Module 12 of 16

Schema design

CREATE TABLE and choosing MySQL types; PRIMARY KEY, UNIQUE, NOT NULL, DEFAULT, CHECK and FOREIGN KEY; ALTER TABLE and DROP with safe migrations; normalization from 1NF to BCNF; one-to-many, many-to-many and self-referencing relationships; and views.

Queries are only as good as the tables under them. This module is about designing those tables: choosing a type for every column, declaring the rules the database must enforce, changing a live schema without breaking it, splitting data so each fact is stored once, connecting the pieces with foreign keys, and giving repeated queries a name with views. It is also one of the most interviewed areas of SQL — primary key versus unique key, the normal forms, DELETE versus TRUNCATE versus DROP, view versus table — and the module answers each of those directly, in MySQL 8 with InnoDB and with the PostgreSQL differences that come up.

CREATE TABLE and types covers integer sizes, `DECIMAL` versus `FLOAT` for money, `CHAR`, `VARCHAR` and `TEXT`, `DATE`, `DATETIME` and `TIMESTAMP`, MySQL's `BOOLEAN`, `ENUM` and `JSON`, AUTO_INCREMENT and `CREATE TABLE ... AS SELECT`. Constraints covers `NOT NULL`, `DEFAULT`, primary and composite keys, `UNIQUE`, natural and surrogate keys, `CHECK` and foreign keys. ALTER and DROP covers adding, renaming, redefining and dropping columns, renaming tables, adding constraints later, and the add-backfill-constrain migration. Normalization covers anomalies, functional dependencies and 1NF, 2NF, 3NF and BCNF on a worked example, and when to denormalize. Relationships covers one-to-one, one-to-many, many-to-many, self-references and `ON DELETE` actions. Views covers creating and replacing them, view versus table, updatable views, `WITH CHECK OPTION` and materialized views.

The exercises build and change real schemas and prove them with data: an events table whose defaults fill imported rows, a daily summary made with `CREATE TABLE ... AS SELECT`, a composite key and a CHECK that filter messy imports, a column added and backfilled, a legacy table cleaned up, a flat order sheet decomposed into third normal form, a many-to-many enrolment table, a manager hierarchy, a live revenue view and a redefined view. The checkpoint adds a library schema built from a loan log, a price-column migration and a course roster view.

CONTENTS

1	CREATE TABLE and data types	15 min
2	Constraints — the rules a table enforces	15 min
3	ALTER TABLE and DROP — changing a schema	14 min
4	Normalization — 1NF, 2NF, 3NF and BCNF	16 min
5	Relationships and foreign keys	15 min
6	Views — saved queries as tables	14 min
✓	Checkpoint — Schema design	25 min

Exercises in this module compile on MySQL 8 syntax on SQLite 3.49. Reference solutions and the answer key follow each lesson.

LESSON 1 · 15 MIN

CREATE TABLE and data types

The schema-design module moves from using tables to building them. This first lesson is the `CREATE TABLE` statement and the decision every column forces: its **data type**. A type decides what values a column accepts, how much space they take, how they compare and sort, and which mistakes the database catches for you. Picking well is cheap at design time and expensive to change later, when the table holds a hundred million rows. The lesson covers the syntax, MySQL's integer, decimal, string, date and boolean types with the choices interviewers ask about, AUTO_INCREMENT, and creating a table from a query.

The statement

```
CREATE TABLE customers (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  email       VARCHAR(255) NOT NULL,
  full_name   VARCHAR(100) NOT NULL,
  country     CHAR(2)      NOT NULL DEFAULT 'IN',
  credit      DECIMAL(10,2) NOT NULL DEFAULT 0.00,
  is_active   BOOLEAN      NOT NULL DEFAULT TRUE,
  born_on     DATE,
  created_at  DATETIME     NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

Each column is a name, a type and optional modifiers: `NOT NULL`, `DEFAULT value`, `PRIMARY KEY`, `UNIQUE`, `AUTO_INCREMENT`, `CHECK (...)`. Table-level constraints (a composite primary key, a foreign key) go after the columns. Conventions that pay off: lowercase `snake_case` names, singular or plural tables but consistently, `id` for the surrogate key and `<table>_id` for references to it, dates named `..._on` or `..._at`.

Two variants you will use constantly:

```
CREATE TABLE IF NOT EXISTS customers (...);   -- no error if it already exists
DROP TABLE IF EXISTS customers;               -- no error if it does not
```

In MySQL a table also has a storage engine (`ENGINE=InnoDB`, the default, with transactions and foreign keys) and a character set (`DEFAULT CHARSET=utf8mb4`, the default since 8.0,

which stores any Unicode including emoji). You see them in `SHOW CREATE TABLE` output; you rarely need to write them.

Integers

Type	Bytes	Signed range
`TINYINT`	1	−128 … 127
`SMALLINT`	2	−32,768 … 32,767
`INT`	4	about ±2.1 billion
`BIGINT`	8	about ±9.2 × 10^18

`UNSIGNED` doubles the positive range and forbids negatives (`INT UNSIGNED` reaches about 4.29 billion). Choose the smallest type that will comfortably hold the values for the life of the table — but for ids of tables that grow without limit (events, logs, orders on a large site) choose `BIGINT`: running out of `INT` ids is a real outage, and widening the column on a huge table is a long migration. The number in `INT(11)` is a display width, deprecated and meaningless for storage; ignore it.

Exact and approximate numbers

```
price     DECIMAL(10,2)   -- up to 99,999,999.99, exact
latitude  DOUBLE          -- approximate, 15-17 significant digits
```

`DECIMAL(p, s)` (also `NUMERIC`) stores exactly `p` digits, `s` of them after the point. It is the type for **money** and anything else that must add up exactly. `FLOAT` and `DOUBLE` are binary floating point: fast and compact, but 0.1 has no exact binary form, so `0.1 + 0.2` is `0.30000000000000004` and a column of prices summed in DOUBLE will not reconcile. Use them for measurements and scientific values. The interview answer: money is `DECIMAL`, never `FLOAT`. (A common alternative is storing integer paise or cents in a `BIGINT`.)

Strings

Type	Use
`CHAR(n)`	fixed length, padded: country codes, fixed-format codes
`VARCHAR(n)`	variable length up to n characters: names, emails, titles
`TEXT` (and `MEDIUMTEXT`, `LONGTEXT`)	long text: descriptions, comments, articles

`n` counts **characters**, not bytes, so `VARCHAR(100)` in utf8mb4 may take up to 400 bytes. `CHAR` pads with spaces and strips them on read. `TEXT` columns live partly off the row, cannot take a plain `DEFAULT` before 8.0.13 and can be indexed only by a prefix, so prefer `VARCHAR` for anything you will filter, sort or join on. Pick limits that reflect real rules — an email address is at most 254 characters, so `VARCHAR(255)` is the classic choice. In strict mode MySQL rejects a string longer than the column; non-strict mode silently truncates, one reason strict mode is the default.

Comparison and sorting of strings follow the column's **collation**. MySQL's default (`utf8mb4_0900_ai_ci`) is case- and accent-insensitive, so `'ADA' = 'ada'` is true and a UNIQUE email column rejects both. PostgreSQL compares case-sensitively by default — a classic surprise when moving an application.

Dates and times

Type	Holds	Notes
`DATE`	`'2024-05-02'`	a calendar day
`TIME`	`'14:30:00'`	a time of day or a duration
`DATETIME`	`'2024-05-02 14:30:00'`	as written, no time zone, years 1000–9999
`TIMESTAMP`	same shape	stored in UTC, converted per session time zone, ends 2038-01-19
`YEAR`	`2024`	a year

Use `DATE` when there is no time (birthdays, due dates) — not a string, so comparisons, ranges and date functions work. Between `DATETIME` and `TIMESTAMP`: `TIMESTAMP` is convenient for "when was this row written" because it converts time zones and can default to `CURRENT_TIMESTAMP` and update itself with `ON UPDATE CURRENT_TIMESTAMP`; `DATETIME` suits times that mean a wall-clock value (a meeting at 10:00 in Mumbai) and dates past 2038. Many teams store everything in UTC `DATETIME` and convert in the application. Both accept fractional seconds, `DATETIME(3)` for milliseconds.

Booleans, ENUM and JSON

MySQL has no true boolean type. `BOOLEAN` is an alias for `TINYINT(1)`; `TRUE` and `FALSE` are 1 and 0, and nothing stops a 2 from being stored. Name boolean columns as questions (`is_active`, `has_paid`).

`ENUM('draft', 'published', 'archived')` stores one of a fixed list of strings compactly. It reads well, but adding a value means an `ALTER TABLE`, the values sort by their position in the list rather than alphabetically, and it is MySQL-specific; a small lookup table or a `VARCHAR` with a `CHECK` constraint is the portable alternative. `JSON` stores validated JSON documents with functions to query them — useful for genuinely variable attributes, a smell when used to avoid designing columns.

AUTO_INCREMENT

