SQL · RELATIONAL DATABASES · MYSQL 8

MySQL Foundations

A practical introduction to SQL and relational database design

COURSE CONTENTS

osztromok.com · MySQL Foundations

Before writing a single line of SQL, it helps to understand what a relational database actually is and why it's structured the way it is. This chapter covers the core ideas — tables, rows, columns, keys, and relationships — and introduces the **bookshop database** that we'll build on throughout this entire course.

Course database. Every chapter in this course uses the same `bookshop` database. We'll build it up gradually — this chapter creates the schema, and subsequent chapters fill it with data and query it. By the end of the course you'll know it well, which makes the SQL examples easy to follow.

A **relational database** stores data in **tables** — a structure you can think of as a spreadsheet with strict rules. Each table holds one type of thing (books, customers, orders), and the tables are linked together using **relationships**. That linking is the "relational" part.

Here's a simple example — a table of books:

BOOK_ID	TITLE	AUTHOR	GENRE	PRICE
1	The Midnight Library	Matt Haig	Fiction	£8.99
2	Dune	Frank Herbert	Sci-Fi	£9.99
3	Sapiens	Yuval Noah Harari	Non-fiction	£10.99

Each horizontal line is a **row** (also called a record or tuple). Each vertical column holds one piece of information. Every row in this table describes exactly one book — no more, no less.

Table

A named collection of related data, organised into rows and columns. A database contains many tables. Think of it

Row (Record)

A single entry in a table — one book, one customer, one order. Every row in a table has the same set of columns, but different values.

as one sheet in a spreadsheet — but with strict rules about what data is allowed.

Column (Field)

One attribute of the thing being described — the book's title, its price, its genre. Every column has a name and a **data type** (text, number, date) that constrains what values it can hold.

Cell

The intersection of a row and a column — a single value. The cell at row 2, column "title" contains "Dune". Cells can also be `NULL`, meaning the value is unknown or not applicable.

Every table needs a way to uniquely identify each row. That's what a **primary key** does. In the books table above, `book_id` is the primary key — no two books share the same ID, and every book has one.

Primary keys have three rules:

- **Unique** — no two rows can have the same primary key value.
- **Not NULL** — every row must have a value; it can't be empty.
- **Stable** — once set, the key shouldn't change. Book 2 is always book 2.

Use surrogate keys, not natural keys. It's tempting to use something that's "naturally unique" — an ISBN for a book, a passport number for a person — as a primary key. These are called *natural keys*. The problem is that natural keys can change (a book gets a new edition and a new ISBN), may not exist for all rows, or may be longer than needed. A simple auto-incrementing integer (`book_id = 1, 2, 3…`) called a *surrogate key* is almost always the better choice.

In MySQL, the most common pattern is an integer column that increments automatically:

```sql
-- AUTO_INCREMENT creates a new unique ID for every inserted row
CREATE TABLE books (
    book_id   INT           NOT NULL AUTO_INCREMENT,
    title     VARCHAR(255)  NOT NULL,
    price     DECIMAL(6,2) NOT NULL,
    PRIMARY KEY (book_id)
);
```

You don't insert the `book_id` — MySQL generates it automatically when you add a new row. The first book gets ID 1, the next gets ID 2, and so on, forever upwards.

A single flat table can only get you so far. The power of a relational database comes from linking tables together. Consider the author problem: if we store the author's name inside the books table, we duplicate it for every book they've written — and if we need to correct a spelling, we'd have to update every row.

The solution is to put authors in their own table and link to them with a **foreign key**:

AUTHOR_ID	FIRST_NAME	LAST_NAME
1	Matt	Haig
2	Frank	Herbert
3	Yuval Noah	Harari

BOOK_ID	TITLE	AUTHOR_ID ↗	GENRE	PRICE
1	The Midnight Library	1	Fiction	8.99
2	Dune	2	Sci-Fi	9.99
3	Sapiens	3	Non-fiction	10.99

The `author_id` column in the books table is a **foreign key** — it refers to the primary key of the authors table. MySQL can enforce this relationship: if you try to insert a book with `author_id = 99` and there's no author with ID 99, MySQL will reject the insert. This is called **referential integrity**.

One-to-Many

The most common relationship. One author can write many books, but each book has one author. The "many" side (books) holds the foreign key pointing back to the "one" side (authors).

Many-to-Many

A customer can order many books; a book can be ordered by many customers. This requires a third *junction table* (e.g. `order_items`) that links both sides. Covered in the Advanced course.

One-to-One

Rare — used to split a table for security or performance reasons. Example: a `customers` table and a separate `customer_payment_details` table, linked 1:1.

Referential Integrity

MySQL's guarantee that foreign key values always point to a real row. You can't delete an author that still has books, and you can't add a book pointing to a non-existent author.

SQL (Structured Query Language, pronounced "sequel" or "S-Q-L") is the language you use to talk to a relational database. You write a statement describing *what data you want* — not *how to find it*. The database engine figures out the how.

1. **You write a SQL statement**
   e.g. `SELECT title, price FROM books WHERE price < 10.00;`
2. **MySQL parses and validates it**
   Checks syntax is correct, that the tables and columns you named exist, and that you have permission to access them.
3. **The query optimiser builds an execution plan**
   MySQL decides the most efficient way to retrieve the data — which indexes to use, what order to check conditions in. You don't control this directly.
4. **MySQL executes the plan and returns a result set**
   The matching rows are returned to your client (MySQL Workbench, phpMyAdmin, your PHP script, etc.) as a table of results.

SQL is divided into sub-languages by what kind of statement you're writing:

DQL — Data Query Language

`SELECT` — reading data. The vast majority of SQL you'll write. Covered in Chapters 2, 3, 8, and 9.

DML — Data Manipulation Language

`INSERT`, `UPDATE`, `DELETE` — creating, modifying, and removing rows. Covered in Chapter 5.

DDL — Data Definition Language

`CREATE`, `ALTER`, `DROP` — defining and changing the structure of tables. Covered in Chapter 6.

TCL — Transaction Control Language

`BEGIN`, `COMMIT`, `ROLLBACK` — grouping statements into atomic units of work. Covered in Chapter 10.

Throughout this course we'll work with a fictional online bookshop. It's simple enough to understand at a glance but realistic enough to demonstrate every concept properly. Here are the four core tables:

AUTHORS

author_id	INT	PK
first_name	VARCHAR(100)	NN
last_name	VARCHAR(100)	NN
nationality	VARCHAR(60)	
birth_year	YEAR	

BOOKS

book_id	INT	PK
author_id	INT	FK
title	VARCHAR(255)	NN
genre	VARCHAR(60)	
published_year	YEAR	
price	DECIMAL(6,2)	NN
stock	INT	

CUSTOMERS

customer_id	INT	PK
first_name	VARCHAR(100)	NN
last_name	VARCHAR(100)	NN
email	VARCHAR(150)	NN
joined_date	DATE	
city	VARCHAR(80)	

ORDERS

order_id	INT	PK
customer_id	INT	FK
book_id	INT	FK
order_date	DATETIME	NN
quantity	INT	NN
total_price	DECIMAL(8,2)	NN

■ PK = Primary Key · ■ FK = Foreign Key · ■ NN = Not Null

The relationships: one author → many books; one customer → many orders; one book → many orders. The `orders` table sits at the centre, linking customers to books they've purchased.

Let's build it. Run these statements in MySQL Workbench, phpMyAdmin, or the MySQL command-line client to create the database and its four tables:

```sql
-- Create the database and switch to it
CREATE DATABASE bookshop
    CHARACTER SET utf8mb4
    COLLATE utf8mb4_unicode_ci;

USE bookshop;
```

```sql
-- Authors table (no foreign keys — it's the "parent" table)
CREATE TABLE authors (
    author_id   INT           NOT NULL AUTO_INCREMENT,
    first_name  VARCHAR(100) NOT NULL,
    last_name   VARCHAR(100) NOT NULL,
    nationality VARCHAR(60),
    birth_year  YEAR,
    PRIMARY KEY (author_id)
);
```

```sql
-- Books table (references authors)
CREATE TABLE books (
    book_id        INT            NOT NULL AUTO_INCREMENT,
    author_id      INT            NOT NULL,
    title          VARCHAR(255)  NOT NULL,
    genre          VARCHAR(60),
    published_year YEAR,
    price          DECIMAL(6,2)  NOT NULL,
    stock          INT            DEFAULT 0,
    PRIMARY KEY (book_id),
    FOREIGN KEY (author_id) REFERENCES authors(author_id)
);
```

```sql
-- Customers table (no foreign keys)
CREATE TABLE customers (
    customer_id  INT            NOT NULL AUTO_INCREMENT,
    first_name   VARCHAR(100)  NOT NULL,
    last_name    VARCHAR(100)  NOT NULL,
    email        VARCHAR(150)  NOT NULL UNIQUE,
    joined_date  DATE,
    city         VARCHAR(80),
```

```
    PRIMARY KEY (customer_id)
);
```

```
-- Orders table (references both customers and books)
CREATE TABLE orders (
    order_id    INT            NOT NULL AUTO_INCREMENT,
    customer_id INT            NOT NULL,
    book_id     INT            NOT NULL,
    order_date  DATETIME       NOT NULL DEFAULT CURRENT_TIMESTAMP,
    quantity    INT            NOT NULL DEFAULT 1,
    total_price DECIMAL(8,2)   NOT NULL,
    PRIMARY KEY (order_id),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id),
    FOREIGN KEY (book_id)     REFERENCES books(book_id)
);
```

Create tables in the right order. Because `books` references `authors`, you must create `authors` first. Similarly, `orders` references both `customers` and `books`, so it must be created last. If you create a table before its parent exists, MySQL will return an error: *Cannot add foreign key constraint*.

Verify the structure

```
-- List all tables in the current database
SHOW TABLES;
+--------------------+
| Tables_in_bookshop |
+--------------------+
| authors            |
| books              |
| customers          |
| orders             |
+--------------------+

-- See the full column definition of a table
DESCRIBE books;
+--------------+-------------+------+-----+---------+----------------+
| Field        | Type        | Null | Key | Default | Extra          |
+--------------+-------------+------+-----+---------+----------------+
| book_id      | int         | NO   | PRI | NULL    | auto_increment |
| author_id    | int         | NO   | MUL | NULL    |                |
```

```
| title          | varchar(255) | NO   |     | NULL    |                |
| genre          | varchar(60)  | YES  |     | NULL    |                |
| published_year | year         | YES  |     | NULL    |                |
| price          | decimal(6,2) | NO   |     | NULL    |                |
| stock          | int          | YES  |     | 0       |                |
+----------------+--------------+------+-----+---------+----------------+
```

Concept	What it means
Table	Named collection of rows and columns describing one type of thing.
Row / Record	A single entry in a table — one book, one customer, one order.
Column / Field	One attribute of each row, with a fixed name and data type.
Primary Key	A column (or combination) that uniquely identifies each row. Never NULL, never duplicate, never changes.
Foreign Key	A column that references the primary key of another table, creating a relationship.
Referential Integrity	MySQL's guarantee that foreign key values always point to a real, existing row.
AUTO_INCREMENT	MySQL automatically assigns the next available integer when a row is inserted.
SQL sub-languages	DQL (SELECT), DML (INSERT/UPDATE/DELETE), DDL (CREATE/ALTER/DROP), TCL (BEGIN/COMMIT/ROLLBACK).
bookshop schema	authors → books → orders ← customers. Used throughout the entire course.

Next: Chapter 2 — SELECT Basics. The bookshop database is built. Now let's query it — choosing columns, filtering rows with WHERE, and combining conditions with AND, OR, and NOT.

The `SELECT` statement is the workhorse of SQL. Every time you want to read data from a database — whether it's one row or a million — you use `SELECT`. This chapter covers the anatomy of a query, choosing which columns to return, filtering rows with `WHERE`, and combining conditions with `AND`, `OR`, and `NOT`.

> **Sample data.** The examples in this chapter assume the bookshop database contains some rows. If you're following along, add a few authors and books with `INSERT` statements — Chapter 5 covers INSERT in full, but for now any handful of rows will work. The logic is the same regardless of which specific values are in your table.

Every SELECT query has the same basic skeleton. Here it is in annotated form:

```
SELECT title, price FROM books WHERE price < 10.00 ;
```

SELECT — the keyword that starts a query **title, price** — the columns you want returned **books** — the table to read from

price < 10.00 — only rows matching this condition

The `WHERE` clause is optional — without it, every row in the table is returned. `SELECT` and `FROM` are always required.

> **SQL is not case-sensitive for keywords.** `SELECT`, `select`, and `Select` all work. Convention is to write keywords in UPPERCASE and table/column names in lowercase — it makes queries easier to read at a glance. MySQL is also case-insensitive for table and column names on Windows and macOS by default (Linux is case-sensitive — something to watch if you move databases between servers).

SELECT * — all columns

The asterisk `*` is shorthand for "all columns". It's useful for quick exploration, but avoid it in production code — if someone adds a column to the table later, your query silently returns it too.

```
SELECT * FROM books;
```

book_id	author_id	title	genre	published_year	price	stock
1	1	The Midnight Library	Fiction	2020	8.99	42
2	2	Dune	Sci-Fi	1965	9.99	18
3	3	Sapiens	Non-fiction	2011	10.99	31
4	1	The Humans	Fiction	2013	7.99	25
5	4	Atomic Habits	Self-help	2018	11.99	60

All 7 columns, all 5 rows returned.

Named columns

Specify exactly the columns you need, separated by commas. The result set contains only those columns, in the order you listed them:

```
SELECT title, genre, price
FROM books;
```

title	genre	price
The Midnight Library	Fiction	8.99
Dune	Sci-Fi	9.99
Sapiens	Non-fiction	10.99
The Humans	Fiction	7.99
Atomic Habits	Self-help	11.99

Column aliases — AS

You can rename a column in the output using `AS`. The alias only affects the result label — it doesn't change anything in the database:

```sql
SELECT
    title          AS book_title,
    price          AS price_gbp,
    stock          AS units_available
FROM books;
```

book_title	price_gbp	units_available
The Midnight Library	8.99	42
Dune	9.99	18
Sapiens	10.99	31
The Humans	7.99	25
Atomic Habits	11.99	60

Aliases with spaces need backticks or quotes: `price AS `Price (GBP)``. Single-word aliases don't need them, though it's fine to include them for clarity.

Expressions in SELECT

You're not limited to raw column values — you can do calculations directly in the query. The result is computed row by row:

```sql
SELECT
    title,
    price,
    price * 1.20 AS price_with_vat,
    stock * price AS stock_value
FROM books;
```

title	price	price_with_vat	stock_value
The Midnight Library	8.99	10.788000	377.58
Dune	9.99	11.988000	179.82
Sapiens	10.99	13.188000	340.69
The Humans	7.99	9.588000	199.75
Atomic Habits	11.99	14.388000	719.40

Literal values and text

```sql
SELECT
    first_name,
    last_name,
    'bookshop.com' AS source
FROM authors;
```

first_name	last_name	source
Matt	Haig	bookshop.com
Frank	Herbert	bookshop.com
Yuval Noah	Harari	bookshop.com

The literal string 'bookshop.com' appears in every row — useful for tagging results when combining data from multiple sources.

DISTINCT — removing duplicates

If multiple rows have the same value in a column, `DISTINCT` collapses them into one:

```sql
-- Without DISTINCT: shows genre for every book (with repeats)
SELECT genre FROM books;
Fiction
Sci-Fi
Non-fiction
Fiction        ← duplicate
Self-help

-- With DISTINCT: each genre appears once
SELECT DISTINCT genre FROM books;
Fiction
Sci-Fi
Non-fiction
Self-help
```

The `WHERE` clause filters which rows are included in the result. MySQL evaluates the condition for every row in the table and only returns rows where the condition is **TRUE**.

```sql
-- Only books priced below £10
SELECT title, price
FROM books
WHERE price < 10.00;
```

title	price
The Midnight Library	8.99
Dune	9.99
The Humans	7.99

Comparison operators

Operator	Name	Example	Matches rows where…
`=`	Equal	`genre = 'Fiction'`	genre is exactly 'Fiction'
`!= or <>`	Not equal	`genre != 'Fiction'`	genre is anything except 'Fiction'
`<`	Less than	`price < 10.00`	price is below 10.00
`>`	Greater than	`price > 10.00`	price is above 10.00
`<=`	Less than or equal	`price <= 9.99`	price is 9.99 or below
`>=`	Greater than or equal	`stock >= 20`	stock is 20 or more
`BETWEEN … AND …`	Inclusive range	`price BETWEEN 8 AND 11`	price is 8, 11, or anything in between (both ends included)
`IN (…)`	Matches any value in a list	`genre IN ('Fiction', 'Sci-Fi')`	genre is either 'Fiction' or 'Sci-Fi'
`NOT IN (…)`	Matches none of the listed values	`genre NOT IN ('Self-help')`	genre is anything except 'Self-help'
`LIKE`	Pattern match	`title LIKE 'The%'`	title starts with 'The'. `%` = any characters, `_` = exactly one character
`IS NULL`	Value is absent	`nationality IS NULL`	the column has no value stored
`IS NOT NULL`	Value is present	`birth_year IS NOT NULL`	the column has a value

String comparisons

```sql
-- Exact match on a text column (single quotes around strings)
SELECT title, genre
FROM books
WHERE genre = 'Fiction';
```

title	genre
The Midnight Library	Fiction
The Humans	Fiction

Use single quotes for strings, not double quotes. In standard SQL and MySQL, string values are wrapped in single quotes: `'Fiction'`. Double quotes are reserved for identifiers (column/table names with spaces). Mixing them up causes confusing errors. If a string itself contains a single quote, escape it by doubling it: `'O''Brien'`.

