLAIN ANALYZE	`EXPLAIN ANALYZE SELECT …` (8.0.18+)	Large gap between estimated and actual rows → run ANALYZE TABLE
Access types (best→worst)	const → eq_ref → ref → range → index → ALL	Investigate anything at index or ALL on tables > a few hundred rows
Composite index order	Put equality columns first, range/ORDER BY column last	Left-prefix rule: skipping a column breaks index use for subsequent columns
Covering index	Include SELECT + WHERE + ORDER BY cols in one index	EXPLAIN shows "Using index" — no table row read. Fastest possible read path.
Function on column	WHERE YEAR(col) → WHERE col BETWEEN …	Any function wrapping an indexed column defeats the index
Leading wildcard	LIKE '%term' → use FULLTEXT index + MATCH … AGAINST	B-tree indexes require a known prefix
NOT IN with NULLs	Add `WHERE col IS NOT NULL` inside subquery	Any NULL in the list silently empties the NOT IN result
Slow query log	`SET GLOBAL slow_query_log = ON; SET GLOBAL long_query_time = 1;`	High Rows_examined vs Rows_sent ratio → missing or wrong index
mysqldumpslow	`mysqldumpslow -s t -t 10 /path/slow.log`	Find the top time-wasting query patterns across thousands of log entries
Refresh statistics	`ANALYZE TABLE tbl;`	Run after large bulk operations; stale stats cause poor plan choices
SHOW PROCESSLIST	`SHOW FULL PROCESSLIST\G`	Long-running queries holding locks; kill with `KILL QUERY id`
Selectivity check	`COUNT(DISTINCT col) / COUNT(*)`	Values near 0 → poor standalone index candidate; combine in composite or skip

COURSE COMPLETE

You've covered all 10 chapters of the Advanced SQL course — from normalisation and every join type through subqueries, CTEs, window functions, views, stored procedures, triggers, events, and query optimisation. The skills in this course represent the full professional SQL toolkit used in production databases worldwide.