```
id INT AUTO_INCREMENT PRIMARY KEY
```

An `AUTO_INCREMENT` column must be a key, and there is at most one per table. InnoDB hands out increasing numbers, never reuses one, and leaves gaps after rollbacks and deletes. PostgreSQL uses `GENERATED ALWAYS AS IDENTITY` (or the older `SERIAL`); SQL Server uses `IDENTITY(1,1)`. Write the modifiers in the order shown above — `INT AUTO_INCREMENT PRIMARY KEY` — which is also the form the practice engine recognises.

Creating a table from a query

```sql
CREATE TABLE daily_totals AS
SELECT sale_date, COUNT(*) AS orders, SUM(amount) AS revenue
FROM sales
GROUP BY sale_date;
```

`CREATE TABLE ... AS SELECT` (CTAS) creates a table whose columns are the query's columns, named by their aliases, and fills it with the result. It copies **no** primary key, indexes, defaults or foreign keys, so it suits snapshots, reports and scratch tables. `CREATE TABLE new_t LIKE old_t` does the opposite: copies the structure with its indexes, and no rows.

Key takeaways

- `CREATE TABLE t (col TYPE modifiers, ...)`; `IF NOT EXISTS` / `DROP TABLE IF EXISTS` for re-runnable scripts.
- Smallest integer that fits for life; `BIGINT` for ids that grow without limit.
- Money in `DECIMAL(p, s)`, never `FLOAT` / `DOUBLE`.
- `VARCHAR(n)` for most text, `CHAR(n)` for fixed codes, `TEXT` for long content; collation decides case sensitivity.
- `DATE` for days, `DATETIME` for wall-clock times, `TIMESTAMP` (UTC, ends 2038) for row times.
- `BOOLEAN` is `TINYINT(1)`; `ENUM` and `JSON` exist but have costs.
- CTAS copies data and column types, not keys or indexes.

EXERCISE 1

Create the events table

Write the `CREATE TABLE` for `events` so that the import already in the starter works and fills in what the raw data lacks:

column	type and rules
`id`	integer, numbered automatically (`INT AUTO_INCREMENT PRIMARY KEY`)
`title`	up to 100 characters, required
`starts_on`	a date, required
`capacity`	integer, required, default 50
`status`	up to 10 characters, required, default `'draft'`

The starter then copies `raw_events(title, starts_on, venue)` into it, ordered by `starts_on` then `title`, and prints `id, title, starts_on, capacity, status` ordered by `id`. Leave those two statements as they are.

`raw_events` : (Data Meetup, 2024-07-12, Hall A), (SQL Workshop, 2024-06-20, Lab 2), (Hack Night, 2024-07-12, Hall B).

```
id | title | starts_on | capacity | status
1 | SQL Workshop | 2024-06-20 | 50 | draft
2 | Data Meetup | 2024-07-12 | 50 | draft
3 | Hack Night | 2024-07-12 | 50 | draft
```

INPUT 1

```
CREATE TABLE raw_events (
  title VARCHAR(100) NOT NULL,
  starts_on DATE NOT NULL,
  venue VARCHAR(50)
);
INSERT INTO raw_events VALUES ('Data
Meetup', '2024-07-12', 'Hall A'), ('SQL
Workshop', '2024-06-20', 'Lab 2'),
('Hack Night', '2024-07-12', 'Hall B');
```

OUTPUT 1

```
id | title | starts_on | capacity |
status
1 | SQL Workshop | 2024-06-20 | 50 |
draft
2 | Data Meetup | 2024-07-12 | 50 |
draft
3 | Hack Night | 2024-07-12 | 50 | draft
```

REFERENCE SOLUTION

```sql
CREATE TABLE events (
  id INT AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR(100) NOT NULL,
  starts_on DATE NOT NULL,
  capacity INT NOT NULL DEFAULT 50,
  status VARCHAR(10) NOT NULL DEFAULT 'draft'
);

INSERT INTO events (title, starts_on)
SELECT title, starts_on
FROM raw_events
ORDER BY starts_on, title;

SELECT id, title, starts_on, capacity, status
FROM events
ORDER BY id;
```

EXERCISE 2

A daily summary with CREATE TABLE AS SELECT

Create a table `daily_totals` from `sales` with one `CREATE TABLE ... AS SELECT`. It has one row per `sale_date` and the columns:

- `sale_date`;
- `orders` — the number of sales that day;
- `revenue` — the sum of `amount`;
- `avg_order` — the average amount, rounded to 2 places.

Then print the whole table: `sale_date, orders, revenue, avg_order`, ordered by `sale_date`.

`sales(id, sale_date, amount)`: (1, 2024-05-01, 12.50), (2, 2024-05-01, 7.25), (3, 2024-05-02, 40.00), (4, 2024-05-01, 30.00), (5, 2024-05-03, 9.99).

```
sale_date | orders | revenue | avg_order
2024-05-01 | 3 | 49.75 | 16.58
2024-05-02 | 1 | 40 | 40
2024-05-03 | 1 | 9.99 | 9.99
```

INPUT 1

```
CREATE TABLE sales (
  id INT PRIMARY KEY,
  sale_date DATE NOT NULL,
  amount DECIMAL(8,2) NOT NULL
);
INSERT INTO sales VALUES (1, '2024-05-
01', 12.50), (2, '2024-05-01', 7.25),
(3, '2024-05-02', 40.00), (4, '2024-05-
01', 30.00), (5, '2024-05-03', 9.99);
```

OUTPUT 1

```
sale_date | orders | revenue | avg_order
2024-05-01 | 3 | 49.75 | 16.58
2024-05-02 | 1 | 40 | 40
2024-05-03 | 1 | 9.99 | 9.99
```

REFERENCE SOLUTION

```sql
CREATE TABLE daily_totals AS
SELECT sale_date,
       COUNT(*) AS orders,
       SUM(amount) AS revenue,
       ROUND(AVG(amount), 2) AS avg_order
FROM sales
GROUP BY sale_date;

SELECT sale_date, orders, revenue, avg_order
FROM daily_totals
ORDER BY sale_date;
```

QUIZ

1. Which type should store a product price such as 1299.99?

A. `FLOAT`

B. `DOUBLE`

C. `DECIMAL(10,2)`

D. `VARCHAR(10)`

2. What is MySQL's `BOOLEAN` type?

A. A true boolean that accepts only TRUE, FALSE and NULL

B. An alias for `TINYINT(1)` — TRUE is 1, FALSE is 0, and other small integers are accepted

C. An alias for `BIT(1)`

D. A one-character string 'Y'/'N'

3. A column must hold appointment times that mean "10:00 in Mumbai" regardless of the server's time zone, including dates in 2045. Which type fits?

A. `TIMESTAMP`

B. `DATETIME`

C. `DATE`

D. `YEAR`

4. What does `CREATE TABLE copy AS SELECT * FROM orders;` copy?

A. Columns, rows, primary key and indexes

B. Columns and rows, but not the primary key, indexes, defaults or foreign keys

C. Only the structure, no rows

D. Only the primary key

5. What is the difference between `CHAR(10)` and `VARCHAR(10)`?

A. CHAR holds up to 10 bytes, VARCHAR up to 10 characters

B. CHAR is fixed length and pads with spaces; VARCHAR stores only the characters given plus a length prefix

C. VARCHAR cannot be indexed

D. There is no difference in MySQL

ANSWER KEY · CREATE TABLE AND DATA TYPES

1. C DECIMAL stores exact decimal digits, so sums reconcile. FLOAT/DOUBLE are binary approximations (0.1 + 0.2 ≠ 0.3); a string cannot be summed or compared numerically.

2. B MySQL maps BOOLEAN to TINYINT(1). PostgreSQL is the one with a real BOOLEAN type.

3. B DATETIME stores the value as written, with no time-zone conversion, through year 9999. TIMESTAMP converts via UTC and ends in January 2038.

4. B CTAS creates columns from the query and fills them. `CREATE TABLE copy LIKE orders` is the form that copies structure and indexes without rows.

5. B Both count characters. CHAR suits fixed-length codes; VARCHAR suits text of varying length.

LESSON 2 · 15 MIN

Constraints — the rules a table enforces

A data type stops a date from landing in an integer column. **Constraints** go further: they state rules about values and rows — this column is required, no two rows share this email, a quantity is never negative, every order points at a real customer — and the database rejects any statement that would break them. Rules enforced in the schema hold for every application, script and late-night manual fix that touches the table, which rules in application code cannot promise. This lesson covers the six constraints, the primary key vs unique key question, composite keys, naming, and what happens when a rule is broken.

NOT NULL and DEFAULT