LIKE — pattern matching

```sql
-- % matches any sequence of characters (including none)
SELECT title FROM books WHERE title LIKE 'The%';
The Midnight Library
The Humans

-- _ matches exactly one character
SELECT title FROM books WHERE title LIKE 'D___';
Dune

-- % at both ends: contains the word anywhere
SELECT title FROM books WHERE title LIKE '%Habit%';
Atomic Habits
```

LIKE with a leading % is slow on large tables. `LIKE '%something'` or `LIKE '%something%'` cannot use an index — MySQL must scan every row. `LIKE 'something%'` (trailing % only) *can* use an index and is much faster. For full-text search, MySQL has a dedicated `FULLTEXT` index type — but that's beyond the scope of this course.

BETWEEN

```sql
-- Both ends are inclusive — same as price >= 8.00 AND price <= 11.00
SELECT title, price
FROM books
WHERE price BETWEEN 8.00 AND 11.00;
```

title	price
The Midnight Library	8.99
Dune	9.99
Sapiens	10.99

IN — matching a list

```sql
-- Cleaner than writing: genre = 'Fiction' OR genre = 'Sci-Fi'
SELECT title, genre
FROM books
WHERE genre IN ('Fiction', 'Sci-Fi');
```

title	genre
The Midnight Library	Fiction
Dune	Sci-Fi
The Humans	Fiction

IS NULL / IS NOT NULL

```sql
-- NULL means "no value stored" — you CANNOT use = NULL
SELECT first_name, last_name
FROM authors
WHERE birth_year IS NULL;     -- correct

-- This will never return any rows, even if birth_year IS NULL:
WHERE birth_year = NULL           -- WRONG — always FALSE in SQL
```

NULL is not a value — it's the absence of a value. NULL does not equal anything, not even itself. `NULL = NULL` evaluates to NULL (not TRUE), which is why `WHERE birth_year = NULL` never matches any row. Always use `IS NULL` and `IS NOT NULL` to test for missing values.

A single `WHERE` clause can contain multiple conditions joined by logical operators. MySQL evaluates the full expression for each row.

AND — both conditions must be true

```sql
-- Fiction books that cost less than £10
SELECT title, genre, price
FROM books
WHERE genre = 'Fiction'
  AND price < 10.00;
```

title	genre	price
The Midnight Library	Fiction	8.99
The Humans	Fiction	7.99

OR — either condition is enough

```sql
-- Books that are Fiction OR are cheaper than £9
SELECT title, genre, price
FROM books
WHERE genre = 'Fiction'
   OR price < 9.00;
```

title	genre	price
The Midnight Library	Fiction	8.99
The Humans	Fiction	7.99

The Humans matches both conditions. The Midnight Library matches genre = 'Fiction'. No other book is below £9 and non-fiction.

NOT — invert a condition

```sql
-- Every book that is NOT Fiction
SELECT title, genre
FROM books
WHERE NOT genre = 'Fiction';
```

```sql
-- Equivalent, and more commonly written as:
WHERE genre != 'Fiction'
```

Operator precedence — AND binds before OR

This is one of the most common SQL mistakes. `AND` is evaluated before `OR`, just like multiplication before addition in arithmetic. Without parentheses, the logic may not be what you intended:

```sql
-- INTENDED: Fiction books, OR any book published after 2015

-- WRONG — AND binds first, so this reads:
-- "Fiction" OR "(price < 9 AND published_year > 2015)"
WHERE genre = 'Fiction'
   OR price < 9.00
  AND published_year > 2015;

-- RIGHT — parentheses make the grouping explicit
WHERE (genre = 'Fiction' OR price < 9.00)
  AND published_year > 2015;
```

> **When in doubt, use parentheses.** They cost nothing and make the logic unambiguous — both for MySQL and for whoever reads the query next. Any time you mix AND and OR in the same WHERE clause, wrap the OR groups in parentheses.

Logic truth tables

If you're ever unsure how AND, OR, and NOT combine, these tables show every combination. NULL is included because it silently affects results when columns can be null:

A AND B

A	B	Result
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE
TRUE	NULL	NULL

A OR B

A	B	Result
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE
TRUE	NULL	TRUE

NOT A

A	Result
TRUE	FALSE
FALSE	TRUE
NULL	NULL

```
FALSE     NULL     FALSE
```

```
FALSE     NULL     NULL
```

Putting it all together

```sql
-- Fiction or Sci-Fi, priced between £8 and £11, with stock available
SELECT title, genre, price, stock
FROM books
WHERE (genre IN ('Fiction', 'Sci-Fi'))
  AND price BETWEEN 8.00 AND 11.00
  AND stock > 0;
```

title	genre	price	stock
The Midnight Library	Fiction	8.99	42
Dune	Sci-Fi	9.99	18

Syntax	What it does
SELECT *	Return all columns. Avoid in production code.
SELECT col1, col2	Return only named columns, in the order listed.
col AS alias	Rename a column in the output. Doesn't change the database.
SELECT DISTINCT col	Remove duplicate values from the result.
WHERE col = 'value'	Filter rows — only rows where the condition is TRUE are returned.
= != < > <= >=	Comparison operators for numbers, strings, and dates.
BETWEEN a AND b	Inclusive range check. Both ends included.
IN ('a', 'b', 'c')	Match any value in a list. Cleaner than multiple OR conditions.
LIKE 'pattern%'	Pattern match. % = any characters, _ = one character.
IS NULL / IS NOT NULL	The only correct way to test for missing values. Never use = NULL.
AND / OR / NOT	Combine conditions. AND binds before OR — use parentheses to be explicit.

Next: Chapter 3 — Sorting and Limiting. Results currently come back in an unpredictable order. Next chapter covers ORDER BY to sort them, LIMIT to cap how many rows are returned, and OFFSET for pagination.

Without an `ORDER BY` clause, MySQL can return rows in any order it pleases — and that order may change between queries as the table grows or indexes are updated. This chapter covers how to sort results predictably, cap the number of rows returned, and page through large result sets efficiently.

Before diving in, here's how all the clauses we've covered so far — plus the new ones in this chapter — fit together. They must appear in this order:

SELECT	Which columns (and expressions) to return
FROM	Which table to read from
WHERE	Filter rows — only matching rows proceed (Chapter 2)
ORDER BY	Sort the result rows — new this chapter
LIMIT	Cap the number of rows returned — new this chapter
OFFSET	Skip the first N rows — new this chapter

Clause order is enforced by MySQL. You can omit optional clauses, but you cannot reorder them. Writing `LIMIT` before `WHERE`, for example, is a syntax error. Later chapters add `GROUP BY` and `HAVING` between `WHERE` and `ORDER BY`.

Basic ascending sort

`ORDER BY` followed by a column name sorts the results by that column. The default direction is **ascending** (A→Z for text, low→high for numbers, oldest→newest for dates):

```sql
SELECT title, price
FROM books
ORDER BY price;
```

title	price
The Humans	7.99
The Midnight Library	8.99
Dune	9.99
Sapiens	10.99
Atomic Habits	11.99

Cheapest first — ascending order is the default.

ASC and DESC

Add `ASC` or `DESC` after the column name to be explicit about direction. `ASC` is the default; `DESC` reverses it:

```sql
-- Most expensive first
SELECT title, price
FROM books
ORDER BY price DESC;
```

title	price
Atomic Habits	11.99
Sapiens	10.99
Dune	9.99
The Midnight Library	8.99
The Humans	7.99

```sql
-- Alphabetical by title (A → Z)
SELECT title, genre
FROM books
ORDER BY title ASC;
```

title	genre
Atomic Habits	Self-help

title	genre
Dune	Sci-Fi
Sapiens	Non-fiction
The Humans	Fiction
The Midnight Library	Fiction

Sorting by multiple columns

You can sort by more than one column — the second sort kicks in when two rows have the same value in the first column. Each column can have its own direction:

```sql
-- Sort by genre A→Z, then within each genre by price low→high
SELECT title, genre, price
FROM books
ORDER BY genre ASC, price ASC;
```

title	genre	price
The Humans	Fiction	7.99
The Midnight Library	Fiction	8.99
Sapiens	Non-fiction	10.99
Dune	Sci-Fi	9.99
Atomic Habits	Self-help	11.99

The two Fiction books are sorted by price within their genre group.

Sorting by an expression or alias

You can sort by a calculated expression, or reference a column alias defined in the `SELECT` list:

```sql
-- Sort by the calculated stock value, highest first
SELECT
    title,
    price,
    stock,
    price * stock AS stock_value
FROM books
ORDER BY stock_value DESC;
```

title	price	stock	stock_value
Atomic Habits	11.99	60	719.40
The Midnight Library	8.99	42	377.58
Sapiens	10.99	31	340.69
The Humans	7.99	25	199.75
Dune	9.99	18	179.82

ORDER BY with WHERE

`ORDER BY` always comes after `WHERE`. MySQL first filters the rows, then sorts only the rows that survived the filter:

```sql
-- Cheapest Fiction books first
SELECT title, price
FROM books
WHERE genre = 'Fiction'
ORDER BY price ASC;
```

title	price
The Humans	7.99
The Midnight Library	8.99

NULL values in ORDER BY

When a column contains NULL values, MySQL sorts them as if they are *lower than any other value* — NULLs appear first in `ASC` order and last in `DESC` order:

ORDER BY birth_year ASC

NULL	row 1
NULL	row 2
1920	row 3
1931	row 4
1976	row 5

ORDER BY birth_year DESC

1976	row 1
1931	row 2
1920	row 3
NULL	row 4
NULL	row 5

Push NULLs to the end in ASC order. If you want NULLs last regardless of sort direction, use `IS NULL` as a secondary sort: `ORDER BY birth_year IS NULL, birth_year ASC`. The `IS NULL` expression evaluates to 0 (false) for real values and 1 (true) for NULLs — so real values sort first.

`LIMIT N` tells MySQL to return at most N rows, even if more rows match the query. It's always applied after filtering and sorting.

```
-- The three cheapest books
SELECT title, price
FROM books
ORDER BY price ASC
LIMIT 3;
```

title	price
The Humans	7.99
The Midnight Library	8.99
Dune	9.99
Sapiens	10.99 ← not returned
Atomic Habits	11.99 ← not returned

Only the first 3 rows after sorting are returned.

LIMIT without ORDER BY is non-deterministic. If you write `LIMIT 3` without `ORDER BY`, MySQL returns whichever 3 rows it happens to find first — and that can change as the table is updated. Always pair LIMIT with ORDER BY so the result is predictable.

Common LIMIT patterns

```
-- The single most expensive book
SELECT title, price
FROM books
ORDER BY price DESC
LIMIT 1;
Atomic Habits  |  11.99
```

```sql
-- The most recently published book
SELECT title, published_year
FROM books
ORDER BY published_year DESC
LIMIT 1;
```

```
The Midnight Library  |  2020
```

```sql
-- Top 5 customers by number of orders (preview — uses COUNT, covered in Ch.8)
SELECT customer_id, COUNT(*) AS order_count
FROM orders
GROUP BY customer_id
ORDER BY order_count DESC
LIMIT 5;
```

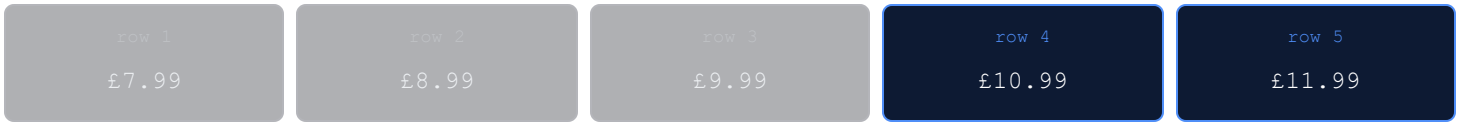
`OFFSET N` tells MySQL to skip the first N rows before starting to return results. Combined with `LIMIT`, this is how you page through a large result set — showing 10 rows per page, for example.

```sql
-- All books sorted by price, showing rows 4 and 5 (skip the first 3)
SELECT title, price
FROM books
ORDER BY price ASC
LIMIT 2 OFFSET 3;
```

title	price
Sapiens	10.99
Atomic Habits	11.99

Visualising LIMIT and OFFSET

Imagine all 5 books sorted by price. OFFSET skips rows from the left; LIMIT takes only the next N:



Grey = skipped by OFFSET 3 · Blue = returned by LIMIT 2

The pagination formula

To show page **P** of results, with **N** rows per page:

```
-- Page 1: LIMIT N OFFSET 0      (skip 0)
-- Page 2: LIMIT N OFFSET N      (skip 1 page)
-- Page 3: LIMIT N OFFSET 2*N    (skip 2 pages)
-- Page P: LIMIT N OFFSET (P-1)*N

-- Showing page 2 of books (5 per page, sorted by title)
SELECT title, price
FROM books
ORDER BY title ASC
LIMIT 5 OFFSET 5;     -- page 2: (2-1) * 5 = 5
```

Alternative syntax: LIMIT offset, count

MySQL also accepts a two-argument form of LIMIT where the first number is the offset and the second is the count. Both are equivalent:

```
-- These two queries are identical:
LIMIT 2 OFFSET 3
LIMIT 3, 2           -- offset first, then count (easy to mix up!)
```

The two-argument form is easy to mix up. `LIMIT 3, 2` means "skip 3, take 2" — not "take 3, skip 2". The explicit `LIMIT 2 OFFSET 3` form is harder to misread and is preferred for clarity, especially when reading someone else's code.

Counting total rows for pagination

When building paginated results in an application, you typically need to know the total row count so you can calculate the number of pages. Run a separate `COUNT(*)` query without `LIMIT`:

```
-- How many Fiction books are there in total?
SELECT COUNT(*) AS total
FROM books
WHERE genre = 'Fiction';
+-------+
```

```
| total |
+-------+
|     2 |
+-------+

-- Then fetch page 1 (5 per page)
SELECT title, price
FROM books
WHERE genre = 'Fiction'
ORDER BY title
LIMIT 5 OFFSET 0;
```

OFFSET pagination gets slow on large tables. `OFFSET 10000` forces MySQL to read and discard 10,000 rows before returning results — even though you only want 10. On tables with hundreds of thousands of rows this becomes a real performance problem. The solution (cursor-based pagination using `WHERE id > last_seen_id`) is an advanced topic, but worth knowing about if you build applications with large data sets.

Here's a realistic bookshop query that uses everything from the last three chapters — filtering, sorting, and limiting in one statement:

```
-- Show the top 3 best-value in-stock books (sorted by price ascending),
-- excluding the Self-help genre, published in 2010 or later
SELECT
    title,
    genre,
    published_year,
    price          AS price_gbp,
    stock          AS units_left
FROM books
WHERE genre != 'Self-help'
  AND published_year >= 2010
  AND stock > 0
ORDER BY price ASC
LIMIT 3;
```

title	genre	published_year	price_gbp	units_left
The Midnight Library	Fiction	2020	8.99	42
Sapiens	Non-fiction	2011	10.99	31

Only 2 rows returned — LIMIT 3 asked for up to 3, but only 2 rows matched all the WHERE conditions.

Syntax	What it does
`ORDER BY col`	Sort results by col ascending (A→Z, low→high). Default direction.
`ORDER BY col DESC`	Sort descending (Z→A, high→low).
`ORDER BY col1, col2`	Sort by col1 first; use col2 to break ties. Each column can have its own ASC/DESC.
`ORDER BY alias`	Sort by a column alias defined in the SELECT list.
`NULLs in ORDER BY`	ASC: NULLs first. DESC: NULLs last. Use `ORDER BY col IS NULL, col` to force NULLs last in ASC.
`LIMIT N`	Return at most N rows. Always pair with ORDER BY for predictable results.
`LIMIT N OFFSET M`	Skip M rows, then return the next N. Used for pagination.
`LIMIT M, N`	Alternative syntax for the same thing — offset first, count second. Prefer the OFFSET keyword form for readability.
`Pagination formula`	Page P with N rows/page: `LIMIT N OFFSET (P-1)*N`.

Next: Chapter 4 — Data Types and NULL. Columns are typed — INT, VARCHAR, DATE, DECIMAL, TEXT — and the choice matters for storage, performance, and correctness. We'll also go deeper on NULL: how it propagates through expressions, and how COALESCE and IFNULL help you handle missing values gracefully.

Every column in a MySQL table has a **data type** — and the type you choose matters more than it might seem. It determines how much storage a value uses, what operations you can perform on it, how it sorts, and what values are valid to insert. This chapter covers the most important types, how to choose between them, and how to handle `NULL` — the absence of a value — gracefully in queries.

TINYINT

−128 to 127 · 1 byte

Boolean flags, small counters, ratings out of 10. `TINYINT(1)` is MySQL's internal representation of a boolean (0/1).

SMALLINT

−32,768 to 32,767 · 2 bytes

Port numbers, year values (though YEAR type is better), small quantities. Rarely used — INT is usually fine.

INT

−2.1 billion to 2.1 billion · 4 bytes

The default choice for whole numbers. Primary keys, foreign keys, counts, stock levels, user IDs. `INT UNSIGNED` doubles the positive range to ~4.3 billion.

BIGINT

$\pm 9.2 \times 10^{18}$ · 8 bytes

When INT isn't big enough — social media post IDs, financial transaction counters, Unix timestamps in milliseconds. Uses twice the storage of INT.

DECIMAL(p, s)

Exact precision · variable

Always use for money. `DECIMAL(10, 2)` stores up to 10 digits total with exactly 2 after the decimal point. Never rounds silently.

FLOAT / DOUBLE

Approximate · 4 / 8 bytes

Scientific measurements, coordinates, ML model weights — where approximate is fine. **Never use for money** — floating-point arithmetic introduces rounding errors: `0.1 + 0.2 = 0.30000000000000004`.

Never store money in FLOAT or DOUBLE. Floating-point types cannot represent most decimal fractions exactly. A price of £8.99 stored as FLOAT may be retrieved as £8.98999977111816. Over thousands of transactions this adds up. Always use `DECIMAL(10, 2)` for currency.

```
-- DECIMAL in action: p=6 means 6 digits total, s=2 means 2 after the point
-- Maximum value: 9999.99
```

```
price DECIMAL(6, 2)    -- £9999.99 max, good for book prices
salary DECIMAL(10, 2)  -- £99,999,999.99 max, good for payroll
lat    DECIMAL(9, 6)   -- GPS latitude: e.g. 51.507351
```

CHAR(n)

Fixed length, 0–255 chars · always n bytes

Always stores exactly n characters, padding with spaces if shorter. Use only when every value is the same length — country codes (`CHAR(2)`), fixed codes, hashed passwords. Slightly faster to read than VARCHAR for fixed-width data.

VARCHAR(n)

Variable length, up to n chars · actual length + 1–2 bytes

The default choice for text with a known maximum — names, email addresses, titles, URLs. `VARCHAR(255)` is a safe default; choose a realistic maximum rather than always using 255.

TEXT

Up to 65,535 bytes

Long freeform text — blog post body, product descriptions, comments. Cannot have a DEFAULT value. Can't be fully indexed (only prefix indexes). Use when content might exceed VARCHAR's practical limit.

MEDIUMTEXT / LONGTEXT

Up to 16 MB / 4 GB

Very large documents — book contents, HTML pages, logs. Rarely needed; TEXT handles most use cases. Stored off-page, so accessing them is slower.

ENUM('a','b','c')

One of a fixed list · 1–2 bytes

Columns with a small, fixed set of valid values — status, gender, priority. MySQL enforces the list on insert. Downside: adding a new value requires an ALTER TABLE.

JSON

Variable · validated JSON

Semi-structured data with varying fields — settings, metadata, API responses. MySQL validates and stores JSON efficiently, and provides functions to query inside it. Avoid if the data could be normalised into columns.

VARCHAR(255) vs VARCHAR(1000) — does size matter? VARCHAR only uses as much storage as the actual content — declaring `VARCHAR(1000)` doesn't waste space if you only store 10-character values. The limit is enforced on insert, not storage. However, MySQL uses the declared length when creating temporary tables during query processing, so unnecessarily large VARCHARs can slow complex queries. Set a realistic maximum.

Type	Storage	Range	Use for	Example value
DATE	3 bytes	1000-01-01 to 9999-12-31	Birth dates, publication dates, order dates without time	'2024-06-15'
TIME	3 bytes	−838:59:59 to 838:59:59	Duration, opening hours. Rarely used for actual times of day.	'14:30:00'
DATETIME	5 bytes	1000-01-01 to 9999-12-31	Order timestamps, log entries. **Stores exactly what you put in — no timezone conversion.**	'2024-06-15 14:30:00'
TIMESTAMP	4 bytes	1970-01-01 to 2038-01-19	Record creation/update times. **Converts to UTC on store, back to local time on retrieve.** Auto-updates with `ON UPDATE CURRENT_TIMESTAMP`.	'2024-06-15 14:30:00'
YEAR	1 byte	1901 to 2155	Publication year, graduation year. Only stores the year — no month or day.	2024

```
-- How our bookshop tables use date types
published_year YEAR       -- just the year: 1965, 2020
joined_date    DATE       -- no time needed: '2023-03-15'
order_date     DATETIME   -- exact moment of purchase: '2024-06-15 09:42:17'

-- Useful date functions (preview — covered in Chapter 9)
SELECT NOW()             -- current date and time: 2024-06-15 14:30:00
SELECT CURDATE()         -- current date only: 2024-06-15
SELECT YEAR(order_date) -- extract the year part from a DATETIME column
```

TIMESTAMP has a 2038 problem. TIMESTAMP stores dates as a 32-bit integer counting seconds since 1970 — it overflows on 19 January 2038. For any date column that might hold values beyond 2038 (birth years, future bookings, expiry dates), use `DATETIME` instead. The 2038 deadline is closer than it sounds for long-running systems.

Type	Use for	Example
BOOLEAN	True/false flags. MySQL stores this as `TINYINT(1)` — 0 is false, 1 is true. `TRUE` and `FALSE` keywords work as aliases.	is_active BOOLEAN DEFAULT TRUE
BLOB / MEDIUMBLOB	Binary data — images, PDFs, files. Generally discouraged: store files on disk/object storage and keep only the path in the database.	thumbnail BLOB

Type	Use for	Example
`BINARY(n)` / `VARBINARY(n)`	Fixed or variable binary strings — hashed values, UUIDs stored as raw bytes. More efficient than storing a hex string.	`password_hash BINARY(32)`

Storing…	Use this type	Why
A whole number (ID, count, quantity)	INT	Standard choice. Use BIGINT only if you genuinely need >2.1 billion values.
Money / currency	DECIMAL(10, 2)	Exact arithmetic. Never FLOAT or DOUBLE.
A short text value with a cap	VARCHAR(n)	Email, name, title, URL. Set n to the realistic maximum.
A long freeform text (description, body)	TEXT	No length limit to worry about. Can't be defaulted or fully indexed.
A fixed-length code (country, status)	CHAR(2) or ENUM	CHAR is efficient when every value is the same width. ENUM enforces a valid value list.
A date only	DATE	Stores year/month/day with no time component.
A date and time (event, order, log)	DATETIME	No timezone surprises. Use TIMESTAMP only if you need auto-update on change.
A year only	YEAR	1 byte. Perfect for publication year, graduation year.
True / false	BOOLEAN (TINYINT(1))	0 or 1. Works with TRUE/FALSE keywords.

We introduced `NULL` in Chapter 2 as "the absence of a value". This section goes deeper — how NULL behaves in expressions and comparisons, and how to handle it cleanly in queries.

NULL propagates through expressions

Any arithmetic or string operation involving NULL produces NULL. Think of NULL as "unknown" — if one input is unknown, the result is unknown too:

```
NULL + 5  =  NULL

You can't add something unknown to 5 — the result is unknown
```

```
NULL * 0  = NULL

Even multiplying by zero — the result is still unknown

CONCAT('Hello, ', NULL)  = NULL

String concatenation with NULL returns NULL

NULL = NULL  = NULL

Not TRUE — two unknowns compared give an unknown result

NULL != NULL  = NULL

Also not TRUE — still unknown

NULL IS NULL  = TRUE

IS NULL is the correct way to test for NULL — always returns TRUE or FALSE
```

```sql
-- NULL in a calculation silently ruins results
SELECT
    title,
    price,
    stock,
    price * stock AS stock_value    -- NULL if stock is NULL
FROM books;
```

title	price	stock	stock_value
The Midnight Library	8.99	42	377.58
Dune	9.99	*NULL*	*NULL*
Sapiens	10.99	31	340.69

Dune has no stock value recorded — the calculation silently returns NULL instead of an error.

NULL in WHERE clauses

Because `NULL = NULL` is NULL (not TRUE), rows with NULL values are silently excluded from results that use `=` comparisons:

```sql
-- If nationality is NULL for some authors, they won't appear here
SELECT first_name, last_name, nationality
FROM authors
WHERE nationality = 'British';    -- NULL rows excluded silently

-- To find authors with NO nationality recorded
```

```sql
SELECT first_name, last_name
FROM authors
WHERE nationality IS NULL;

-- To find authors WITH a nationality recorded
SELECT first_name, last_name, nationality
FROM authors
WHERE nationality IS NOT NULL;
```

Rather than letting NULL propagate through your query, you can replace it with a sensible default using `IFNULL()` or `COALESCE()`.

`IFNULL(expr, fallback)`

Returns `expr` if it is not NULL; returns `fallback` if it is. Takes exactly two arguments. MySQL-specific (though widely supported).

`IFNULL(stock, 0)`
→ returns stock if not NULL, otherwise 0

`IFNULL(nationality, 'Unknown')`
→ returns nationality if not NULL, otherwise 'Unknown'

`COALESCE(a, b, c, …)`

Returns the first non-NULL value from a list of arguments. Standard SQL — works across all databases. More flexible than IFNULL when you have multiple fallback options.

`COALESCE(mobile, home_phone, 'No phone')`
→ tries mobile first, then home_phone, then the string

`COALESCE(stock, 0)`
→ same as IFNULL when only two arguments

IFNULL in practice

```sql
-- Replace NULL stock with 0 so the calculation works
SELECT
    title,
    price,
    IFNULL(stock, 0)                AS stock,
    price * IFNULL(stock, 0)        AS stock_value
FROM books;
```

title	price	stock	stock_value
The Midnight Library	8.99	42	377.58
Dune	9.99	0	0.00

title	price	stock	stock_value
Sapiens	10.99	31	340.69

NULL stock is treated as 0 — the calculation now produces a result for every row.

COALESCE with multiple fallbacks

```
-- Show the best available contact for each customer
-- Try email first, then phone, then a placeholder
SELECT
    first_name,
    last_name,
    COALESCE(email, phone, 'No contact info') AS contact
FROM customers;
```

NULLIF — the reverse

`NULLIF(a, b)` returns NULL if `a = b`, otherwise returns `a`. Useful for turning sentinel values (like 0 or an empty string) back into NULL so they don't interfere with calculations:

```
-- Avoid division by zero: if quantity is 0, treat it as NULL
SELECT
    order_id,
    total_price,
    quantity,
    total_price / NULLIF(quantity, 0) AS unit_price
FROM orders;
-- If quantity = 0, NULLIF returns NULL, making the division return NULL
-- rather than causing a division-by-zero error
```

The best way to deal with NULL is often to prevent it from being stored in the first place. `NOT NULL` makes a column mandatory — MySQL will reject any INSERT that doesn't provide a value. `DEFAULT` provides a fallback so the column is never empty even if the user doesn't supply a value:

```sql
CREATE TABLE books (
    book_id        INT            NOT NULL AUTO_INCREMENT,
    title          VARCHAR(255)   NOT NULL,          -- must be provided
    genre          VARCHAR(60),                     -- optional (nullable)
    price          DECIMAL(6,2)   NOT NULL,          -- must be provided
    stock          INT            NOT NULL DEFAULT 0, -- defaults to 0
    published_year YEAR,                            -- unknown year is valid
    PRIMARY KEY (book_id)
);
```

Use NOT NULL by default; allow NULL only when absence is meaningful. A NULL `nationality` on an author record means "we don't know" — a valid state. A NULL `price` on a book would be a data error — the column should be `NOT NULL`. When in doubt, ask: "is the absence of this value meaningful, or is it always a mistake?" If it's always a mistake, add `NOT NULL`.

Concept	Key points
`INT`	Whole numbers. Primary keys, counts, quantities. BIGINT if you need >2.1 billion.
`DECIMAL(p,s)`	Exact decimal numbers. Always use for money. Never FLOAT/DOUBLE for currency.
`VARCHAR(n)`	Variable-length text up to n characters. Default choice for names, emails, titles.
`TEXT`	Long freeform text. No DEFAULT value; only prefix-indexed.
`DATE / DATETIME`	DATE for dates only. DATETIME for dates with time. Avoid TIMESTAMP for dates past 2038.
`BOOLEAN`	Stored as TINYINT(1). Use TRUE/FALSE keywords; 0 = false, 1 = true.
`NULL`	Absence of a value. Propagates through arithmetic and string operations. Never use = NULL — always IS NULL / IS NOT NULL.
`IFNULL(x, y)`	Returns x if not NULL, otherwise y. Two arguments. MySQL-specific.
`COALESCE(a,b,c…)`	Returns the first non-NULL argument. Standard SQL. Prefer over IFNULL for portability and multiple fallbacks.
`NULLIF(a, b)`	Returns NULL if a = b, otherwise a. Useful to suppress sentinel values like 0 to prevent division by zero.
`NOT NULL / DEFAULT`	NOT NULL makes a column mandatory. DEFAULT provides a fallback. Prefer NOT NULL wherever absence of a value is an error.

Next: Chapter 5 — Modifying Data. The bookshop database has a schema but no data. Chapter 5 covers INSERT (adding rows), UPDATE (changing existing values), DELETE (removing rows), and the difference between TRUNCATE and DELETE.

The first four chapters focused on reading data with SELECT. Now it's time to write. This chapter covers the three DML statements that change the content of your tables: `INSERT` to add rows, `UPDATE` to change existing rows, and `DELETE` to remove them. We also look at `TRUNCATE` — a faster way to empty a table entirely — and when to use each.

DML recap. Data Manipulation Language statements change row contents without altering the table structure. They are transactional — you can wrap them in `BEGIN` / `COMMIT` / `ROLLBACK` to undo mistakes (covered in Chapter 10). For now we'll run them directly and see the effect immediately.

INSERT adds one or more new rows to a table. There are two forms: explicit column names (recommended) and positional (fragile — avoid it).

Single-row INSERT with column names

```
INSERT INTO authors (first_name, last_name, nationality) VALUES ('Matt', 'Haig',
'British');
```

```
-- Insert one author
INSERT INTO authors (first_name, last_name, nationality)
VALUES ('Matt', 'Haig', 'British');

Query OK, 1 row affected (0.01 sec)

-- MySQL auto-assigned author_id. Check it:
SELECT LAST_INSERT_ID();    -- returns the AUTO_INCREMENT value just generated
```

Always name your columns. Omitting the column list and relying on position (`INSERT INTO authors VALUES (1, 'Matt', 'Haig', 'British')`) breaks silently when someone adds, removes, or reorders a

column on the table. Named columns make the INSERT self-documenting and resilient to schema changes.