```
CREATE TABLE products (
  id       INT AUTO_INCREMENT PRIMARY KEY,
  name     VARCHAR(80)  NOT NULL,
  price    DECIMAL(8,2) NOT NULL,
  in_stock INT          NOT NULL DEFAULT 0,
  notes    TEXT
);
```

`NOT NULL` makes a value required. Make it the default habit: a column should allow NULL only when "unknown" or "not applicable" is a real state for it, because every nullable column brings three-valued logic into every query that touches it. `DEFAULT` supplies the value when an INSERT omits the column (remember from the INSERT lesson that an explicit NULL is not "omitted"). `NOT NULL DEFAULT 0` together give a column that is always filled without every INSERT having to mention it.

PRIMARY KEY

The primary key identifies each row. It is **unique and NOT NULL** by definition, and a table has **one**. In InnoDB it is also the **clustered index**: rows are stored in primary-key order, and every secondary index stores the primary key to find the row. That is why a short, never-changing, ever-increasing key (an `AUTO_INCREMENT` integer) suits InnoDB, and a random UUID string as primary key spreads inserts across the whole table.

A key on one column is written inline; on several columns it is a table constraint:

```
CREATE TABLE enrolments (
  student_id  INT  NOT NULL,
  course_id   INT  NOT NULL,
  enrolled_on DATE NOT NULL,
  PRIMARY KEY (student_id, course_id)
);
```

This **composite key** says that the pair is unique: a student can take many courses and a course has many students, but the same student cannot be enrolled in the same course twice. The order of the columns matters for the index (it can serve lookups by `student_id` alone, not by `course_id` alone — the indexes module explains why).

UNIQUE

```
CREATE TABLE users (
  id       INT AUTO_INCREMENT PRIMARY KEY,
  email    VARCHAR(255) NOT NULL UNIQUE,
  username VARCHAR(30)  NOT NULL,
  phone    VARCHAR(20),
  CONSTRAINT uq_users_username UNIQUE (username),
  CONSTRAINT uq_users_phone UNIQUE (phone)
);
```

`UNIQUE` forbids two rows with the same value (or the same combination, for `UNIQUE (a, b)`). Unlike the primary key, a table can have many, and a unique column **may be NULL** — and since NULL is not equal to NULL, MySQL and PostgreSQL allow any number of NULLs in a unique column (SQL Server allows one). So `phone` above may be missing for many users but never shared.

Primary key vs unique key — the interview answer in one breath: both reject duplicates and both build an index; a table has exactly one primary key but any number of unique keys; the primary key can never be NULL, a unique key can hold NULLs; in InnoDB the primary key is the clustered index that orders the table, a unique key is a secondary index; foreign keys normally reference the primary key.

Natural and surrogate keys

An email address, an ISBN or a vehicle number is a **natural key**: the real world already guarantees it is unique. An `AUTO_INCREMENT` id is a **surrogate key**: meaningless, but stable. Natural keys change (people change email), can be long (every foreign key repeats them) and are sometimes less unique than promised. The common design is a surrogate primary key plus a UNIQUE constraint on the natural key — the database still refuses duplicates, and references stay small and permanent.

CHECK

```
CREATE TABLE order_lines (
  order_id INT NOT NULL,
  line_no  INT NOT NULL,
  qty      INT NOT NULL CHECK (qty > 0),
  discount DECIMAL(4,2) NOT NULL DEFAULT 0,
  PRIMARY KEY (order_id, line_no),
  CONSTRAINT chk_discount CHECK (discount BETWEEN 0 AND 0.5)
);
```

A `CHECK` constraint is a condition every row must satisfy. MySQL **enforces** them only since **8.0.16**; earlier versions accepted the syntax and ignored it, a fact that still trips teams on old servers. A check that evaluates to NULL passes — `CHECK (qty > 0)` lets a NULL qty through — so pair it with `NOT NULL` for required columns. Checks can compare columns of the same row (`CHECK (ends_on ⩾ starts_on)`) but cannot query other tables.

FOREIGN KEY

```
CREATE TABLE orders (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  customer_id INT NOT NULL,
  placed_on   DATE NOT NULL,
  CONSTRAINT fk_orders_customer
    FOREIGN KEY (customer_id) REFERENCES customers (id)
    ON DELETE RESTRICT ON UPDATE CASCADE
);
```

A foreign key says every `customer_id` must exist as a `customers.id`: InnoDB refuses an order for a missing customer and refuses deleting a customer who still has orders (or cascades, per

the `ON DELETE` action). It is the subject of the relationships lesson. InnoDB requires an index on the referencing column and creates one if needed. (The practice engine here accepts foreign-key declarations but does not enforce them, so exercises never depend on a refusal.)

Naming constraints and what a violation looks like

`CONSTRAINT name` before a constraint gives it a name you choose (`uq_users_email`, `chk_discount`, `fk_orders_customer`); unnamed ones get generated names. Named constraints make error messages readable and can be dropped by name in a migration (`ALTER TABLE users DROP INDEX uq_users_phone`, `ALTER TABLE order_lines DROP CHECK chk_discount`).

A statement that breaks a constraint fails with an error and changes nothing:

```
ERROR 1062 (23000): Duplicate entry 'ada@example.com' for key 'users.email'
ERROR 1048 (23000): Column 'name' cannot be null
ERROR 3819 (HY000): Check constraint 'chk_discount' is violated.
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint
```

Applications catch these — a duplicate-key error on signup becomes "that email is already registered" — rather than checking first and racing. And `INSERT IGNORE` turns duplicate and check violations into skipped rows, which is a quick way to load data through the table's rules.

Key takeaways

- `NOT NULL` by default; `DEFAULT` fills omitted columns.
- One `PRIMARY KEY` per table: unique, never NULL, the clustered index in InnoDB; composite keys are declared at table level.
- `UNIQUE`: many per table, NULLs allowed (many of them in MySQL).
- Surrogate primary key plus UNIQUE natural key is the common design.
- `CHECK` is enforced from MySQL 8.0.16; NULL passes a check.
- `FOREIGN KEY ... REFERENCES` keeps references valid; name constraints so errors and migrations read well.

EXERCISE 1

A composite key keeps enrolments unique

Create the `enrolments` table so that a student can be enrolled in many courses and a course can have many students, but the **same student–course pair** can be stored only once:

- `student_id` integer, required;
- `course_id` integer, required;
- `enrolled_on` date, required;
- a **composite primary key** on `(student_id, course_id)`.

The starter then loads `enrolment_requests` with `INSERT IGNORE` in request order — a repeated pair is skipped by your key, so the earliest request wins — and prints `student_id, course_id, enrolled_on` ordered by `student_id`, `course_id`. Leave those statements as they are.

`enrolment_requests(id, student_id, course_id, requested_on)`:

id	student_id	course_id	requested_on
1	1	10	2024-01-05
2	1	11	2024-01-06
3	2	10	2024-01-06
4	1	10	2024-01-09
5	3	12	2024-01-10
6	2	10	2024-01-11

```
student_id | course_id | enrolled_on
1 | 10 | 2024-01-05
1 | 11 | 2024-01-06
2 | 10 | 2024-01-06
3 | 12 | 2024-01-10
```

INPUT 1

```
CREATE TABLE enrolment_requests (
  id INT PRIMARY KEY,
  student_id INT NOT NULL,
  course_id INT NOT NULL,
  requested_on DATE NOT NULL
);
INSERT INTO enrolment_requests VALUES
(1, 1, 10, '2024-01-05'), (2, 1, 11,
'2024-01-06'), (3, 2, 10, '2024-01-06'),
(4, 1, 10, '2024-01-09'), (5, 3, 12,
'2024-01-10'), (6, 2, 10, '2024-01-11');
```

OUTPUT 1

```
student_id | course_id | enrolled_on
1 | 10 | 2024-01-05
1 | 11 | 2024-01-06
2 | 10 | 2024-01-06
3 | 12 | 2024-01-10
```

REFERENCE SOLUTION

```sql
CREATE TABLE enrolments (
  student_id INT NOT NULL,
  course_id INT NOT NULL,
  enrolled_on DATE NOT NULL,
  PRIMARY KEY (student_id, course_id)
);

INSERT IGNORE INTO enrolments (student_id, course_id, enrolled_on)
SELECT student_id, course_id, requested_on
FROM enrolment_requests
ORDER BY id;

SELECT student_id, course_id, enrolled_on
FROM enrolments
ORDER BY student_id, course_id;
```

EXERCISE 2

Let UNIQUE and CHECK filter an import

Create the `accounts` table so its own rules reject the bad rows of an import:

- `id` integer primary key;
- `email` up to 100 characters, required, **unique**;
- `age` integer, required, with a **CHECK** that it is at least 13.

The starter then copies `raw_accounts` with `INSERT IGNORE` in id order — rows that break your UNIQUE or CHECK are skipped, so for a repeated email the lowest id wins — and prints `id, email, age` ordered by `id`. Leave those statements as they are.

`raw_accounts(id, email, age)` : (1, ana@x.io, 20), (2, bo@x.io, 12), (3, ana@x.io, 30), (4, cy@x.io, 13), (5, di@x.io, 45).

```
id | email | age
1 | ana@x.io | 20
4 | cy@x.io | 13
5 | di@x.io | 45
```

INPUT 1

```
CREATE TABLE raw_accounts (
  id INT PRIMARY KEY,
  email VARCHAR(100) NOT NULL,
  age INT NOT NULL
);
INSERT INTO raw_accounts VALUES (1,
'ana@x.io', 20), (2, 'bo@x.io', 12), (3,
'ana@x.io', 30), (4, 'cy@x.io', 13), (5,
'di@x.io', 45);
```

OUTPUT 1

```
id | email | age
1 | ana@x.io | 20
4 | cy@x.io | 13
5 | di@x.io | 45
```

REFERENCE SOLUTION

```
CREATE TABLE accounts (
  id INT PRIMARY KEY,
  email VARCHAR(100) NOT NULL UNIQUE,
  age INT NOT NULL,
  CONSTRAINT chk_accounts_age CHECK (age ⩾ 13)
);

INSERT IGNORE INTO accounts (id, email, age)
SELECT id, email, age
FROM raw_accounts
ORDER BY id;

SELECT id, email, age
FROM accounts
ORDER BY id;
```

QUIZ

1. Which statement about primary keys and unique keys is true in MySQL?

A. A table can have several primary keys and one unique key

B. A unique column may hold NULLs (any number of them); a primary key column never can

C. A unique key does not create an index

D. Only a unique key can be referenced by a foreign key

2. `qty INT CHECK (qty > 0)` — which value is accepted by the check?

A. 0

B. -1

C. NULL

D. None of them

3. Before which MySQL version were CHECK constraints parsed but silently ignored?

A. 5.5

B. 8.0.16

C. 8.4

D. They have always been enforced

4. A `users` table uses `id INT AUTO_INCREMENT PRIMARY KEY` and `email ... UNIQUE`. What is this design called and why is it common?

A. A composite key; it saves space

B. A surrogate primary key plus a unique natural key; references stay small and stable while duplicates are still refused

C. A natural primary key; emails never change

D. Denormalization; it avoids joins

5. How do you stop the same (student, course) pair from being stored twice in a junction table?

A. `UNIQUE` on `student_id` and `UNIQUE` on `course_id`

B. A composite `PRIMARY KEY (student_id, course_id)` (or `UNIQUE (student_id, course_id)`)

C. A CHECK constraint

D. An AUTO_INCREMENT id

ANSWER KEY · CONSTRAINTS — THE RULES A TABLE ENFORCES

1. B One primary key per table, never NULL, the clustered index in InnoDB. Many unique keys, each with an index, NULLs allowed because NULL ≠ NULL.

2. C `NULL > 0` is unknown, and a CHECK fails only when the condition is false. Add NOT NULL to require a value.

3. B Enforcement arrived in MySQL 8.0.16. Older servers accepted the syntax and enforced nothing.

4. B The surrogate id never changes and keeps foreign keys short; the UNIQUE constraint still enforces the real-world rule.

5. B The pair must be unique, not each column — separate UNIQUEs would allow each student only one course and each course only one student.

LESSON 3 · 14 MIN

ALTER TABLE and DROP — changing a schema

Schemas change as the application does: a new feature needs a column, a name turns out to be wrong, an old flag is retired. `ALTER TABLE` changes a table's structure in place, keeping its data; `DROP` removes a table (or view, or index) entirely. This lesson covers adding, renaming, redefining and dropping columns, renaming tables, adding and dropping constraints, and the migration habits — add, backfill, constrain; re-runnable scripts; one change at a time — that keep schema changes from becoming outages.

Adding columns

```
ALTER TABLE customers ADD COLUMN phone VARCHAR(20);
ALTER TABLE customers ADD COLUMN tier VARCHAR(10) NOT NULL DEFAULT 'basic';
ALTER TABLE customers ADD COLUMN middle_name VARCHAR(50) AFTER first_name;
```

Existing rows receive the column's default, or NULL if it has none. That is why adding a `NOT NULL` column **without** a default fails on a non-empty table in strict mode — there is no value to give the old rows. Several changes can go in one statement, separated by commas (`ALTER TABLE t ADD COLUMN a INT, ADD COLUMN b INT;`), and MySQL then rebuilds the table at most once. `AFTER col` and `FIRST` place the column; without them it goes last. Column order has no effect on queries that name their columns, only on `SELECT *` output.

Backfilling: add, fill, constrain

When the new column's value depends on the row, the safe sequence is three steps:

```
ALTER TABLE customers ADD COLUMN full_name VARCHAR(101);          -- 1. add,

UPDATE customers                                                  -- 2. backf
SET full_name = CONCAT_WS(' ', first_name, last_name);

ALTER TABLE customers MODIFY full_name VARCHAR(101) NOT NULL;     -- 3. const
```

Step 3 fails if step 2 missed a row, which is exactly the check you want. On a big table, step 2 runs in batches. This pattern — expand, migrate data, then contract — is how teams change

schemas without downtime: the application is first changed to write both the old and new shape, data is migrated, and only then is the old column removed.

Renaming and redefining columns

```
ALTER TABLE customers RENAME COLUMN phone TO phone_number;                  --
ALTER TABLE customers MODIFY phone_number VARCHAR(30) NOT NULL DEFAULT '';  --
ALTER TABLE customers CHANGE phone_number mobile VARCHAR(30);               --
```

`RENAME COLUMN` changes only the name. `MODIFY` replaces the column's whole definition — type, nullability, default — so anything you leave out is lost (forgetting `NOT NULL` in a MODIFY makes the column nullable again). `CHANGE` is the older form that renames and redefines at once, requiring the full definition. Narrowing a type (`VARCHAR(100)` to `VARCHAR(20)` , `BIGINT` to `INT`) fails in strict mode if existing data does not fit — check with a query such as `SELECT MAX(CHAR_LENGTH(col))` first. PostgreSQL spells these `ALTER COLUMN col TYPE ...` , `ALTER COLUMN col SET NOT NULL` and `RENAME COLUMN` . (In this course's practice engine, `ADD COLUMN` , `RENAME COLUMN` , `DROP COLUMN` and `RENAME TO` run; `MODIFY` and `CHANGE` are for reading.)

Dropping columns

```
ALTER TABLE customers DROP COLUMN legacy_flag;
```

The column and its data are gone; there is no undo except a backup. Before dropping, make sure nothing reads it — views, reports, other services. A safer rollout first stops the application from using the column, waits a release, then drops it. MySQL also drops any index that consisted only of that column.

Renaming tables

```
ALTER TABLE tbl_users RENAME TO users;
RENAME TABLE users TO users_old, users_new TO users;     -- MySQL: several, at
```

`RENAME TABLE` can rename several tables in one atomic step, which is how a table rebuilt offline (`users_new`) replaces the live one: there is no instant at which `users` does not exist. Foreign keys follow the renamed table; views that name the old table do not and must be recreated.

Constraints and indexes after the fact

```
ALTER TABLE users ADD CONSTRAINT uq_users_email UNIQUE (email);
ALTER TABLE orders ADD CONSTRAINT fk_orders_customer
  FOREIGN KEY (customer_id) REFERENCES customers (id);
ALTER TABLE order_lines ADD CONSTRAINT chk_qty CHECK (qty > 0);
ALTER TABLE users DROP INDEX uq_users_email;
ALTER TABLE orders DROP FOREIGN KEY fk_orders_customer;
ALTER TABLE users ADD INDEX ix_users_created (created_at);
```

Adding a constraint validates the existing rows: a UNIQUE fails if duplicates already exist, a CHECK if any row breaks it. So clean the data first — find duplicates with `GROUP BY email HAVING COUNT(*) > 1`, fix or remove them, then add the constraint.

DROP TABLE, DROP VIEW, DROP INDEX