Omitting optional columns

You don't have to include every column. Columns with `DEFAULT` values or that allow NULL can be left out entirely:

```
-- nationality is nullable — we can omit it
INSERT INTO authors (first_name, last_name)
VALUES ('Ursula', 'Le Guin');
-- nationality will be NULL for this row

-- stock has DEFAULT 0, so we can omit it when inserting a book
INSERT INTO books (author_id, title, price, published_year)
VALUES (3, 'The Left Hand of Darkness', 8.49, 1969);
-- stock defaults to 0 automatically
```

Multi-row INSERT

You can insert many rows in a single statement by listing multiple value groups. This is much faster than running one INSERT per row — MySQL processes the batch in a single operation:

```
INSERT INTO authors (first_name, last_name, nationality)
VALUES
    ('Frank',   'Herbert',    'American'),
    ('Yuval',   'Harari',     'Israeli'),
    ('Agatha',  'Christie',   'British'),
    ('George',  'Orwell',     'British');

Query OK, 4 rows affected (0.01 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

INSERT ... SELECT

You can insert the results of a SELECT directly into a table. Useful for copying rows between tables, archiving data, or seeding test tables:

```sql
-- Copy all British authors into an audit table
INSERT INTO british_authors (first_name, last_name)
SELECT first_name, last_name
FROM   authors
WHERE  nationality = 'British';
```

Seeding the bookshop — inserting our full dataset

Let's populate all four tables with enough data to make the rest of the course meaningful. Run these in order — foreign key constraints require parent rows to exist first:

```sql
-- 1. Authors (no dependencies)
INSERT INTO authors (first_name, last_name, nationality)
VALUES
    ('Matt',    'Haig',     'British'),
    ('Frank',   'Herbert',  'American'),
    ('Yuval',   'Harari',   'Israeli'),
    ('Agatha',  'Christie', 'British'),
    ('Ursula',  'Le Guin',  'American');

-- 2. Books (depend on authors)
INSERT INTO books (author_id, title, genre, price, stock, published_year)
VALUES
    (1, 'The Midnight Library',        'Fiction',    8.99,  42, 2020),
    (1, 'Reasons to Stay Alive',       'Non-fiction',7.99,  18, 2015),
    (2, 'Dune',                        'Sci-fi',     9.99,  27, 1965),
    (3, 'Sapiens',                     'Non-fiction',10.99, 31, 2011),
    (4, 'And Then There Were None',    'Mystery',    6.99,  55, 1939),
    (5, 'The Left Hand of Darkness',   'Sci-fi',     8.49,   9, 1969);

-- 3. Customers (no dependencies)
INSERT INTO customers (first_name, last_name, email, joined_date)
VALUES
    ('Alice',  'Nguyen',   'alice@example.com',  '2022-03-10'),
    ('Ben',    'Okafor',   'ben@example.com',    '2023-07-22'),
    ('Chloe',  'Martinez', 'chloe@example.com',  '2021-11-05'),
    ('David',  'Singh',    'david@example.com',  '2024-01-18');

-- 4. Orders (depend on customers and books)
INSERT INTO orders (customer_id, book_id, quantity, total_price, order_date)
VALUES
```

```
    (1, 1, 2, 17.98, '2024-02-14 10:30:00'),
    (1, 4, 1, 10.99, '2024-02-14 10:30:00'),
    (2, 3, 1,  9.99, '2024-03-05 14:15:00'),
    (3, 5, 3, 20.97, '2024-03-20 09:00:00'),
    (4, 2, 1,  7.99, '2024-04-10 16:45:00');
```

With data in all four tables, every SELECT from here on will produce real results to verify against.

UPDATE changes the values of one or more columns in rows that match a condition. It does not add or remove rows — only modifies them.

```
UPDATE books SET price = 11.99, stock = 35 WHERE book_id = 4;
```

```
-- Correct a price error on Sapiens
UPDATE books
SET    price = 11.99
WHERE  book_id = 4;

Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

BEFORE

book_id	title	price
4	Sapiens	10.99

AFTER

book_id	title	price
4	Sapiens	11.99

Updating multiple columns at once

```
-- Update both price and stock in one statement
UPDATE books
SET    price = 9.49,
       stock = 50
WHERE  book_id = 3;    -- Dune
```

Updating with expressions

SET can use the column's current value in the expression — useful for increments, percentage changes, or clearing a field relative to what's there now:

```sql
-- Apply a 10% price increase to all Fiction books
UPDATE books
SET    price = ROUND(price * 1.10, 2)
WHERE  genre = 'Fiction';

-- Restock all books that have run low
UPDATE books
SET    stock = stock + 20
WHERE  stock < 10;

-- Clear a nullable field (set it to NULL)
UPDATE authors
SET    nationality = NULL
WHERE  author_id = 5;
```

UPDATE with a subquery

```sql
-- Give a 5% discount to books by British authors
UPDATE books
SET    price = ROUND(price * 0.95, 2)
WHERE  author_id IN (
    SELECT author_id
    FROM   authors
    WHERE  nationality = 'British'
);
```

UPDATE without WHERE updates every row in the table. `UPDATE books SET price = 0` will zero out all prices — there is no undo unless you are inside a transaction. Always write the WHERE clause *first*, run it as a SELECT to confirm it matches the right rows, then wrap it in UPDATE. The safe pattern is below.

The safe UPDATE pattern

Write SELECT with the same WHERE clause — confirm the right rows come back → Count matches — is it the number of rows you expected? → Replace SELECT … with UPDATE … SET … keeping the same WHERE →

STEP 4

Check "Rows matched" in the output matches your expected count

```
-- Step 1 & 2: verify which rows will be affected
SELECT book_id, title, price
FROM   books
WHERE  genre = 'Sci-fi';    -- shows 2 rows: Dune and The Left Hand of Darkness

-- Step 3: now run the update
UPDATE books
SET    price = ROUND(price * 0.90, 2)
WHERE  genre = 'Sci-fi';

-- Step 4: check the output
Rows matched: 2  Changed: 2  Warnings: 0   -- ✓ matches expected
```

DELETE removes rows from a table. Like UPDATE, it only affects rows that match the WHERE clause — and without a WHERE clause, it removes every row.

```
DELETE FROM books WHERE stock = 0 AND published_year < 1950;
```

```
-- Remove a single order by its primary key
DELETE FROM orders
WHERE       order_id = 5;

Query OK, 1 row affected (0.01 sec)

-- Remove all out-of-print books with zero stock
DELETE FROM books
```

```
WHERE       stock = 0
  AND       published_year < 2000;
```

DELETE and foreign keys

If a row is referenced by another table via a foreign key, MySQL will block the delete by default — this is referential integrity in action:

```
-- Attempt to delete an author who has books
DELETE FROM authors
WHERE       author_id = 1;    -- Matt Haig has 2 books

ERROR 1451 (23000): Cannot delete or update a parent row:
a foreign key constraint fails (`bookshop`.`books`,
CONSTRAINT `fk_books_author` FOREIGN KEY (`author_id`)
REFERENCES `authors` (`author_id`))

-- Solution: delete the child rows first, then the parent
DELETE FROM books   WHERE author_id = 1;     -- remove books first
DELETE FROM authors WHERE author_id = 1;     -- now the author can go
```

Apply the same safe pattern as UPDATE. Run the WHERE clause as a SELECT first to confirm which rows will be deleted. Delete is irreversible outside of a transaction — getting it wrong means restoring from a backup.

LIMIT on DELETE

You can add `LIMIT n` to a DELETE to cap how many rows are removed at once. Useful when cleaning up a large number of rows in batches to avoid locking the table for too long:

```
-- Delete old orders in batches of 1000 to avoid long locks
DELETE FROM orders
WHERE       order_date < '2020-01-01'
LIMIT       1000;
-- Run this repeatedly until 0 rows affected
```

Both `DELETE FROM table` (with no WHERE) and `TRUNCATE TABLE table` remove all rows. They behave very differently, though:

Feature	DELETE (no WHERE)	TRUNCATE TABLE
Speed	Slow on large tables — deletes row by row, logs each deletion	Very fast — drops and recreates the data pages
Transaction log	Fully logged — each row delete is recorded	Minimal logging — only the deallocation is logged
Can be rolled back?	Yes — if inside a transaction	No — TRUNCATE implicitly commits; can't be rolled back
WHERE clause	Supported — you can filter which rows to delete	Not supported — always removes all rows
AUTO_INCREMENT reset	Counter keeps its current value	Counter resets to 1
Foreign key checks	Enforced — will fail if rows are referenced elsewhere	Will fail if the table is a parent in an FK relationship
Triggers	Fires DELETE triggers	Does NOT fire DELETE triggers
Use when...	You need rollback safety, need to fire triggers, or are deleting a subset	You want to completely wipe a table fast — typically for dev/test resets

```sql
-- Wipe all rows — slow, logged, can be rolled back, keeps AUTO_INCREMENT
DELETE FROM orders;

-- Wipe all rows — fast, cannot be rolled back, resets AUTO_INCREMENT
TRUNCATE TABLE orders;
```

TRUNCATE resets AUTO_INCREMENT to 1. If any other table holds foreign keys pointing at the truncated table's primary key, and you then INSERT new rows that get IDs 1, 2, 3 again, those IDs will now clash with the references in the child table. In development this is usually fine — you truncate everything and reinsert. In production it is almost always the wrong choice. Use DELETE there.

Sometimes you want to insert a row if it doesn't exist, or update it if it does. This is called an *upsert*. MySQL offers two ways to do it.

INSERT … ON DUPLICATE KEY UPDATE

The most precise upsert: if the INSERT would violate a unique constraint, run the UPDATE clause instead. Existing rows that don't conflict are inserted normally:

```
-- Add stock if the book already exists, insert if it doesn't
INSERT INTO books (book_id, title, price, stock)
VALUES (3, 'Dune', 9.99, 10)
ON DUPLICATE KEY UPDATE
    stock = stock + VALUES(stock);    -- add the 10 to existing stock

-- VALUES(col) refers to the value that was in the attempted INSERT
```

REPLACE INTO

`REPLACE INTO` is simpler but more destructive: it DELETEs the conflicting row and INSERTs a new one. This resets AUTO_INCREMENT IDs and loses any columns you didn't include in the REPLACE. Use `INSERT … ON DUPLICATE KEY UPDATE` instead for precision:

```
-- REPLACE deletes the old row and inserts a brand new one
-- Any columns not listed get their DEFAULT values (or NULL)
REPLACE INTO books (book_id, title, price, stock)
VALUES (3, 'Dune', 9.99, 30);
-- The old Dune row is gone; a new one is inserted
-- genre and published_year would reset to NULL/DEFAULT!
```

Statement	What it does	Key rules
`INSERT INTO … VALUES`	Add one or more new rows	Always name the columns. Insert parents before children (foreign keys). Use multi-row VALUES for bulk inserts.
`INSERT … SELECT`	Insert the result of a SELECT	Column count and types in the SELECT must match the INSERT column list.

Statement	What it does	Key rules
`UPDATE … SET … WHERE`	Change values in matching rows	No WHERE = every row updated. Always SELECT-first to verify. SET can reference the current column value.
`DELETE FROM … WHERE`	Remove matching rows	No WHERE = every row deleted. Blocked by foreign key constraints. SELECT-first pattern applies.
`TRUNCATE TABLE`	Remove all rows instantly	Cannot be rolled back. Resets AUTO_INCREMENT. Does not fire triggers. Use only in dev/test resets.
`INSERT … ON DUPLICATE KEY UPDATE`	Insert if new, update if exists (upsert)	Preferred over REPLACE. Only updates the conflicting row; doesn't replace other columns.
`LAST_INSERT_ID()`	Return the AUTO_INCREMENT value from the last INSERT	Session-scoped — returns your own last insert, not someone else's concurrent insert.

Next: Chapter 6 — CREATE TABLE and Schema Design. Chapter 5 assumed the tables already existed. Chapter 6 goes back to the beginning: how to create tables from scratch, choose constraints, define primary and foreign keys, and alter or drop tables when requirements change.

Every table you've queried in this course was created with `CREATE TABLE`. This chapter explains how to write that statement from scratch — choosing the right columns, applying constraints to enforce data quality, defining relationships with foreign keys, and modifying or removing tables as requirements evolve.

DDL vs DML. CREATE TABLE is a Data Definition Language (DDL) statement — it changes the structure of the database, not its data. DDL statements in MySQL auto-commit immediately and cannot be rolled back, even inside a transaction. Double-check DDL before running it on a production database.

```
CREATE TABLE table_name (
    column_name  data_type  [constraints],
    column_name  data_type  [constraints],
    …
    [table-level constraints]
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4  COLLATE=utf8mb4_unicode_ci;
```

- **ENGINE=InnoDB** — the default and only choice for most tables. InnoDB supports transactions, foreign keys, and row-level locking. The older MyISAM engine does not.
- **DEFAULT CHARSET=utf8mb4** — stores any Unicode character, including emoji. Avoid the misleadingly-named `utf8` charset — MySQL's `utf8` only stores 3-byte characters and will silently corrupt 4-byte ones (most emoji).
- **COLLATE=utf8mb4_unicode_ci** — case-insensitive comparison. `ci` = case-insensitive, `cs` = case-sensitive. Most applications want `ci`.

IF NOT EXISTS. Add `IF NOT EXISTS` after `CREATE TABLE` to prevent an error if the table already exists — the statement becomes a no-op instead of crashing. Useful in setup scripts that may be run more than once:

```
CREATE TABLE IF NOT EXISTS authors ( … );
```

Constraints are rules MySQL enforces on every INSERT and UPDATE. They guarantee the data in a column is always valid — without them, you'd have to replicate the same validation logic in every application that talks to the database.

NOT NULL

column-level

The column must always have a value — MySQL rejects any INSERT or UPDATE that would leave it NULL. Use for columns where "unknown" is never a valid state: `title`, `price`, `email`.

DEFAULT value

column-level

Provides an automatic value when a column is omitted from an INSERT. Common defaults: `DEFAULT 0`, `DEFAULT ''`, `DEFAULT TRUE`, `DEFAULT CURRENT_TIMESTAMP`.

UNIQUE

column or table-level

No two rows may have the same value in this column (or combination of columns). Unlike PRIMARY KEY, a UNIQUE column can contain NULL — and multiple NULLs are allowed because NULL ≠ NULL.

PRIMARY KEY

column or table-level

Combines NOT NULL + UNIQUE. Every table should have one. Uniquely identifies each row. Usually an `INT AUTO_INCREMENT` surrogate key. Only one PRIMARY KEY per table.

FOREIGN KEY

table-level

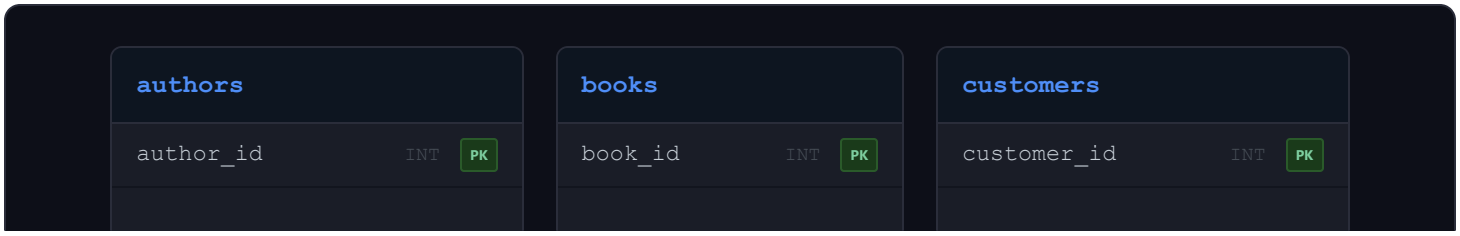
Enforces a relationship between tables — the value in this column must exist in the referenced parent table's column. Prevents orphaned child rows. Requires InnoDB.

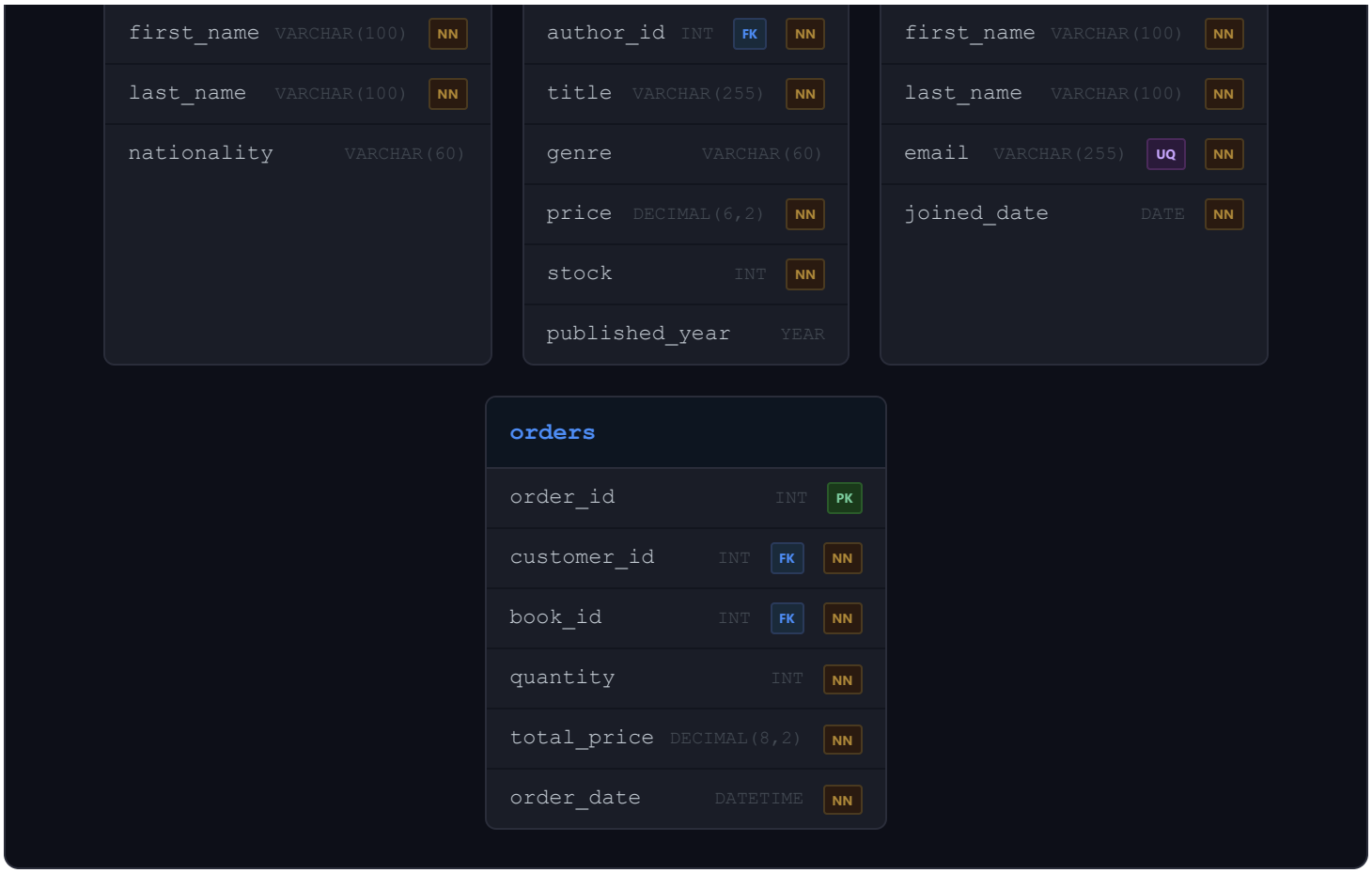
CHECK

column or table-level

A boolean expression that must be true for every row. Example: `CHECK (price > 0)`, `CHECK (stock >= 0)`. Enforced in MySQL 8.0.16+. Older versions parse but ignore CHECK constraints.

Here are all four bookshop tables written from scratch, with every constraint annotated. Notice the creation order — parent tables must exist before child tables that reference them via foreign keys:






```sql
-- Step 1: authors — no foreign keys, create first
CREATE TABLE IF NOT EXISTS authors (
    author_id   INT          NOT NULL AUTO_INCREMENT,
    first_name  VARCHAR(100) NOT NULL,
    last_name   VARCHAR(100) NOT NULL,
    nationality VARCHAR(60),                -- nullable: may be unknown
    PRIMARY KEY (author_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

-- Step 2: books — references authors
CREATE TABLE IF NOT EXISTS books (
    book_id        INT           NOT NULL AUTO_INCREMENT,
    author_id      INT           NOT NULL,
    title          VARCHAR(255)  NOT NULL,
    genre          VARCHAR(60),
    price          DECIMAL(6,2)  NOT NULL  CHECK (price > 0),
    stock          INT           NOT NULL  DEFAULT 0  CHECK (stock >= 0),
    published_year YEAR,
    PRIMARY KEY  (book_id),
    CONSTRAINT fk_books_author
        FOREIGN KEY (author_id) REFERENCES authors (author_id)
```

```
        ON DELETE RESTRICT
        ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

-- Step 3: customers — no foreign keys
CREATE TABLE IF NOT EXISTS customers (
    customer_id INT           NOT NULL AUTO_INCREMENT,
    first_name  VARCHAR(100)  NOT NULL,
    last_name   VARCHAR(100)  NOT NULL,
    email       VARCHAR(255)  NOT NULL  UNIQUE,
    joined_date DATE          NOT NULL  DEFAULT (CURDATE()),
    PRIMARY KEY (customer_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

-- Step 4: orders — references both customers and books
CREATE TABLE IF NOT EXISTS orders (
    order_id    INT           NOT NULL AUTO_INCREMENT,
    customer_id INT           NOT NULL,
    book_id     INT           NOT NULL,
    quantity    INT           NOT NULL  CHECK (quantity > 0),
    total_price DECIMAL(8,2)  NOT NULL,
    order_date  DATETIME      NOT NULL  DEFAULT NOW(),
    PRIMARY KEY (order_id),
    CONSTRAINT fk_orders_customer
        FOREIGN KEY (customer_id) REFERENCES customers (customer_id)
        ON DELETE RESTRICT
        ON UPDATE CASCADE,
    CONSTRAINT fk_orders_book
        FOREIGN KEY (book_id) REFERENCES books (book_id)
        ON DELETE RESTRICT
        ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

When a parent row is updated or deleted, MySQL needs to know what to do with the child rows that reference it. You control this with `ON DELETE` and `ON UPDATE` actions:

Action	Effect on child rows	Typical use
`RESTRICT`	Blocks the parent update/delete if child rows exist. The default — and the safest choice.	Default. Prevents accidental data loss. Forces you to deal with children explicitly.
`NO ACTION`	Same as RESTRICT in MySQL — the error is raised at statement end. Behaviourally identical.	Equivalent to RESTRICT. Mentioned in SQL standard; MySQL treats them the same.
`CASCADE`	Propagates the change: if parent is deleted, child rows are deleted too. If parent PK is updated, child FK is updated to match.	ON UPDATE CASCADE is very useful (PK changes ripple down). ON DELETE CASCADE is powerful but dangerous — one delete can silently wipe thousands of rows.
`SET NULL`	Sets the FK column in child rows to NULL when the parent is deleted or updated. The FK column must be nullable.	Soft orphaning — e.g. when an author is deleted, keep their books but clear author_id. Only valid if the FK column allows NULL.
`SET DEFAULT`	Sets the FK column to its DEFAULT value. Rarely used — InnoDB doesn't support it in most configurations.	Avoid. Poorly supported and almost never the right semantic.

```
-- Common pattern: restrict deletes, cascade updates
CONSTRAINT fk_books_author
    FOREIGN KEY (author_id) REFERENCES authors (author_id)
    ON DELETE RESTRICT    -- can't delete author if they have books
    ON UPDATE CASCADE     -- if author_id changes, books.author_id follows
```

Name your constraints. Always give foreign keys an explicit name with `CONSTRAINT fk_table_column`. MySQL generates one if you don't, but the auto-generated name is unreadable in error messages (`fk_books_ibfk_1`). A named constraint gives you a clear error: *"Cannot delete parent row: constraint fk_books_author"* — you know exactly which relationship was violated.

```
-- List all tables in the current database
SHOW TABLES;

-- Describe a table's columns, types, and constraints
DESCRIBE books;
-- or the shorter alias:
DESC books;
```

Field	Type	Null	Key	Default	Extra
book_id	int	NO	PRI	NULL	auto_increment
author_id	int	NO	MUL	NULL	
title	varchar(255)	NO		NULL	
genre	varchar(60)	YES		NULL	
price	decimal(6,2)	NO		NULL	
stock	int	NO		0	
published_year	year	YES		NULL	

Key column: PRI = primary key, MUL = non-unique index (here: a foreign key). Default NULL means no default was set — the column must be supplied on INSERT if it's NOT NULL.

```
-- Show the full CREATE TABLE statement MySQL is using
-- Useful for seeing indexes and foreign keys that DESC doesn't show
SHOW CREATE TABLE books\G
```

Requirements change — columns get added, renamed, resized, or removed. ALTER TABLE modifies an existing table's structure without losing data (mostly — dropping a column deletes its data permanently).

Operation	What it does	Notes
ADD COLUMN	Add a new column to the table	Existing rows get NULL or the DEFAULT value. Adding NOT NULL without a DEFAULT fails if there are existing rows.
DROP COLUMN	Remove a column and all its data permanently	Irreversible. Cannot drop if the column is part of an index or FK.
MODIFY COLUMN	Change a column's type, constraints, or default — keeping the same name	Narrowing a type (VARCHAR 255 → 50) may truncate existing data. MySQL warns but proceeds.
RENAME COLUMN	Rename a column without changing its type	MySQL 8.0+. Updates any indexes on the column automatically.
ADD CONSTRAINT	Add a UNIQUE, CHECK, or FOREIGN KEY constraint after creation	Adding FK or UNIQUE validates all existing rows — fails if violations exist.

Operation	What it does	Notes
`DROP CONSTRAINT / DROP FOREIGN KEY`	Remove a named constraint	Must use the constraint name. Use SHOW CREATE TABLE to find it.
`RENAME TABLE`	Rename the table itself	Also works as a standalone `RENAME TABLE old TO new` statement.

```sql
-- Add a biography column to authors
ALTER TABLE authors
    ADD COLUMN bio TEXT;    -- nullable, existing rows get NULL

-- Add a non-nullable column safely: provide a DEFAULT so existing rows aren't rejected
ALTER TABLE books
    ADD COLUMN is_active BOOLEAN NOT NULL DEFAULT TRUE;

-- Widen a column that's proving too short
ALTER TABLE authors
    MODIFY COLUMN nationality VARCHAR(100);   -- was VARCHAR(60)

-- Rename a column (MySQL 8.0+)
ALTER TABLE customers
    RENAME COLUMN joined_date TO registered_date;

-- Drop a column (data is gone permanently)
ALTER TABLE authors
    DROP COLUMN bio;

-- Add a UNIQUE constraint after the fact
ALTER TABLE authors
    ADD CONSTRAINT uq_author_name UNIQUE (first_name, last_name);

-- Add a foreign key after creation
ALTER TABLE books
    ADD CONSTRAINT fk_books_author
        FOREIGN KEY (author_id) REFERENCES authors (author_id)
        ON DELETE RESTRICT ON UPDATE CASCADE;

-- Drop a foreign key (use the constraint name)
ALTER TABLE books
    DROP FOREIGN KEY fk_books_author;

-- Rename the table
RENAME TABLE orders TO purchases;
```

ALTER TABLE locks the table on older MySQL versions. On MySQL 5.x, ALTER TABLE rewrites the entire table, which can lock it for minutes on large datasets. MySQL 8 performs most ALTERs online (no lock), but adding FULLTEXT indexes or changing primary keys still require a table rebuild. On production systems, use a tool like *pt-online-schema-change* or *gh-ost* for zero-downtime schema changes.

`DROP TABLE` deletes the table entirely — structure and all data. It cannot be rolled back. If you want to keep the structure but remove the data, use TRUNCATE instead.

```sql
-- Drop a table (fails if it doesn't exist)
DROP TABLE orders;

-- Safe version — no error if the table doesn't exist
DROP TABLE IF EXISTS orders;

-- Drop multiple tables in one statement
DROP TABLE IF EXISTS orders, books, customers, authors;
-- Drop child tables before parents when FK constraints are present

-- Or temporarily disable FK checks to drop in any order (dev/test only)
SET FOREIGN_KEY_CHECKS = 0;
DROP TABLE IF EXISTS authors, books, customers, orders;
SET FOREIGN_KEY_CHECKS = 1;    -- always re-enable!
```

DROP TABLE is irreversible. There is no recycle bin. The data is gone the moment the statement commits. In production, always take a backup before dropping a table — even if you think it's empty. If you're unsure, rename the table first and leave it for a week before dropping.

Statement / concept	Key points
`CREATE TABLE`	Define columns, types, and constraints. Use IF NOT EXISTS for idempotent scripts. Always specify ENGINE=InnoDB and CHARSET=utf8mb4.

Statement / concept	Key points
`NOT NULL`	Makes a column mandatory. Use by default; allow NULL only when absence is meaningful.
`DEFAULT`	Auto-fills a column when omitted from INSERT. Allows adding NOT NULL columns to existing tables safely.
`UNIQUE`	No duplicate values. NULLs are exempt — multiple NULLs are allowed.
`PRIMARY KEY`	NOT NULL + UNIQUE. One per table. Usually INT AUTO_INCREMENT.
`FOREIGN KEY`	Enforces parent-child relationships. Name constraints explicitly. Create parents before children; drop children before parents.
`ON DELETE / ON UPDATE`	RESTRICT (safe default), CASCADE (propagates change), SET NULL (orphans child row). Avoid SET DEFAULT.
`CHECK`	Boolean expression enforced on every write. MySQL 8.0.16+. Good for price > 0, stock >= 0.
`ALTER TABLE`	ADD / DROP / MODIFY / RENAME COLUMN; ADD / DROP CONSTRAINT. DDL auto-commits — no rollback.
`DROP TABLE`	Deletes table and all data permanently. Use IF NOT EXISTS. Drop children before parents.

Next: Chapter 7 — Aggregate Functions and GROUP BY. How to summarise data across rows: COUNT, SUM, AVG, MIN, MAX, the GROUP BY clause for grouping, and HAVING for filtering groups — plus the difference between WHERE and HAVING.

Every query so far has returned one row per row in the table. Aggregate functions change that — they collapse many rows into a single summary value. Combined with `GROUP BY`, they let you produce per-category summaries: total sales per genre, average price per author, number of orders per customer. This chapter covers the five core aggregate functions, how GROUP BY works, and how HAVING lets you filter those summaries.

`COUNT()`

`COUNT(*)` or `COUNT(col)`

Count rows. `COUNT(*)` counts all rows including NULLs. `COUNT(col)` counts only non-NULL values in that column.

`SUM()`

`SUM(col)`

Total of all non-NULL values. Returns NULL if all values are NULL (use COALESCE to default to 0).

`AVG()`

`AVG(col)`

Mean of all non-NULL values. NULLs are excluded from both numerator and denominator — they don't count as zero.

`MIN()`

`MIN(col)`

Smallest value. Works on numbers, strings (alphabetical), and dates. Ignores NULLs.

`MAX()`

`MAX(col)`

Largest value. Works on numbers, strings, and dates. Ignores NULLs.

`COUNT(DISTINCT)`

`COUNT(DISTINCT col)`

Count unique non-NULL values. Useful for "how many different genres do we stock?" rather than "how many books?"

Aggregates across the whole table

Without GROUP BY, an aggregate function collapses the entire table into one row:

```sql
SELECT
    COUNT(*)                AS total_books,
    COUNT(genre)            AS books_with_genre,   -- excludes NULLs
    COUNT(DISTINCT genre)   AS unique_genres,
    SUM(stock)              AS total_stock,
    AVG(price)              AS avg_price,
    MIN(price)              AS cheapest,
    MAX(price)              AS most_expensive,
    MIN(published_year)     AS oldest_year,
    MAX(published_year)     AS newest_year
FROM books;
```

total_books	books_with_genre	unique_genres	total_stock	avg_price	cheapest	most_expensive	oldest_year	newest_year
6	6	4	182	9.07	6.99	10.99	1939	2020

All six books collapsed into one summary row. avg_price is rounded here for display — MySQL returns the full decimal.

All aggregate functions silently ignore NULL values — except `COUNT(*)` which counts rows regardless. This asymmetry causes a common mistake with AVG:

```
-- Suppose 2 of our 6 books have no stock value (NULL)
-- These two queries give DIFFERENT answers:

SELECT AVG(stock)                    AS avg_stock        FROM books;
-- MySQL: ignores the 2 NULLs, averages the 4 known values
-- Result: 30.25  (121 / 4)

SELECT SUM(stock) / COUNT(*)         AS avg_stock        FROM books;
-- Divides by 6 (all rows), treating NULLs as if stock were 0
-- Result: 20.17  (121 / 6)

-- If you mean "NULL stock counts as zero in the average", be explicit:
SELECT AVG(IFNULL(stock, 0))         AS avg_stock        FROM books;
-- Result: 20.17 — matches SUM/COUNT(*) calculation
```

AVG skips NULLs — it doesn't treat them as zero. If stock is NULL for some books and you want those to contribute 0 to the average, wrap the column in `IFNULL(stock, 0)`. If you genuinely want to average only books where stock is known, `AVG(stock)` is correct as-is.

GROUP BY splits the rows into groups — one group per distinct value of the grouping column(s) — and runs the aggregate function separately on each group. The result has one row per group.