```
DROP TABLE IF EXISTS staging_import;
DROP VIEW IF EXISTS v_active_users;
DROP INDEX ix_users_created ON users;
```

`DROP TABLE` removes the structure, data, indexes and triggers of a table. Recall the comparison from the DELETE lesson: DELETE removes rows, TRUNCATE removes all rows and keeps the table, DROP removes the table. InnoDB refuses to drop a table that another table's foreign key references until the key is dropped. `IF EXISTS` makes the statement safe to re-run.

How ALTER behaves on a live table

In MySQL, every `ALTER TABLE` commits any open transaction implicitly and briefly takes a metadata lock on the table. Beyond that, the cost depends on the change: MySQL 8 adds a

column at the end **instantly** (`ALGORITHM=INSTANT`, a metadata change only), many index changes run **in place** while reads and writes continue, and some changes — a column type change, a new primary key — **copy** the whole table, which on a large table takes a long time and doubles its disk use. You can ask for an algorithm and get an error instead of a surprise: `ALTER TABLE t ADD COLUMN c INT, ALGORITHM=INSTANT;`. Very large tables are often changed with online schema-change tools that build a copy in the background and swap it in with `RENAME TABLE`.

Migration habits

- Keep schema changes in versioned scripts (migrations) run in order, never as hand edits on production.
- Make each script re-runnable or guarded: `IF EXISTS`, `IF NOT EXISTS`.
- One logical change per migration; add, backfill, then constrain.
- Never drop or rename what running code still reads.
- Test the migration on a copy of production-sized data to learn how long it takes.

Key takeaways

- `ADD COLUMN` gives old rows the default or NULL; a NOT NULL column needs a default or a backfill.
- Add nullable → backfill with UPDATE → `MODIFY ... NOT NULL` is the standard migration.
- `RENAME COLUMN` renames; `MODIFY` redefines (restate everything); `CHANGE` does both.
- `DROP COLUMN` and `DROP TABLE` destroy data; `IF EXISTS` keeps scripts re-runnable.
- `RENAME TABLE a TO b, c TO a` swaps tables atomically.
- Adding a constraint validates existing rows; clean the data first.

EXERCISE 1

Add, backfill and rename

Change `customers` in three steps:

1. add a column `full_name` (up to 101 characters);
2. backfill it with the first and last name separated by one space — a customer with no last name gets just the first name (use `CONCAT_WS`, which skips NULLs; `CONCAT` would return NULL);

3. rename `phone` to `phone_number`.

Then print `id, full_name, phone_number`, ordered by `id`.

id	first_name	last_name	phone
1	Asha	Rao	98200 11111
2	Ben	NULL	NULL
3	Chen	Li	98200 33333

```
id | full_name | phone_number
1 | Asha Rao | 98200 11111
2 | Ben | NULL
3 | Chen Li | 98200 33333
```

INPUT 1

```
CREATE TABLE customers (
  id INT PRIMARY KEY,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50),
  phone VARCHAR(20)
);
INSERT INTO customers VALUES (1, 'Asha',
'Rao', '98200 11111'), (2, 'Ben', NULL,
NULL), (3, 'Chen', 'Li', '98200 33333');
```

OUTPUT 1

```
id | full_name | phone_number
1 | Asha Rao | 98200 11111
2 | Ben | NULL
3 | Chen Li | 98200 33333
```

REFERENCE SOLUTION

```sql
ALTER TABLE customers ADD COLUMN full_name VARCHAR(101);

UPDATE customers
SET full_name = CONCAT_WS(' ', first_name, last_name);

ALTER TABLE customers RENAME COLUMN phone TO phone_number;

SELECT id, full_name, phone_number
FROM customers
ORDER BY id;
```

EXERCISE 2

Clean up a legacy table

The old `tbl_users` table needs tidying. In order:

1. drop the unused column `legacy_flag`;
2. rename the table to `users`;
3. add a column `status` (up to 10 characters, required, default `'active'`);
4. set `status` to `'dormant'` for users who signed up **before 2023-01-01**.

Then print every column of `users` with `SELECT *`, ordered by `id` — the columns are `id, username, signup_date, status` in that order.

id	username	legacy_flag	signup_date
1	asha	1	2022-03-10
2	ben	0	2023-05-01
3	chen	NULL	2022-12-31

```
id | username | signup_date | status
1 | asha | 2022-03-10 | dormant
2 | ben | 2023-05-01 | active
3 | chen | 2022-12-31 | dormant
```

INPUT 1

```
CREATE TABLE tbl_users (
  id INT PRIMARY KEY,
  username VARCHAR(30) NOT NULL,
  legacy_flag INT,
  signup_date DATE NOT NULL
);
INSERT INTO tbl_users VALUES (1, 'asha',
1, '2022-03-10'), (2, 'ben', 0, '2023-
05-01'), (3, 'chen', NULL, '2022-12-
31');
```

OUTPUT 1

```
id | username | signup_date | status
1 | asha | 2022-03-10 | dormant
2 | ben | 2023-05-01 | active
3 | chen | 2022-12-31 | dormant
```

REFERENCE SOLUTION

```sql
ALTER TABLE tbl_users DROP COLUMN legacy_flag;
ALTER TABLE tbl_users RENAME TO users;
ALTER TABLE users ADD COLUMN status VARCHAR(10) NOT NULL DEFAULT 'active';

UPDATE users
SET status = 'dormant'
WHERE signup_date < '2023-01-01';

SELECT * FROM users ORDER BY id;
```

QUIZ

1. `ALTER TABLE orders ADD COLUMN channel VARCHAR(10) NOT NULL;` fails on a table with rows (strict mode). Why?

A. VARCHAR columns cannot be added later

B. The existing rows would need a value and the column has no DEFAULT

C. NOT NULL can only be declared in CREATE TABLE

D. ADD COLUMN requires AFTER

2. What does `ALTER TABLE products MODIFY price DECIMAL(12,2);` do if `price` was `DECIMAL(10,2) NOT NULL DEFAULT 0`?

A. Widens the type and keeps NOT NULL and the default

B. Widens the type; the column becomes nullable with no default, because MODIFY replaces the whole definition

C. Fails: MODIFY cannot change types

D. Renames the column

3. Which MySQL statement swaps a rebuilt table in for the live one with no moment where the name is missing?

A. `DROP TABLE users; ALTER TABLE users_new RENAME TO users;`

B. `RENAME TABLE users TO users_old, users_new TO users;`

C. `TRUNCATE TABLE users; INSERT INTO users SELECT * FROM users_new;`

D. `ALTER TABLE users REPLACE users_new;`

4. You need to add `UNIQUE (email)` to a table that already contains duplicate emails. What happens?

A. MySQL keeps the first duplicate and deletes the rest

B. The ALTER fails; find and fix the duplicates first (e.g. GROUP BY email HAVING COUNT(*) > 1)

C. The constraint is added and applies only to new rows

D. The duplicates are set to NULL

5. What does `RENAME COLUMN` change in MySQL 8?

A. Only the column's name

B. The name and the type

C. The column's position

D. The table's name

ANSWER KEY · ALTER TABLE AND DROP — CHANGING A SCHEMA

1. B Old rows get the default or NULL. With NOT NULL and no default there is nothing to give them. Add a DEFAULT, or add it nullable, backfill, then MODIFY to NOT NULL.

2. B MODIFY restates the column from scratch; anything omitted is dropped. Repeat `NOT NULL DEFAULT 0` to keep them.

3. B RENAME TABLE with several pairs is atomic. The DROP-then-rename version leaves a gap, and the TRUNCATE version is slow and not atomic in MySQL.

4. B Adding a constraint validates the existing rows. Clean the data, then add the constraint.

5. A RENAME COLUMN keeps the definition. CHANGE renames and redefines (full definition required); MODIFY redefines without renaming.

LESSON 4 · 16 MIN

Normalization — 1NF, 2NF, 3NF and BCNF

A spreadsheet stores everything in one sheet: every order row repeats the customer's name and city and the product's name and price. A relational schema splits that into tables so each fact lives in one row of one table. **Normalization** is the method for that split, and the normal forms — 1NF, 2NF, 3NF, BCNF — are the checkpoints along the way. Interviewers ask about them constantly, usually with a table to normalize on the spot. This lesson covers the anomalies redundancy causes, functional dependencies, each normal form with a worked example, and when undoing some of it (denormalization) is the right call.

The problem: anomalies

Here is one wide table of orders, the kind an export or a first draft produces:

order_id	order_date	customer_email	customer_name	customer_city	sku	product_nam
1	2024-05-01	asha@x.io	Asha	Pune	P1	Pen
1	2024-05-01	asha@x.io	Asha	Pune	P2	Pad
2	2024-05-03	ben@x.io	Ben	Delhi	P1	Pen

It works until data changes:

- **Update anomaly** — Asha moves to Mumbai: her city must be changed on every row of every order she ever placed; miss one and the database disagrees with itself.
- **Insert anomaly** — a new product cannot be recorded until someone orders it, because every row is an order line.
- **Delete anomaly** — delete Ben's only order and the fact that Ben exists, and lives in Delhi, disappears with it.

All three come from the same cause: facts about customers and products are stored in rows about something else.

Normalizing: each fact moves to the one table it belongs in

orders

order_id	date	email
1	2024-05-01	asha@x.io
2	2024-05-03	ben@x.io

customers

email	name	city
asha@x.io	Asha	Mumbai
ben@x.io	Ben	Delhi

order_lines

order_id	sku	qty
1	P1	10
1	P2	2
2	P1	4

products

sku	product	price
P1	Pen	1.50
P2	Pad	3.00

1. The wide table repeats facts about other things in rows about order lines: Asha's email, name and city are copied onto all 2 of her rows, and the Pen's name and price onto both of its lines.
2. Update anomaly: Asha moves to Mumbai and only one of her rows is changed. The table now says she lives in two cities — the database disagrees with itself.
3. 2NF: the key is (order_id, sku), but the product columns depend on sku alone and the order columns on order_id alone. Split them out: each product and each order is now one row, and the repeated copies are gone.
4. 3NF: in orders, name and city depend on email, not on the key — a transitive dependency. Customers become their own table and orders keep only the pointer.
5. Now Asha's city is one cell. Moving her to Mumbai is one update, a product can exist before anyone orders it, and deleting an order no longer deletes its customer.

Functional dependencies

A **functional dependency** `X → Y` says: rows that agree on X always agree on Y. In the table above:

- `customer_email → customer_name, customer_city`
- `sku → product_name, unit_price`
- `order_id → order_date, customer_email`
- `(order_id, sku) → qty`

The key of this table is `(order_id, sku)` — it determines everything. Each normal form is a rule about which dependencies may exist in a table, and normalizing means splitting a table along the dependencies that break the rule. Dependencies come from the business rules, not from the sample data: three rows cannot prove that `sku → unit_price`, only the business can (and if prices change over time, it does not — more on that below).

First normal form (1NF)

A table is in 1NF when every column holds a single, atomic value and there are no repeating groups. These violate it:

```
order_id | items
1        | Pen x10, Pad x2              -- a list in one cell

order_id | item1 | qty1 | item2 | qty2  -- repeating columns
```

A list in a cell cannot be indexed, joined or constrained, and searching it means string matching. Repeating columns cap the number of items and make "total quantity" a sum over columns. The fix is one row per item — which is what the wide table above already does, so it is in 1NF.

Second normal form (2NF)

2NF: in 1NF, and **no non-key column depends on only part of a composite key** (no *partial dependency*). In our table the key is `(order_id, sku)`, but `product_name` and `unit_price` depend on `sku` alone, and `order_date` and the customer columns depend on `order_id` alone. Split them out:

```
orders(order_id PK, order_date, customer_email, customer_name, customer_city)
products(sku PK, product_name, unit_price)
order_lines(order_id, sku, qty, PRIMARY KEY (order_id, sku))
```

Now each product's name and price is stored once. A table whose primary key is a single column is automatically in 2NF; the form only bites on composite keys.

Third normal form (3NF)

3NF: in 2NF, and **no non-key column depends on another non-key column** (no *transitive dependency*). In `orders`, `customer_name` and `customer_city` depend on `customer_email`, which is not the key — the chain is `order_id → customer_email → customer_city`. Split again:

```
customers(customer_id PK, email UNIQUE, name, city)
orders(order_id PK, order_date, customer_id)
products(product_id PK, sku UNIQUE, name, unit_price)
order_lines(order_id, product_id, qty, PRIMARY KEY (order_id, product_id))
```

The three anomalies are gone: Asha's city is one row, a product can exist with no orders, deleting an order leaves the customer. A good one-line summary of 1NF to 3NF, often quoted: every non-key column depends on **the key, the whole key, and nothing but the key**.

Boyce–Codd normal form (BCNF)

BCNF tightens 3NF: for **every** functional dependency `X → Y`, X must be a candidate key (a minimal set of columns that uniquely identifies rows). 3NF allows one exception — when Y is itself part of a candidate key — and BCNF does not. The difference only appears with overlapping composite candidate keys. The textbook case: each tutor teaches exactly one subject, and a student has one tutor per subject.

```
tutoring(student, subject, tutor)
-- (student, subject) → tutor        candidate key: (student, subject)
-- tutor → subject                    tutor is not a candidate key
```

The table is in 3NF (subject is part of a candidate key), but not BCNF, and the anomaly is real: the fact "Ravi teaches maths" is repeated for each of Ravi's students. The BCNF decomposition is `tutors(tutor PK, subject)` and `tutoring(student, tutor)`. In practice most tables in 3NF are already in BCNF. Higher forms (4NF for independent multi-valued facts, 5NF for join dependencies) exist and are rarely asked beyond their names.

A judgement, not a reflex: what the data means

Normalization follows the business rules, and those include **time**. In the decomposition above, `order_lines` gets the price from `products`. But prices change; last month's order must keep the price paid. So the line stores its own `unit_price` — not redundancy, but a different fact: "the price on this order", which `order_id, product_id → unit_price` captures correctly. The same goes for a shipping address copied onto an order. Ask what each value *means* before deciding it is a duplicate.

Denormalization

Normalized schemas write cleanly and read with joins. Sometimes the reads win and you store something twice on purpose:

- a reporting table or data warehouse star schema, rebuilt from the normalized tables;
- a cached aggregate such as `orders.total` or `posts.comment_count`, maintained on write;
- a copied column that saves a join on a very hot query.

Each copy must be kept consistent — by the application, a trigger, or a scheduled rebuild — and that cost is why the rule is: normalize first, measure, then denormalize specific paths deliberately.

Key takeaways

- Redundancy causes update, insert and delete anomalies; normalization stores each fact once.
- Functional dependencies (`X → Y`) from the business rules drive every split.
- 1NF: atomic values, no repeating groups.
- 2NF: no non-key column depends on part of a composite key.
- 3NF: no non-key column depends on another non-key column.
- BCNF: every determinant is a candidate key.
- A copied value with its own meaning (price at the time of order) is not redundancy; deliberate denormalization serves measured reads.

EXERCISE 1

Pull customers and products out of a flat sheet

`order_sheet` is an export with one row per order line, repeating every customer's and product's details:

order_id	order_date	customer_email	customer_name	sku	product_name	unit_price	qty
1	2024-05-01	asha@x.io	Asha	P1	Pen	1.50	10
1	2024-05-01	asha@x.io	Asha	P2	Pad	3.00	2
2	2024-05-03	ben@x.io	Ben	P1	Pen	1.50	4
3	2024-05-04	asha@x.io	Asha	P3	Ink	7.25	1
3	2024-05-04	asha@x.io	Asha	P1	Pen	1.50	5

Store each customer and each product once:

- `customers(email VARCHAR(100) PRIMARY KEY, name VARCHAR(50) NOT NULL)`;
- `products(sku VARCHAR(10) PRIMARY KEY, name VARCHAR(50) NOT NULL, unit_price DECIMAL(8,2) NOT NULL)`.

Create both and fill them from the sheet with `INSERT ... SELECT DISTINCT`. Then print `email, name` from customers ordered by `email`, and `sku, name, unit_price` from products ordered by `sku`.

```
email | name
asha@x.io | Asha
ben@x.io | Ben

sku | name | unit_price
P1 | Pen | 1.5
P2 | Pad | 3
P3 | Ink | 7.25
```

INPUT 1

```
CREATE TABLE order_sheet (
  order_id INT NOT NULL,
  order_date DATE NOT NULL,
  customer_email VARCHAR(100) NOT NULL,
  customer_name VARCHAR(50) NOT NULL,
  sku VARCHAR(10) NOT NULL,
  product_name VARCHAR(50) NOT NULL,
  unit_price DECIMAL(8,2) NOT NULL,
  qty INT NOT NULL
);
INSERT INTO order_sheet VALUES (1,
'2024-05-01', 'asha@x.io', 'Asha', 'P1',
'Pen', 1.50, 10), (1, '2024-05-01',
'asha@x.io', 'Asha', 'P2', 'Pad', 3.00,
2), (2, '2024-05-03', 'ben@x.io', 'Ben',
'P1', 'Pen', 1.50, 4), (3, '2024-05-04',
'asha@x.io', 'Asha', 'P3', 'Ink', 7.25,
1), (3, '2024-05-04', 'asha@x.io',
'Asha', 'P1', 'Pen', 1.50, 5);
```

OUTPUT 1

```
email | name
asha@x.io | Asha
ben@x.io | Ben

sku | name | unit_price
P1 | Pen | 1.5
P2 | Pad | 3
P3 | Ink | 7.25
```

REFERENCE SOLUTION

```sql
CREATE TABLE customers (
  email VARCHAR(100) PRIMARY KEY,
  name VARCHAR(50) NOT NULL
);
CREATE TABLE products (
  sku VARCHAR(10) PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  unit_price DECIMAL(8,2) NOT NULL
);

INSERT INTO customers (email, name)
SELECT DISTINCT customer_email, customer_name
FROM order_sheet;

INSERT INTO products (sku, name, unit_price)
SELECT DISTINCT sku, product_name, unit_price
FROM order_sheet;

SELECT email, name FROM customers ORDER BY email;

SELECT sku, name, unit_price FROM products ORDER BY sku;
```

EXERCISE 2

Finish the decomposition and rebuild the report

`customers` and `products` already exist (as in the previous exercise). Finish moving `order_sheet` into third normal form:

- `orders(order_id INT PRIMARY KEY, order_date DATE NOT NULL, customer_email VARCHAR(100) NOT NULL)` — one row per order;
- `order_lines(order_id INT NOT NULL, sku VARCHAR(10) NOT NULL, qty INT NOT NULL, unit_price DECIMAL(8,2) NOT NULL, PRIMARY KEY (order_id, sku))` — one row per line, keeping the price paid on that order.

Create and fill both from the sheet. The starter's first SELECT counts their rows — keep it. Then write the report from the **new** tables (not from `order_sheet`): per order, `order_id, order_date, customer` (the customer's name from `customers`), `items` (total quantity) and `total` (sum of qty × unit_price), ordered by `order_id`.