```
-- How many books and what is the average price per genre?
SELECT
    genre,
    COUNT(*)       AS book_count,
    AVG(price)     AS avg_price,
    SUM(stock)     AS total_stock
FROM   books
GROUP BY genre
ORDER BY book_count DESC;
```

genre	book_count	avg_price	total_stock
Non-fiction	2	9.49	49
Sci-fi	2	9.24	36
Fiction	1	8.99	42
Mystery	1	6.99	55

Four genres, four result rows. MySQL grouped all Sci-fi books together, calculated COUNT/AVG/SUM for that group, then repeated for each genre.

Grouping by multiple columns

You can group by two or more columns to get a cross-tabulation. Each unique combination of the grouped columns becomes its own row:

```
-- Orders per customer per book
SELECT
    customer_id,
    book_id,
    COUNT(*)          AS times_ordered,
    SUM(quantity)     AS total_qty
FROM   orders
GROUP BY customer_id, book_id
ORDER BY customer_id;
```

The golden rule of GROUP BY

> **Every column in SELECT must either be in GROUP BY, or wrapped in an aggregate function.** If you put a non-grouped, non-aggregated column in SELECT, MySQL (in strict mode) will error. In non-strict mode it picks an arbitrary value from the group — which is almost certainly wrong.

```
-- ✗ WRONG: title is not in GROUP BY and not aggregated
SELECT genre, title, COUNT(*) FROM books GROUP BY genre;
-- ERROR 1055: 'title' is not in GROUP BY and is not aggregated

-- ✓ CORRECT option 1: add title to GROUP BY (group by both)
SELECT genre, title, COUNT(*) FROM books GROUP BY genre, title;

-- ✓ CORRECT option 2: aggregate title instead
SELECT genre, MIN(title) AS sample_title, COUNT(*) FROM books GROUP BY genre;
```

WHERE filters individual rows before grouping. HAVING filters groups after the aggregate has been calculated. It's the only place you can reference aggregate functions in a filter condition.

```sql
-- Only show genres with more than one book and average price above £8
SELECT
    genre,
    COUNT(*)      AS book_count,
    AVG(price)  AS avg_price
FROM   books
GROUP BY genre
HAVING   COUNT(*) > 1
  AND    AVG(price) > 8.00
ORDER BY avg_price DESC;
```

genre	book_count	avg_price
Non-fiction	2	9.49
Sci-fi	2	9.24

Fiction (1 book) and Mystery (1 book) were excluded by HAVING COUNT(*) > 1.

WHERE vs HAVING — the key distinction

WHERE — filters rows before grouping

Runs **before** GROUP BY. Each row is evaluated individually. Cannot reference aggregate functions — the aggregates don't exist yet.

- Use for column values: `WHERE price > 5`
- Use for date ranges: `WHERE order_date > '2024-01-01'`
- Use for row-level filters: `WHERE genre IS NOT NULL`
- Faster — reduces rows before the costly grouping step

HAVING — filters groups after aggregating

Runs **after** GROUP BY. Each group is evaluated. Can (and usually does) reference aggregate functions.

- Use for aggregate results: `HAVING COUNT(*) > 1`
- Use for group totals: `HAVING SUM(quantity) >= 10`
- Use for group averages: `HAVING AVG(price) < 8`
- Can also filter by grouped columns, but WHERE is faster for that

```sql
-- Combined WHERE + GROUP BY + HAVING + ORDER BY
-- "For fiction and sci-fi books only, which genres have average stock above 20?"
SELECT
    genre,
    COUNT(*)       AS book_count,
    AVG(stock)     AS avg_stock,
    SUM(stock)     AS total_stock
FROM     books
WHERE    genre IN ('Fiction', 'Sci-fi')    -- row filter: only these genres enter grouping
GROUP BY genre
HAVING   AVG(stock) > 20                   -- group filter: only groups with high avg stock
ORDER BY avg_stock DESC;
```

MySQL evaluates clauses in a fixed logical order that is different from the order you write them. Understanding this explains every "column not found" and "can't use aggregate in WHERE" error:

1	`FROM`	Identify the source table(s) and load the rows
2	`WHERE`	Filter individual rows — aggregates don't exist yet
3	`GROUP BY`	Partition surviving rows into groups
4	`HAVING`	Filter groups — aggregates are now calculated and available
5	`SELECT`	Choose columns, compute expressions, assign aliases
6	`ORDER BY`	Sort the final result — aliases from SELECT are now available
7	`LIMIT / OFFSET`	Trim the result to the requested page

Why can't I use a SELECT alias in WHERE or HAVING? Because SELECT runs after both WHERE and HAVING. The alias doesn't exist at the time WHERE and HAVING are evaluated. You can use aliases in ORDER BY because that runs after SELECT. To reuse a calculation in HAVING without repeating it, some databases support column aliases in HAVING — MySQL does allow this as an extension, but it's not standard SQL and can behave unexpectedly. Repeating the expression is safer.

Revenue per customer

```
SELECT
    customer_id,
    COUNT(*)           AS order_count,
    SUM(total_price)   AS total_spent,
    AVG(total_price)   AS avg_order_value
FROM     orders
GROUP BY customer_id
ORDER BY total_spent DESC;
```

customer_id	order_count	total_spent	avg_order_value
1	2	28.97	14.49
3	1	20.97	20.97
2	1	9.99	9.99

customer_id	order_count	total_spent	avg_order_value
4	1	7.99	7.99

Customer 1 (Alice) has placed two orders totalling £28.97 — the highest spender.

Finding customers who spent more than £15 total

```
SELECT
    customer_id,
    SUM(total_price) AS total_spent
FROM     orders
GROUP BY customer_id
HAVING   SUM(total_price) > 15.00
ORDER BY total_spent DESC;
```

customer_id	total_spent
1	28.97
3	20.97

Books per author — minimum one book

```
SELECT
    author_id,
    COUNT(*)            AS books_published,
    MIN(published_year) AS first_year,
    MAX(published_year) AS latest_year,
    SUM(stock)          AS total_copies_in_stock
FROM     books
GROUP BY author_id
ORDER BY books_published DESC;
```

author_id	books_published	first_year	latest_year	total_copies_in_stock
1	2	2015	2020	60
2	1	1965	1965	27
3	1	2011	2011	31
4	1	1939	1939	55
5	1	1969	1969	9

Author 1 (Matt Haig) has two books in the catalogue. IDs rather than names appear here — Chapter 8 (JOINs) will combine both tables to show the name alongside.

Counting NULL vs non-NULL — COUNT(*) vs COUNT(col)

```sql
-- How many authors have a nationality recorded vs unknown?
SELECT
    COUNT(*)                AS total_authors,
    COUNT(nationality)      AS with_nationality,
    COUNT(*) - COUNT(nationality) AS unknown_nationality
FROM authors;
```

total_authors	with_nationality	unknown_nationality
5	4	1

One author has no nationality recorded (Ursula Le Guin's nationality was set to NULL in Chapter 5). COUNT(*) caught it; COUNT(nationality) skipped it.

WITH ROLLUP — adding a grand total row

`WITH ROLLUP` is a GROUP BY modifier that appends an extra summary row for each grouping level, with the final row representing the grand total across all groups:

```sql
SELECT
    IFNULL(genre, 'TOTAL') AS genre,
    COUNT(*)               AS book_count,
    SUM(stock)             AS total_stock
FROM   books
GROUP BY genre WITH ROLLUP;
-- ROLLUP sets the grouping column to NULL in the summary rows
-- IFNULL converts NULL → 'TOTAL' for the label
```

genre	book_count	total_stock
Fiction	1	42
Mystery	1	55
Non-fiction	2	49
Sci-fi	2	36
TOTAL	*6*	*182*

The final row is the grand total across all genres — added automatically by WITH ROLLUP.

Concept	Key points
`COUNT(*)`	Counts all rows, including those with NULL values in any column.
`COUNT(col)`	Counts only non-NULL values in that column. Use COUNT(*) − COUNT(col) to count NULLs.
`SUM / AVG / MIN / MAX`	All ignore NULL values. AVG denominates by non-NULL count — not total row count.
`COUNT(DISTINCT col)`	Counts unique non-NULL values — useful for "how many different X" questions.

Concept	Key points
`GROUP BY`	Splits rows into groups; aggregates run per group. Every non-aggregated SELECT column must appear in GROUP BY.
`HAVING`	Filters groups after aggregation. The only place you can use aggregate functions in a condition. WHERE filters rows; HAVING filters groups.
`Execution order`	FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT. Aliases from SELECT are only available in ORDER BY.
`WITH ROLLUP`	Appends subtotal and grand total rows to a GROUP BY result. ROLLUP sets grouping columns to NULL in summary rows — use IFNULL to label them.

Next: Chapter 8 — JOINs. The queries in this chapter returned author IDs rather than author names — because the name lives in a different table. JOINs combine rows from two or more tables based on a shared key. Chapter 8 covers INNER JOIN, LEFT JOIN, RIGHT JOIN, self-joins, and the difference between them.

Every chapter so far has queried a single table. Real databases split data across multiple related tables — orders live in `orders`, the customer name lives in `customers`, the book title lives in `books`. A JOIN combines rows from two or more tables based on a matching column, letting you query that connected data as if it were one result set.

> **Why split data across tables?** If we stored the customer name inside every order row, changing a customer's name would require updating every one of their orders — and any that were missed would have inconsistent data. By keeping names in one place (`customers`) and referencing them by ID, a single UPDATE fixes everything. JOINs are the mechanism that puts it back together at query time.

JOIN queries reference two or more tables. Qualifying every column name with the full table name (`orders.customer_id`, `customers.customer_id`) gets verbose fast. A table alias — a short nickname assigned in the FROM clause — keeps things concise:

```sql
-- Without aliases — verbose
SELECT orders.order_id, customers.first_name, customers.last_name
FROM   orders
JOIN   customers ON orders.customer_id = customers.customer_id;

-- With aliases — the standard way
SELECT o.order_id, c.first_name, c.last_name
FROM   orders    o
JOIN   customers c  ON o.customer_id = c.customer_id;
```

The alias follows the table name in the FROM or JOIN clause, with or without the optional keyword `AS`. Both forms are identical — `orders o` and `orders AS o` mean the same thing.

`a → authors` `b → books` `c → customers` `o → orders`

We'll use these single-letter aliases consistently throughout this chapter.

An INNER JOIN returns rows where the ON condition matches in *both* tables. Rows that have no match on the other side are silently excluded. `JOIN` and `INNER JOIN` are identical — `INNER` is optional.

```sql
-- Every order with the customer name and book title
SELECT
    o.order_id,
    c.first_name,
    c.last_name,
    b.title,
    o.quantity,
    o.total_price
FROM        orders    o
INNER JOIN  customers c  ON o.customer_id = c.customer_id
INNER JOIN  books     b  ON o.book_id     = b.book_id
ORDER BY    o.order_id;
```

order_id	first_name	last_name	title	quantity	total_price
1	Alice	Nguyen	The Midnight Library	2	17.98
2	Alice	Nguyen	Sapiens	1	10.99
3	Ben	Okafor	Dune	1	9.99
4	Chloe	Martinez	And Then There Were None	3	20.97
5	David	Singh	Reasons to Stay Alive	1	7.99

Three tables joined in one query. Customer IDs and book IDs are replaced with the real names and titles.

Joining all four tables — a full order receipt

```sql
-- Full receipt: order → customer → book → author
SELECT
    o.order_id,
    o.order_date,
    CONCAT(c.first_name, ' ', c.last_name)  AS customer,
    b.title,
    CONCAT(a.first_name, ' ', a.last_name)  AS author,
    o.quantity,
    o.total_price
```

```
FROM        orders    o
JOIN        customers c  ON o.customer_id = c.customer_id
JOIN        books     b  ON o.book_id     = b.book_id
JOIN        authors   a  ON b.author_id   = a.author_id
ORDER BY    o.order_date;
```

order_id	order_date	customer	title	author	qty	total
1	2024-02-14	Alice Nguyen	The Midnight Library	Matt Haig	2	17.98
2	2024-02-14	Alice Nguyen	Sapiens	Yuval Harari	1	10.99
3	2024-03-05	Ben Okafor	Dune	Frank Herbert	1	9.99
4	2024-03-20	Chloe Martinez	And Then There Were None	Agatha Christie	3	20.97
5	2024-04-10	David Singh	Reasons to Stay Alive	Matt Haig	1	7.99

All four tables joined in one query. Notice David (customer 4) was added in Chapter 5 but never ordered — he won't appear here because INNER JOIN only keeps matched rows.

INNER JOIN

Only rows that match in *both* tables. No match = excluded.

LEFT JOIN

All rows from the left table. Right side NULLs where no match.

RIGHT JOIN

All rows from the right table. Left side NULLs where no match.

LEFT ANTI-JOIN

Left rows with *no* match on the right. Find orphans / unmatched records.

JOIN type	Keeps from left	Keeps from right	Best for
`INNER JOIN`	Matched only	Matched only	The common case — you only care about rows that have data on both sides
`LEFT JOIN`	All rows	Matched rows + NULLs for non-matches	Keep all left rows even without a right-side match — "show all customers, even those with no orders"
`RIGHT JOIN`	Matched rows + NULLs for non-matches	All rows	Rarely needed — rewrite as LEFT JOIN by swapping table order instead
`LEFT JOIN + WHERE right.col IS NULL`	Unmatched only	Excluded	Find rows with no match — "customers who have never ordered", "books with no orders"
`CROSS JOIN`	All rows	All rows	Cartesian product — every left row paired with every right row. Rarely useful; almost always accidental.

LEFT JOIN returns all rows from the left table, whether or not they have a matching row in the right table. When there is no match, every column from the right table is NULL in that result row.

```sql
-- All customers, with their orders if they have any
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    o.order_id,
    o.total_price
FROM       customers c
LEFT JOIN  orders    o  ON c.customer_id = o.customer_id
ORDER BY   c.customer_id;
```

customer_id	first_name	last_name	order_id	total_price
1	Alice	Nguyen	1	17.98

customer_id	first_name	last_name	order_id	total_price
1	Alice	Nguyen	2	10.99
2	Ben	Okafor	3	9.99
3	Chloe	Martinez	4	20.97
4	David	Singh	*NULL*	*NULL*
5	David	Singh (if added)	*NULL*	*NULL*

David (customer 4) has never ordered — LEFT JOIN still includes him, with NULLs for the orders columns.

LEFT JOIN + aggregate — counting orders per customer including zeros

```sql
-- Count orders per customer — include customers with zero orders
SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer,
    COUNT(o.order_id)                      AS order_count,
    COALESCE(SUM(o.total_price), 0)         AS total_spent
FROM      customers c
LEFT JOIN orders    o  ON c.customer_id = o.customer_id
GROUP BY  c.customer_id, c.first_name, c.last_name
ORDER BY  total_spent DESC;
```

customer_id	customer	order_count	total_spent
1	Alice Nguyen	2	28.97
3	Chloe Martinez	1	20.97
2	Ben Okafor	1	9.99
4	David Singh	0	0.00

David appears with 0 orders and £0.00 spent. COUNT(o.order_id) returns 0 (not COUNT(*)) because o.order_id is NULL for David — COUNT(col) skips NULLs. COALESCE converts NULL SUM to 0.00.

COUNT(*) vs COUNT(right_table_col) in a LEFT JOIN. After a LEFT JOIN, unmatched rows have NULL in every right-table column. `COUNT(*)` would count those NULL rows as 1. Use `COUNT(o.order_id)` — it skips NULLs and correctly returns 0 for customers with no orders.

Combine LEFT JOIN with `WHERE right_table.col IS NULL` to find rows that have no corresponding entry in the joined table. This is one of the most practical JOIN patterns in real applications:

```sql
-- Customers who have never placed an order
SELECT
    c.customer_id,
    c.first_name,
    c.last_name
FROM      customers c
LEFT JOIN orders    o  ON c.customer_id = o.customer_id
WHERE     o.customer_id IS NULL;    -- only keep rows where no order matched
```

customer_id	first_name	last_name
4	David	Singh

```sql
-- Books that have never been ordered
SELECT
    b.book_id,
    b.title
FROM      books  b
LEFT JOIN orders o  ON b.book_id = o.book_id
WHERE     o.book_id IS NULL;
```

book_id	title
6	The Left Hand of Darkness

The Left Hand of Darkness has never been ordered — a good candidate for a promotion or a stock review.

RIGHT JOIN keeps all rows from the right table and NULLs for the left where there is no match — the mirror image of LEFT JOIN. In practice, RIGHT JOIN is rarely used because any RIGHT JOIN can be rewritten as a LEFT JOIN by swapping the table order, which most developers find easier to read:

```sql
-- These two queries return identical results:

-- Using RIGHT JOIN
SELECT b.title, o.order_id
```

```sql
FROM        orders o
RIGHT JOIN  books  b  ON o.book_id = b.book_id;

-- Equivalent LEFT JOIN (just swap the table order)
SELECT b.title, o.order_id
FROM       books  b
LEFT JOIN  orders o  ON b.book_id = o.book_id;
```

> **Prefer LEFT JOIN over RIGHT JOIN.** Both do the same job but most developers read SQL top-to-bottom and find it more natural to start with the "primary" table on the left. Mixing LEFT and RIGHT JOINs in the same query quickly becomes confusing. Stick to LEFT JOIN and swap table order when needed.

A self-join joins a table to itself using two different aliases. It's used when rows in the same table have a relationship to each other — the classic example is an `employees` table where each employee has a `manager_id` that points to another row in the same table:

```sql
-- Suppose our authors table had a "mentored_by" column
-- pointing to another author_id in the same table
-- Self-join: show each author alongside their mentor
SELECT
    a.first_name  AS author,
    m.first_name  AS mentor
FROM       authors a
LEFT JOIN  authors m  ON a.mentored_by = m.author_id;
```

The bookshop schema doesn't have a recursive relationship, but self-joins come up often in organisational charts (employees + managers), category hierarchies (parent category → child category), and graph structures stored in relational tables.

JOINs compose cleanly with every clause from previous chapters. The execution order stays the same — FROM + JOIN happens first, then WHERE, GROUP BY, HAVING, SELECT, ORDER BY, LIMIT:

Sales report by genre

```sql
-- Total revenue and units sold per genre, top genres first
SELECT
    b.genre,
    COUNT(DISTINCT o.order_id)  AS order_lines,
    SUM(o.quantity)              AS units_sold,
    SUM(o.total_price)           AS revenue
FROM     orders o
JOIN     books  b  ON o.book_id = b.book_id
GROUP BY b.genre
ORDER BY revenue DESC;
```

genre	order_lines	units_sold	revenue
Mystery	1	3	20.97
Fiction	1	2	17.98
Non-fiction	2	2	18.98
Sci-fi	1	1	9.99

Best-selling authors

```sql
-- Revenue per author, only those who have sold something
SELECT
    CONCAT(a.first_name, ' ', a.last_name) AS author,
    SUM(o.quantity)                         AS units_sold,
    SUM(o.total_price)                      AS revenue
FROM     authors a
JOIN     books   b  ON a.author_id  = b.author_id
JOIN     orders  o  ON b.book_id    = o.book_id
GROUP BY a.author_id, a.first_name, a.last_name
ORDER BY revenue DESC;
```

author	units_sold	revenue
Agatha Christie	3	20.97
Matt Haig	3	25.97
Yuval Harari	1	10.99
Frank Herbert	1	9.99

Ursula Le Guin doesn't appear — her book has no orders (INNER JOIN excludes her). Use LEFT JOIN chains if you need to include authors with zero sales.

ON vs WHERE — filtering in the right place

```sql
-- Row filter (same for INNER and LEFT JOIN — fine in either place)
SELECT c.first_name, o.order_id, o.total_price
FROM      customers c
LEFT JOIN orders    o  ON c.customer_id = o.customer_id
WHERE     o.total_price > 10.00;   -- filters AFTER join; loses unmatched customers

-- To filter on the joined table WITHOUT losing unmatched rows from the left,
-- move the condition into the ON clause:
SELECT c.first_name, o.order_id, o.total_price
FROM      customers c
LEFT JOIN orders    o  ON  c.customer_id = o.customer_id
                       AND o.total_price  > 10.00;
-- Customers with no qualifying orders still appear; their order columns are NULL
```

In a LEFT JOIN, filtering on a right-table column in WHERE converts it to an INNER JOIN. Once WHERE filters out rows where the right-table column is NULL, the "keep all left rows" guarantee is broken. Move right-table filters into the ON clause if you still need unmatched left rows in the result.

Concept	Key points
Table alias	Short nickname (a, b, c, o) assigned in FROM/JOIN. Required when the same column name exists in multiple tables. Use alias.column to qualify.
INNER JOIN	Returns only rows that match in both tables. Unmatched rows on either side are excluded. Most common join type.
LEFT JOIN	Returns all rows from the left table. Right-table columns are NULL for rows with no match. Use to include records regardless of whether a related record exists.
Anti-join pattern	LEFT JOIN + WHERE right.col IS NULL — finds left-table rows with no match on the right. "Customers who never ordered", "books never sold".
RIGHT JOIN	Mirror of LEFT JOIN. Rarely used — rewrite as LEFT JOIN by swapping table order.
Self-join	Join a table to itself using two aliases. Used for hierarchical/recursive data: manager → employee, parent category → child category.
COUNT in LEFT JOIN	Use COUNT(right_table.pk) not COUNT(*) — COUNT(*) counts NULL rows as 1; COUNT(col) skips them, giving the correct 0 for unmatched rows.
ON vs WHERE in LEFT JOIN	Filtering on a right-table column in WHERE silently converts LEFT JOIN to INNER JOIN. Move right-table filters into the ON clause to preserve unmatched left rows.

Next: Chapter 9 — String, Date, and Numeric Functions. MySQL has dozens of built-in functions for transforming data: CONCAT, SUBSTRING, REPLACE, UPPER/LOWER for strings; DATE_FORMAT, DATEDIFF, DATE_ADD for dates; ROUND, FLOOR, CEIL, MOD for numbers. Chapter 9 covers the ones you'll reach for every week.

MySQL has hundreds of built-in functions. This chapter covers the ones you'll use every week — functions for cleaning and reshaping strings, working with dates and times, and performing precise numeric calculations. Each section follows the same pattern: a quick-reference card of the core functions, then practical bookshop queries showing how they compose.

> **Functions go in the SELECT list, WHERE clause, ORDER BY, and ON conditions.** They can be nested, combined with operators, and aliased with AS. Anywhere you can write a column name, you can write a function call that produces a value.

CORE STRING FUNCTIONS

`CONCAT(a, b, …)`	Join two or more strings. Returns NULL if any argument is NULL — use CONCAT_WS to avoid this.	`CONCAT('Matt', ' ', 'Haig') → 'Matt Haig'`
`CONCAT_WS(sep, a, b, …)`	Concat With Separator. Skips NULL arguments (unlike CONCAT). The separator is the first argument.	`CONCAT_WS(' ', 'Matt', NULL, 'Haig') → 'Matt`
`LENGTH(str)`	Number of bytes in the string. For multi-byte characters (emoji, accents), use	`LENGTH('Dune') → 4`

	CHAR_LENGTH instead.	
`CHAR_LENGTH(str)`	Number of characters (not bytes). Safe for Unicode — 'café' is 4 chars, 5 bytes in utf8mb4.	`CHAR_LENGTH('café') → 4`
`UPPER(str) / LOWER(str)`	Convert to upper or lower case. Respects the column's collation.	`UPPER('dune') → 'DUNE'`
`TRIM(str)`	Remove leading and trailing spaces. LTRIM removes only left spaces, RTRIM only right.	`TRIM(' Dune ') → 'Dune'`
`SUBSTRING(str, pos, len)`	Extract a portion of a string. Position is 1-based. Omit len to go to end of string. Also: SUBSTR, MID.	`SUBSTRING('Dune', 1, 2) → 'Du'`
`LEFT(str, n) / RIGHT(str, n)`	First or last n characters. Cleaner than SUBSTRING when you just want the start or end.	`LEFT('Sapiens', 3) → 'Sap'`
`REPLACE(str, from, to)`	Replace all occurrences of a substring. Case-sensitive. Returns the modified string.	`REPLACE('Sci-fi', '-', '/') → 'Sci/fi'`
`LOCATE(needle, haystack)`	Position of the first occurrence of	`LOCATE('mid', 'Midnight') → 1`

	needle in haystack. Returns 0 if not found. Case-insensitive by default.	
`LPAD(str, len, pad)` / `RPAD`	Left or right pad a string to a given total length with the pad character. Useful for formatting IDs.	`LPAD('5', 4, '0')` → `'0005'`
`REPEAT(str, n)`	Repeat a string n times.	`REPEAT('★', 3)` → `'★★★'`
`REVERSE(str)`	Reverse a string character by character.	`REVERSE('Dune')` → `'enuD'`

Practical string queries

```sql
-- Full name as a single string, title-truncated to 20 chars
SELECT
    CONCAT(a.first_name, ' ', a.last_name)   AS author_name,
    LEFT(b.title, 20)                         AS short_title,
    CHAR_LENGTH(b.title)                      AS title_length
FROM   books   b
JOIN   authors a  ON b.author_id = a.author_id
ORDER BY title_length DESC;
```

author_name	short_title	title_length
Ursula Le Guin	The Left Hand of Dark	25
Agatha Christie	And Then There Were	24
Matt Haig	The Midnight Library	20
Matt Haig	Reasons to Stay Aliv	20
Yuval Harari	Sapiens	7
Frank Herbert	Dune	4

```sql
-- Search for titles containing a word, case-insensitively
SELECT title
FROM   books
WHERE  LOCATE('the', LOWER(title)) > 0;

-- Pad order IDs to 6 digits for a reference number
SELECT
    CONCAT('ORD-', LPAD(order_id, 6, '0')) AS reference,
    total_price
FROM   orders;
```

reference	total_price
ORD-000001	17.98
ORD-000002	10.99
ORD-000003	9.99
ORD-000004	20.97
ORD-000005	7.99

```sql
-- Clean up messy email addresses from a data import
UPDATE customers
SET    email = LOWER(TRIM(email));

-- Replace a genre label that was entered inconsistently
UPDATE books
SET    genre = REPLACE(genre, 'Sci-fi', 'Science Fiction')
WHERE  genre = 'Sci-fi';
```

CONCAT_WS for nullable columns. If any argument to plain `CONCAT()` is NULL, the entire result is NULL. Use `CONCAT_WS(' ', first_name, middle_name, last_name)` to safely build a full name even when middle_name is NULL — CONCAT_WS skips NULL arguments but keeps the separator between the rest.

CORE DATE AND TIME FUNCTIONS

`NOW()`	Current date and time as DATETIME. Stays fixed for the duration of the statement.	`2024-06-15 14:30:00`
`CURDATE()` / `CURRENT_DATE`	Current date only (no time).	`2024-06-15`
`CURTIME()`	Current time only (no date).	`14:30:00`
`DATE(datetime)`	Extract just the date portion from a DATETIME value.	`DATE('2024-06-15 14:30:00') → '2024-0`
`YEAR(date)` / `MONTH(date)` / `DAY(date)`	Extract an individual component from a date. Also: HOUR, MINUTE, SECOND.	`YEAR('2024-06-15') → 2024`
`DAYOFWEEK(date)`	Day number 1=Sunday … 7=Saturday. DAYNAME returns the name ('Monday' etc).	`DAYOFWEEK('2024-06-15') → 7 (Saturday`
`DATE_FORMAT(date, fmt)`	Format a date/datetime as a string using format specifiers (see table below).	`DATE_FORMAT(NOW(), '%d %M %Y') → '15`
`DATEDIFF(end, start)`	Number of days between two dates (end minus start). Ignores the time component.	`DATEDIFF('2024-06-15', '2024-01-01')`
`TIMESTAMPDIFF(unit, start, end)`	Difference in a given unit: SECOND, MINUTE, HOUR,	`TIMESTAMPDIFF(YEAR, '1990-03-01', CUR`

	DAY, MONTH, YEAR. Accounts for time.	
`DATE_ADD(date, INTERVAL n unit)`	Add an interval to a date. Units: DAY, WEEK, MONTH, YEAR, HOUR, MINUTE, SECOND.	`DATE_ADD('2024-01-01', INTERVAL 3 MON`
`DATE_SUB(date, INTERVAL n unit)`	Subtract an interval from a date. Equivalent to DATE_ADD with a negative interval.	`DATE_SUB(CURDATE(), INTERVAL 30 DAY)`
`STR_TO_DATE(str, fmt)`	Parse a string into a DATE/DATETIME using a format pattern. The inverse of DATE_FORMAT.	`STR_TO_DATE('15/06/2024', '%d/%m/%Y')`
`LAST_DAY(date)`	The last day of the month for the given date. Useful for month-end reports.	`LAST_DAY('2024-02-01') → '2024-02-29'`

DATE_FORMAT specifiers

Specifier	Meaning	Example output
`%Y`	4-digit year	`2024`
`%y`	2-digit year	`24`
`%m`	Month number, zero-padded (01–12)	`06`
`%M`	Month name	`June`
`%b`	Abbreviated month name	`Jun`
`%d`	Day of month, zero-padded (01–31)	`15`
`%e`	Day of month, no padding (1–31)	`15`
`%W`	Weekday name	`Saturday`
`%a`	Abbreviated weekday name	`Sat`
`%H`	Hour in 24-hour format (00–23)	`14`

Specifier	Meaning	Example output
%h	Hour in 12-hour format (01–12)	02
%i	Minutes (00–59)	30
%s	Seconds (00–59)	00
%p	AM or PM	PM
%T	Time as HH:MM:SS (24-hour shorthand)	14:30:00

Practical date queries

```sql
-- Format order dates for a customer-facing receipt
SELECT
    CONCAT('ORD-', LPAD(o.order_id, 6, '0'))      AS reference,
    DATE_FORMAT(o.order_date, '%d %M %Y')         AS order_date,
    DATE_FORMAT(o.order_date, '%H:%i')            AS time_placed,
    CONCAT(c.first_name, ' ', c.last_name)       AS customer
FROM   orders    o
JOIN   customers c  ON o.customer_id = c.customer_id
ORDER BY o.order_date;
```

reference	order_date	time_placed	customer
ORD-000001	14 February 2024	10:30	Alice Nguyen
ORD-000002	14 February 2024	10:30	Alice Nguyen
ORD-000003	05 March 2024	14:15	Ben Okafor
ORD-000004	20 March 2024	09:00	Chloe Martinez
ORD-000005	10 April 2024	16:45	David Singh

```sql
-- Orders placed in the last 90 days
SELECT order_id, order_date, total_price
FROM   orders
WHERE  order_date >= DATE_SUB(CURDATE(), INTERVAL 90 DAY);

-- How many days has each customer been registered?
SELECT
    CONCAT(first_name, ' ', last_name)   AS customer,
    joined_date,
    DATEDIFF(CURDATE(), joined_date)   AS days_registered
FROM     customers
ORDER BY days_registered DESC;
```

customer	joined_date	days_registered
Chloe Martinez	2021-11-05	956
Alice Nguyen	2022-03-10	827
Ben Okafor	2023-07-22	328
David Singh	2024-01-18	149

```sql
-- Monthly revenue breakdown — group by year and month
SELECT
    DATE_FORMAT(order_date, '%Y-%m')   AS month,
    COUNT(*)                           AS orders,
    SUM(total_price)                   AS revenue
FROM     orders
GROUP BY DATE_FORMAT(order_date, '%Y-%m')
ORDER BY month;
```

month	orders	revenue
2024-02	2	28.97
2024-03	2	30.96
2024-04	1	7.99

Grouping by DATE_FORMAT produces one row per calendar month — a classic pattern for revenue dashboards and trend reports.

```sql
-- Find customers who joined more than 1 year ago
-- and flag their loyalty tier
SELECT
    CONCAT(first_name, ' ', last_name)          AS customer,
    TIMESTAMPDIFF(YEAR, joined_date, CURDATE()) AS years_member,
    CASE
        WHEN TIMESTAMPDIFF(YEAR, joined_date, CURDATE()) >= 3 THEN 'Gold'
        WHEN TIMESTAMPDIFF(YEAR, joined_date, CURDATE()) >= 1 THEN 'Silver'
        ELSE 'Bronze'
    END                                         AS tier
FROM     customers
ORDER BY years_member DESC;
```

customer	years_member	tier
Chloe Martinez	2	Silver
Alice Nguyen	2	Silver
Ben Okafor	0	Bronze

customer	years_member	tier
David Singh	0	Bronze

CASE expressions (covered in Chapter 10) combine beautifully with date functions for business logic like loyalty tiers.

CORE NUMERIC FUNCTIONS

`ROUND(n, d)`	Round n to d decimal places. d defaults to 0 (round to integer). Standard rounding: 0.5 rounds up.	`ROUND(8.567, 2) → 8.57`
`FLOOR(n)`	Round down to the nearest integer, regardless of the decimal part. Negative: FLOOR(-2.3) = -3.	`FLOOR(8.9) → 8`
`CEIL(n) / CEILING(n)`	Round up to the nearest integer. Negative: CEIL(-2.3) = -2.	`CEIL(8.1) → 9`
`TRUNCATE(n, d)`	Remove digits after d decimal places without rounding. Different from ROUND — always truncates towards zero.	`TRUNCATE(8.999, 2) → 8.99`
`ABS(n)`	Absolute value — removes the sign. ABS(-5) = ABS(5) = 5.	`ABS(-42.50) → 42.50`

`MOD(n, d)`	Remainder after dividing n by d. Also written as n % d. Useful for odd/even checks and cycle detection.	`MOD(10, 3) → 1`
`POWER(n, exp)`	n raised to the power of exp. Also: POW(n, exp).	`POWER(2, 10) → 1024`
`SQRT(n)`	Square root of n. Returns NULL for negative values.	`SQRT(25) → 5`
`GREATEST(a, b, …) / LEAST(a, b, …)`	Maximum or minimum of a list of values. Returns NULL if any argument is NULL.	`GREATEST(3, 9, 2) → 9`
`FORMAT(n, d)`	Format a number with comma thousands separator and d decimal places. Returns a string — don't use for further calculations.	`FORMAT(1234567.8, 2) → '1,234,567.80'`
`RAND()`	Random float between 0 and 1. RAND(seed) for repeatable results. ORDER BY RAND() for random row order.	`FLOOR(RAND() * 100) → 0–99`

Practical numeric queries

```sql
-- Apply a 15% VAT to prices, rounded to 2dp
SELECT
    title,
    price                              AS price_ex_vat,
```

```
    ROUND(price * 1.15, 2)          AS price_inc_vat,
    ROUND(price * 0.15, 2)          AS vat_amount
FROM   books
ORDER BY price DESC;
```

title	price_ex_vat	price_inc_vat	vat_amount
Sapiens	10.99	12.64	1.65
Dune	9.99	11.49	1.50
The Midnight Library	8.99	10.34	1.35
The Left Hand of Darkness	8.49	9.76	1.27
Reasons to Stay Alive	7.99	9.19	1.20
And Then There Were None	6.99	8.04	1.05

```
-- FLOOR vs ROUND vs TRUNCATE — see the difference clearly
SELECT
    8.567                   AS original,
    ROUND(8.567, 2)         AS rounded,      -- 8.57
    TRUNCATE(8.567, 2)      AS truncated,    -- 8.56 (no rounding)
    FLOOR(8.567)            AS floored,      -- 8
    CEIL(8.567)             AS ceiled;       -- 9

-- MOD use case: find books with an odd book_id
SELECT book_id, title
FROM   books
WHERE  MOD(book_id, 2) = 1;    -- remainder 1 = odd

-- Pick 3 random books for a "You might also like" feature
SELECT title, price
FROM   books
ORDER BY RAND()
LIMIT  3;
```

```
-- FORMAT for display — human-readable thousands separators
SELECT
    CONCAT('£', FORMAT(SUM(total_price), 2)) AS total_revenue,
    CONCAT('£', FORMAT(AVG(total_price), 2)) AS avg_order
FROM   orders;
```

total_revenue	avg_order
£67.92	£13.58

FORMAT returns a string, not a number. Don't use FORMAT in a calculation or WHERE clause — you'll get unexpected results or implicit type conversion. Use FORMAT only in the SELECT list for final display output. For rounding in calculations, use ROUND.

Function	Key points
CONCAT / CONCAT_WS	Join strings. Use CONCAT_WS to safely handle NULL arguments — it skips them; plain CONCAT returns NULL if any argument is NULL.
LENGTH / CHAR_LENGTH	LENGTH counts bytes; CHAR_LENGTH counts characters. Use CHAR_LENGTH for Unicode text.
SUBSTRING / LEFT / RIGHT	Extract parts of a string. Position is 1-based. Omit length to read to the end.
REPLACE / TRIM / UPPER / LOWER	Clean and normalise text. Useful in UPDATE statements for fixing data quality issues.
NOW / CURDATE / CURTIME	Current datetime / date / time. NOW() stays fixed for the duration of one statement.
DATE_FORMAT	Format a date as a string using % specifiers. Essential for display and for grouping by month/year.
DATEDIFF / TIMESTAMPDIFF	DATEDIFF returns days; TIMESTAMPDIFF takes a unit (YEAR, MONTH, DAY, HOUR…) for flexible intervals.
DATE_ADD / DATE_SUB	Arithmetic on dates. Use INTERVAL syntax: DATE_ADD(col, INTERVAL 7 DAY).
ROUND / FLOOR / CEIL / TRUNCATE	ROUND rounds normally. FLOOR always rounds down, CEIL always rounds up. TRUNCATE drops digits without rounding.
MOD	Remainder after division. Useful for odd/even detection and cyclic patterns.
FORMAT	Human-readable number with thousands separators. Returns a STRING — for display only, not further calculations.
RAND()	Random float 0–1. Use ORDER BY RAND() LIMIT n for random sampling (slow on large tables — use with care).

Next: Chapter 10 — Transactions and the CASE Expression. Chapter 10 covers how to wrap multiple statements in a transaction so they either all succeed or all roll back together (BEGIN / COMMIT / ROLLBACK / SAVEPOINT), and the CASE expression for conditional logic inside queries — including how it powers loyalty tiers, status labels, and conditional aggregation.

This final chapter brings together two topics that sit at the heart of production SQL. Transactions let you group multiple statements into a single all-or-nothing unit — essential any time one logical operation requires several writes. The CASE expression brings conditional logic directly into your queries, powering everything from status labels to conditional aggregation.

Why transactions exist

Imagine a customer buys a book. You need to do two things: insert a row into `orders` and decrement `stock` in `books`. If the server crashes or an error occurs between the two statements, you end up with an order but no stock reduction — or vice versa. The database is now inconsistent.

A transaction makes those two writes atomic: either both happen, or neither does. The database moves from one consistent state to another without any in-between half-state being visible to other users.

A

ATOMICITY

All statements in a transaction succeed together, or none of them do. There is no partial commit.

C

CONSISTENCY

A transaction always moves the database from one valid state to another. Constraints and rules are enforced at commit.

I

ISOLATION

Concurrent transactions don't see each other's uncommitted changes. The level of isolation is configurable.

D

DURABILITY

Once committed, a transaction's changes survive a crash. InnoDB writes to the redo log before confirming success.

The transaction statements

```
START TRANSACTION;    -- or BEGIN; — marks the start of an explicit transaction
COMMIT;               -- make all changes permanent
```

```
ROLLBACK;              -- undo all changes since START TRANSACTION
SAVEPOINT name;        -- set a named checkpoint within the transaction
ROLLBACK TO name;      -- roll back to a savepoint without ending the transaction
RELEASE SAVEPOINT name; -- discard a savepoint (frees the name)
```

A complete purchase transaction

▶ **START TRANSACTION** — all writes from here on are held in a private "undo buffer". Other sessions still see the old data.

1 INSERT a new row into `orders`

2 UPDATE `books` to reduce stock by the quantity ordered

✓ **COMMIT** — both changes become permanent and visible to everyone. The undo buffer is discarded.

✗ **Or: ROLLBACK** — if anything went wrong (foreign key error, constraint violation, application crash), both changes are undone as if the transaction never happened.

```
START TRANSACTION;

-- Step 1: create the order
INSERT INTO orders (customer_id, book_id, quantity, total_price, order_date)
VALUES (2, 1, 1, 8.99, NOW());

-- Step 2: reduce stock
UPDATE books
SET    stock = stock - 1
WHERE  book_id = 1;

-- Only commit if both succeeded
COMMIT;
```

```
-- If something goes wrong, roll everything back
START TRANSACTION;

INSERT INTO orders (customer_id, book_id, quantity, total_price, order_date)
VALUES (99, 1, 1, 8.99, NOW());  -- customer 99 doesn't exist!

ERROR 1452 (23000): Cannot add or update a child row:
a foreign key constraint fails (`bookshop`.`orders`,
```

```
  CONSTRAINT `fk_orders_customer` ...)

ROLLBACK;    -- nothing was changed — the database is untouched
```

SAVEPOINT — partial rollbacks

Within a transaction you can plant named savepoints. Rolling back to a savepoint undoes everything after it while keeping everything before it. The transaction remains open — you can continue adding more statements or commit/rollback the whole thing:

```
START TRANSACTION;

INSERT INTO authors (first_name, last_name) VALUES ('Terry', 'Pratchett');

SAVEPOINT author_inserted;          -- checkpoint after inserting the author

INSERT INTO books (author_id, title, price)
VALUES (LAST_INSERT_ID(), 'Good Omens', 8.99);   -- oops wrong price

ROLLBACK TO author_inserted;        -- undo only the book INSERT
-- Terry Pratchett is still inserted; only the book was rolled back

INSERT INTO books (author_id, title, price)
VALUES (LAST_INSERT_ID(), 'Good Omens', 7.99);   -- correct price

COMMIT;
```

Autocommit — the default behaviour

By default MySQL runs in **autocommit** mode — every statement is automatically wrapped in its own transaction and committed immediately. This is why a bare `DELETE FROM orders WHERE order_id = 5` takes effect straight away with no `COMMIT` needed.

Mode	How it works	When to use
`Autocommit ON (default)`	Each statement commits immediately. You can still use START TRANSACTION to open a multi-statement transaction — autocommit is suspended until COMMIT or ROLLBACK.	Most application code — quick one-off queries, reads, single-table inserts.
`Autocommit OFF`	`SET autocommit = 0` — every statement is implicitly inside a transaction. You must COMMIT manually or changes are lost	Rarely used directly. Some ORMs and connection pools

Mode	How it works	When to use
	when the session closes.	manage this for you.

DDL statements auto-commit immediately and cannot be rolled back. `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE` implicitly commit the current transaction before executing. You cannot roll back a CREATE or DROP — even inside an explicit transaction. Always issue DDL outside of data-change transactions, and double-check before running in production.

CASE is SQL's way of writing if-then-else logic inside a query. It's an *expression* — it returns a value — so it can go anywhere a value is valid: the SELECT list, ORDER BY, WHERE, HAVING, and even inside aggregate functions.

Searched CASE (the common form)

```
CASE WHEN condition_1 THEN result_1    — evaluated top to bottom; first match wins WHEN condition_2 THEN result_2 WHEN condition_3 THEN result_3 ELSE default_result    — if no WHEN matched; ELSE is optional END    — required closing keyword
```

Simple CASE (equality only)

```
CASE column WHEN value_1 THEN result_1    — tests column = value_1 WHEN value_2 THEN result_2 ELSE default_result END
```

CASE evaluates top to bottom and stops at the first match. Put your most specific conditions first and the broadest catch-all last. If no WHEN matches and there's no ELSE, the result is NULL. Always add an ELSE unless NULL is a meaningful sentinel for "no match".

Stock status labels

```sql
SELECT
    title,
    stock,
    CASE
        WHEN stock =  0  THEN 'Out of stock'
        WHEN stock <  10 THEN 'Low stock'
        WHEN stock <  30 THEN 'In stock'
        ELSE                  'Well stocked'
    END AS stock_status
FROM   books
ORDER BY stock;
```

title	stock	stock_status
The Left Hand of Darkness	9	Low stock
Reasons to Stay Alive	18	In stock
Dune	27	In stock
Sapiens	31	Well stocked
The Midnight Library	42	Well stocked
And Then There Were None	55	Well stocked

CASE in ORDER BY — custom sort order

```sql
-- Sort by a business priority, not alphabetically or numerically
SELECT title, stock
FROM   books
ORDER BY
    CASE
        WHEN stock =  0  THEN 1    -- out of stock: highest priority
        WHEN stock <  10 THEN 2    -- low stock: second
        WHEN stock <  30 THEN 3
        ELSE                  4
    END,
    stock ASC;    -- secondary sort within each priority bucket
```

Simple CASE — mapping a code to a label

```sql
-- Map genre codes to display labels
SELECT
    title,
```

```
    CASE genre
        WHEN 'Non-fiction'       THEN '🔍 Non-fiction'
        WHEN 'Fiction'           THEN '📖 Fiction'
        WHEN 'Sci-fi'            THEN '🚀 Science Fiction'
        WHEN 'Mystery'           THEN '🔎 Mystery'
        ELSE                          '📚 Other'
    END AS genre_label
FROM   books;
```

One of the most powerful patterns in SQL: put a CASE expression *inside* an aggregate function to aggregate only the rows that match a condition. This lets you pivot rows into columns, compute multiple metrics in a single pass, and avoid repeating GROUP BY queries.

Counting by category in one query

```
-- How many books fall into each stock status — in one row
SELECT
    COUNT(CASE WHEN stock =  0  THEN 1 END) AS out_of_stock,
    COUNT(CASE WHEN stock <  10 AND stock > 0 THEN 1 END) AS low_stock,
    COUNT(CASE WHEN stock >= 10 THEN 1 END) AS in_stock
FROM   books;
```

out_of_stock	low_stock	in_stock
0	1	5

The CASE returns 1 when the condition matches and NULL otherwise. COUNT ignores NULLs — so it counts only the matching rows. No ELSE needed: the implicit ELSE is NULL, which COUNT skips.

Revenue split by genre in one row

```
SELECT
    SUM(CASE WHEN b.genre = 'Fiction'     THEN o.total_price END) AS fiction_rev,
    SUM(CASE WHEN b.genre = 'Mystery'     THEN o.total_price END) AS mystery_rev,
    SUM(CASE WHEN b.genre = 'Non-fiction' THEN o.total_price END) AS nonfiction_rev,
    SUM(CASE WHEN b.genre = 'Sci-fi'      THEN o.total_price END) AS scifi_rev,
    SUM(o.total_price)                                            AS total_rev
```

```
FROM   orders o
JOIN   books  b  ON o.book_id = b.book_id;
```

fiction_rev	mystery_rev	nonfiction_rev	scifi_rev	total_rev
17.98	20.97	18.98	9.99	67.92

Four different SUM calculations in a single table scan — no UNION or subqueries needed. This "pivot" pattern is extremely common in reporting queries.

Conditional aggregation with HAVING

```
-- Authors who have at least one book with low stock (< 15 copies)
SELECT
    CONCAT(a.first_name, ' ', a.last_name)   AS author,
    COUNT(b.book_id)                          AS total_books,
    COUNT(CASE WHEN b.stock < 15 THEN 1 END)  AS low_stock_titles
FROM     authors a
JOIN     books   b  ON a.author_id = b.author_id
GROUP BY a.author_id, a.first_name, a.last_name
HAVING   COUNT(CASE WHEN b.stock < 15 THEN 1 END) > 0
ORDER BY low_stock_titles DESC;
```

author	total_books	low_stock_titles
Ursula Le Guin	1	1

Using CASE in UPDATE — conditional bulk update

```
-- Apply different discounts based on genre in a single UPDATE
UPDATE books
SET    price = ROUND(
    price *
    CASE
        WHEN genre = 'Mystery'     THEN 0.85   -- 15% off Mystery
        WHEN genre = 'Sci-fi'      THEN 0.90   -- 10% off Sci-fi
        ELSE                            1.00   -- no change for others
    END,
2)
WHERE  genre IN ('Mystery', 'Sci-fi');   -- only affect these genres
```

Concept	Key points
START TRANSACTION / BEGIN	Opens an explicit transaction. Autocommit is suspended until COMMIT or ROLLBACK.
COMMIT	Makes all changes in the transaction permanent and visible to other sessions.
ROLLBACK	Undoes all changes since START TRANSACTION. Leaves the database as if the transaction never happened.
SAVEPOINT / ROLLBACK TO	Named checkpoints within a transaction for partial rollbacks. The transaction stays open after ROLLBACK TO.
ACID	Atomicity (all or nothing), Consistency (valid state to valid state), Isolation (concurrent transactions don't interfere), Durability (committed changes survive crashes).
Autocommit	On by default — each bare statement commits immediately. START TRANSACTION temporarily disables it for that block.
DDL and transactions	CREATE TABLE, ALTER TABLE, DROP TABLE implicitly commit the current transaction. Cannot be rolled back.
Searched CASE	CASE WHEN condition THEN result ... ELSE default END. First matching WHEN wins. ELSE is optional — no match without ELSE returns NULL.
Simple CASE	CASE col WHEN val THEN result ... END. Tests equality only. Shorter syntax for fixed value mappings.
Conditional aggregation	CASE inside SUM/COUNT/AVG. CASE returns a value or NULL; aggregate skips NULLs — effectively aggregating only matching rows. Powerful pivot pattern.

MySQL Foundations — Course Complete

You have completed all 10 chapters of MySQL Foundations.

You can now read, write, modify, and design relational databases with confidence.

Ch 1 · Relational Concepts | Ch 2 · SELECT Basics | Ch 3 · Sorting & Limiting

Ch 4 · Data Types & NULL | Ch 5 · Modifying Data | Ch 6 · CREATE TABLE

Ch 7 · Aggregates & GROUP BY | Ch 8 · JOINs | Ch 9 · String, Date & Numeric Functions

Ch 10 · Transactions & CASE