```
order_rows | line_rows
3 | 5

order_id | order_date | customer | items | total
1 | 2024-05-01 | Asha | 12 | 21
2 | 2024-05-03 | Ben | 4 | 6
3 | 2024-05-04 | Asha | 6 | 14.75
```

INPUT 1

```
CREATE TABLE order_sheet (
  order_id INT NOT NULL,
  order_date DATE NOT NULL,
  customer_email VARCHAR(100) NOT NULL,
  customer_name VARCHAR(50) NOT NULL,
  sku VARCHAR(10) NOT NULL,
  product_name VARCHAR(50) NOT NULL,
  unit_price DECIMAL(8,2) NOT NULL,
  qty INT NOT NULL
);
CREATE TABLE customers (
  email VARCHAR(100) PRIMARY KEY,
  name VARCHAR(50) NOT NULL
);
CREATE TABLE products (
  sku VARCHAR(10) PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  unit_price DECIMAL(8,2) NOT NULL
);
INSERT INTO order_sheet VALUES (1,
'2024-05-01', 'asha@x.io', 'Asha', 'P1',
'Pen', 1.50, 10), (1, '2024-05-01',
'asha@x.io', 'Asha', 'P2', 'Pad', 3.00,
2), (2, '2024-05-03', 'ben@x.io', 'Ben',
'P1', 'Pen', 1.50, 4), (3, '2024-05-04',
'asha@x.io', 'Asha', 'P3', 'Ink', 7.25,
1), (3, '2024-05-04', 'asha@x.io',
'Asha', 'P1', 'Pen', 1.50, 5);
INSERT INTO customers VALUES
('asha@x.io', 'Asha'), ('ben@x.io',
'Ben');
INSERT INTO products VALUES ('P1',
'Pen', 1.50), ('P2', 'Pad', 3.00),
('P3', 'Ink', 7.25);
```

OUTPUT 1

```
order_rows | line_rows
3 | 5

order_id | order_date | customer | items
| total
1 | 2024-05-01 | Asha | 12 | 21
2 | 2024-05-03 | Ben | 4 | 6
3 | 2024-05-04 | Asha | 6 | 14.75
```

REFERENCE SOLUTION

```sql
CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  order_date DATE NOT NULL,
  customer_email VARCHAR(100) NOT NULL
);
CREATE TABLE order_lines (
  order_id INT NOT NULL,
  sku VARCHAR(10) NOT NULL,
  qty INT NOT NULL,
  unit_price DECIMAL(8,2) NOT NULL,
  PRIMARY KEY (order_id, sku)
);

INSERT INTO orders (order_id, order_date, customer_email)
SELECT DISTINCT order_id, order_date, customer_email
FROM order_sheet;

INSERT INTO order_lines (order_id, sku, qty, unit_price)
SELECT order_id, sku, qty, unit_price
FROM order_sheet;

SELECT (SELECT COUNT(*) FROM orders) AS order_rows,
       (SELECT COUNT(*) FROM order_lines) AS line_rows;

SELECT o.order_id,
       o.order_date,
       c.name AS customer,
       SUM(l.qty) AS items,
       SUM(l.qty * l.unit_price) AS total
FROM orders o
JOIN customers c ON c.email = o.customer_email
JOIN order_lines l ON l.order_id = o.order_id
GROUP BY o.order_id, o.order_date, c.name
ORDER BY o.order_id;
```

QUIZ

1. A customer moves city, and her city must be changed on 40 order rows; one is missed. Which anomaly is this?

A. Insert anomaly

B. Update anomaly

C. Delete anomaly

D. Phantom read

2. `order_lines(order_id, product_id, product_name, qty)` has primary key `(order_id, product_id)`. Which normal form does `product_name` break?

A. 1NF

B. 2NF

C. 3NF only

D. None

3. `employees(id PK, dept_id, dept_name)` — `dept_name` depends on `dept_id`. Which normal form is broken?

A. 1NF

B. 2NF

C. 3NF

D. None

4. A `tags` column holds `'sql,mysql,joins'`. Which normal form does that break, and what is the fix?

A. 3NF; move tags to another database

B. 1NF; store one row per tag in a separate table (e.g. `post_tags(post_id, tag)`)

C. BCNF; add an index

D. None; lists are atomic strings

5. What does BCNF require that 3NF does not?

A. No NULLs

B. Every determinant of a functional dependency is a candidate key, with no exception for prime attributes

C. A surrogate key in every table

D. No composite keys

ANSWER KEY · NORMALIZATION — 1NF, 2NF, 3NF AND BCNF

1. B The same fact stored many times must be updated everywhere, and a missed copy leaves the data contradicting itself.

2. B `product_name` depends on `product_id` alone — part of the composite key. That partial dependency is what 2NF forbids.

3. C The key is a single column, so 2NF holds, but `id → dept_id → dept_name` is a transitive dependency between non-key columns, which 3NF forbids.

4. B A list in a cell is not atomic: it cannot be indexed, joined or constrained per tag.

5. B 3NF allows X → Y when Y is part of a candidate key; BCNF does not. They differ only with overlapping candidate keys.

LESSON 5 · 15 MIN

Relationships and foreign keys

Normalization splits data into tables; **relationships** are how the pieces connect again. A relationship is stored as a column in one table holding the key of a row in another — a **foreign key** — and the foreign-key constraint makes the database guarantee that the reference is valid. This lesson covers the three cardinalities (one-to-one, one-to-many, many-to-many), where the key goes in each, self-referencing tables, what `ON DELETE` and `ON UPDATE` actions do, and how to read a relationship back with joins.

One-to-many

The most common shape. A customer has many orders; each order has one customer. The key goes on the **many** side:

```
CREATE TABLE customers (
  id   INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(80) NOT NULL
);

CREATE TABLE orders (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  customer_id INT NOT NULL,
  placed_on   DATE NOT NULL,
  FOREIGN KEY (customer_id) REFERENCES customers (id)
);
```

Every order row names its customer. The reverse — a list of order ids in the customer row — would be a repeating group, breaking 1NF. `NOT NULL` on `customer_id` makes the relationship **mandatory** (an order must have a customer); a nullable foreign key makes it **optional** (a ticket may or may not have an assignee). Cardinality and optionality are the two questions to ask of every relationship.

Reading it back is a join, and a LEFT JOIN when the "one" side may have no children:

```sql
SELECT c.name, COUNT(o.id) AS orders
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
GROUP BY c.id, c.name;
```

Many-to-many

A student takes many courses; a course has many students. Neither table can hold the other's key, so a third table — a **junction** (bridge, associative, link) table — holds one row per pairing:

```sql
CREATE TABLE students (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(80) NO
CREATE TABLE courses  (id INT AUTO_INCREMENT PRIMARY KEY, code VARCHAR(10) NO

CREATE TABLE enrolments (
  student_id  INT  NOT NULL,
  course_id   INT  NOT NULL,
  enrolled_on DATE NOT NULL,
  grade       CHAR(2),
  PRIMARY KEY (student_id, course_id),
  FOREIGN KEY (student_id) REFERENCES students (id),
  FOREIGN KEY (course_id)  REFERENCES courses (id)
);
```

The composite primary key stops a duplicate pairing and indexes lookups by student; add an index on `course_id` for lookups the other way. Facts about the **pairing** itself — when the student enrolled, the grade — belong in the junction table, because they depend on both keys. A many-to-many relationship is really two one-to-many relationships meeting at the junction. Reading it back is two joins:

```sql
SELECT s.name, c.code
FROM enrolments e
JOIN students s ON s.id = e.student_id
JOIN courses  c ON c.id = e.course_id
ORDER BY s.name, c.code;
```

One-to-one

Each user has at most one profile. Model it with a foreign key that is also unique — most simply, by making the child's primary key the foreign key:

```
CREATE TABLE user_profiles (
  user_id INT PRIMARY KEY,
  bio     TEXT,
  avatar  VARCHAR(255),
  FOREIGN KEY (user_id) REFERENCES users (id)
);
```

Why not just put the columns in `users`? One-to-one splits are used for columns that are large or rarely read (kept off the hot table), for sensitive data with different access rules, and for optional extensions (only some users are sellers, so `seller_details` exists only for them). Without one of those reasons, one table is simpler.

Self-referencing relationships

A foreign key can point at its own table, which models a hierarchy:

```
CREATE TABLE employees (
  id         INT AUTO_INCREMENT PRIMARY KEY,
  name       VARCHAR(80) NOT NULL,
  manager_id INT,
  FOREIGN KEY (manager_id) REFERENCES employees (id)
);
```

The head of the company has `manager_id` NULL. One level is read with a self join (`LEFT JOIN employees m ON m.id = e.manager_id`); the whole tree with a recursive CTE, as in the CTE module. Categories with subcategories, threaded comments and bill-of-materials parts are the same shape.

Foreign key actions

When a referenced row is deleted or its key changes, the constraint's actions decide what happens to the rows that reference it:

```
FOREIGN KEY (customer_id) REFERENCES customers (id)
  ON DELETE RESTRICT
  ON UPDATE CASCADE
```

Action	On the parent's delete or key change
`RESTRICT` / `NO ACTION`	refused while children reference it (the default)
`CASCADE`	the change follows: children deleted, or their key updated
`SET NULL`	children kept, their foreign key set to NULL (column must be nullable)
`SET DEFAULT`	parsed by MySQL but rejected by InnoDB

Choose per relationship. Order lines belong to their order: `ON DELETE CASCADE`. Orders are financial records that must be kept: `RESTRICT`, so a customer with orders cannot be deleted by accident. A ticket's assignee leaves: `SET NULL`. `ON UPDATE CASCADE` matters only if keys change, which surrogate keys never do.

Facts about InnoDB foreign keys interviewers like: both columns must have compatible types (an `INT` cannot reference a `BIGINT UNSIGNED`); the referenced column must be indexed, usually the primary key; InnoDB adds an index on the referencing column if none exists; and `SET FOREIGN_KEY_CHECKS = 0` turns checking off for a session, which bulk loads use and which can leave orphans behind. The practice engine here accepts the declarations but does not enforce them, so these exercises keep references valid themselves.

Polymorphic references: a warning

A `comments` table with `target_type` ('post' or 'photo') and `target_id` cannot have a foreign key — the id points into different tables depending on the type — so the database cannot keep it valid. Prefer one nullable foreign key per possible parent (`post_id`, `photo_id`, with a CHECK that exactly one is set), or a junction table per parent type.

Key takeaways

- One-to-many: the foreign key goes on the many side; NOT NULL makes it mandatory.
- Many-to-many: a junction table with a composite primary key and the pairing's own attributes.

- One-to-one: a unique foreign key, often the child's primary key; use it for a reason.
- Self-references model hierarchies; the root has NULL.
- `ON DELETE RESTRICT | CASCADE | SET NULL` per relationship; RESTRICT is the default.
- Polymorphic id columns cannot be foreign keys; avoid them.

EXERCISE 1

Build a many-to-many

Students take many courses and courses have many students. `enrol_raw(student_name, course_code)` lists enrolments by name — and may name the same pairing twice.

1. Create the junction table `enrolments(student_id INT NOT NULL, course_id INT NOT NULL)` with a composite primary key on the pair and foreign keys to `students(id)` and `courses(id)`.
2. Fill it from `enrol_raw`, translating names to ids with joins, each pairing once.
3. Print every course with its number of students — including courses nobody takes — as `code, students`, ordered by `students` descending, then `code`.

`students(id, name)`: (1, Asha), (2, Ben), (3, Chen). `courses(id, code)`: (1, DB101), (2, OS201), (3, NET301). `enrol_raw`: (Asha, DB101), (Ben, DB101), (Asha, OS201), (Chen, DB101), (Ben, OS201).

```
code | students
DB101 | 3
OS201 | 2
NET301 | 0
```

INPUT 1

```
CREATE TABLE students (
  id INT PRIMARY KEY,
  name VARCHAR(50) NOT NULL UNIQUE
);
CREATE TABLE courses (
  id INT PRIMARY KEY,
  code VARCHAR(10) NOT NULL UNIQUE
);
CREATE TABLE enrol_raw (
  student_name VARCHAR(50) NOT NULL,
  course_code VARCHAR(10) NOT NULL
);
INSERT INTO students VALUES (1, 'Asha'),
(2, 'Ben'), (3, 'Chen');
INSERT INTO courses VALUES (1, 'DB101'),
(2, 'OS201'), (3, 'NET301');
INSERT INTO enrol_raw VALUES ('Asha',
'DB101'), ('Ben', 'DB101'), ('Asha',
'OS201'), ('Chen', 'DB101'), ('Ben',
'OS201');
```

OUTPUT 1

```
code | students
DB101 | 3
OS201 | 2
NET301 | 0
```

REFERENCE SOLUTION

```sql
CREATE TABLE enrolments (
  student_id INT NOT NULL,
  course_id INT NOT NULL,
  PRIMARY KEY (student_id, course_id),
  FOREIGN KEY (student_id) REFERENCES students (id),
  FOREIGN KEY (course_id) REFERENCES courses (id)
);

INSERT INTO enrolments (student_id, course_id)
SELECT DISTINCT s.id, c.id
FROM enrol_raw r
JOIN students s ON s.name = r.student_name
JOIN courses c ON c.code = r.course_code;

SELECT c.code, COUNT(e.student_id) AS students
FROM courses c
LEFT JOIN enrolments e ON e.course_id = c.id
GROUP BY c.id, c.code
ORDER BY students DESC, c.code;
```

EXERCISE 2

Managers: a self-referencing key

`employees` stores each manager by **name**. Replace that with a self-referencing key:

1. add a column `manager_id INT`;
2. fill it with the manager's `id`. MySQL refuses an UPDATE whose subquery reads the table being updated (error 1093), so first copy `id, name` into a helper table with `CREATE TABLE emp_lookup AS SELECT ...`, update from it, then drop it;
3. drop the column `manager_name`.

Then print `SELECT * FROM employees ORDER BY id` (columns `id, name, manager_id`), and each employee with their manager's name as `employee, manager` (NULL for the top), ordered by the employee's `id`.

id	name	manager_name
1	Meera	NULL
2	Ravi	Meera
3	Sana	Meera
4	Tom	Ravi

```
id | name | manager_id
1 | Meera | NULL
2 | Ravi | 1
3 | Sana | 1
4 | Tom | 2

employee | manager
Meera | NULL
Ravi | Meera
Sana | Meera
Tom | Ravi
```

INPUT 1

```
CREATE TABLE employees (
  id INT PRIMARY KEY,
  name VARCHAR(50) NOT NULL UNIQUE,
  manager_name VARCHAR(50)
);
INSERT INTO employees VALUES (1,
'Meera', NULL), (2, 'Ravi', 'Meera'),
(3, 'Sana', 'Meera'), (4, 'Tom',
'Ravi');
```

OUTPUT 1

```
id | name | manager_id
1 | Meera | NULL
2 | Ravi | 1
3 | Sana | 1
4 | Tom | 2

employee | manager
Meera | NULL
Ravi | Meera
Sana | Meera
Tom | Ravi
```

REFERENCE SOLUTION

```sql
ALTER TABLE employees ADD COLUMN manager_id INT;

CREATE TABLE emp_lookup AS SELECT id, name FROM employees;

UPDATE employees
SET manager_id = (SELECT l.id FROM emp_lookup l WHERE l.name = employees.manager_nam
WHERE manager_name IS NOT NULL;

DROP TABLE emp_lookup;

ALTER TABLE employees DROP COLUMN manager_name;

SELECT * FROM employees ORDER BY id;

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

QUIZ

1. Each order belongs to one customer; a customer places many orders. Where does the foreign key go?

A. `customers.order_id`

B. `orders.customer_id`

C. In a junction table `customer_orders`

D. On both tables

2. Where does a student's grade in a particular course belong?

A. In `students`

B. In `courses`

C. In the junction table `enrolments(student_id, course_id, grade)`

D. In a separate `grades` table with only a grade column

3. `order_items.order_id` references `orders.id` with `ON DELETE CASCADE`. What does `DELETE FROM orders WHERE id = 7` do?

A. Fails while order 7 has items

B. Deletes order 7 and its items

C. Deletes order 7 and sets its items' order_id to NULL

D. Deletes nothing

4. How is a one-to-one relationship between `users` and `user_profiles` usually enforced?

A. A foreign key in `user_profiles` that is also its primary key (or UNIQUE)

B. A junction table

C. A CHECK constraint

D. It cannot be enforced

5. `comments(target_type, target_id)` points at posts or photos depending on `target_type`. What is the problem?

A. Nothing; it is the recommended design

B. `target_id` cannot be a foreign key, so the database cannot keep references valid

C. It breaks 1NF

D. It needs a composite primary key

ANSWER KEY · RELATIONSHIPS AND FOREIGN KEYS

1. B One-to-many puts the key on the many side. A list of order ids on the customer would break 1NF.

2. C The grade depends on both the student and the course — the pairing — so it is an attribute of the junction row.

3. B CASCADE propagates the delete to the children. RESTRICT/NO ACTION, the default, would refuse; SET NULL would orphan them deliberately.

4. A A unique foreign key allows at most one profile per user; making it the primary key does that and identifies the row.

5. B A foreign key references one table. Polymorphic ids leave integrity to the application; nullable `post_id` / `photo_id` columns with a CHECK are the enforceable alternative.

LESSON 6 · 14 MIN

Views — saved queries as tables

Some queries get written again and again: the join that gives each order its customer's name, the filter for active accounts, the report of revenue per customer. A **view** gives such a query a name and lets everyone select from it as if it were a table. This lesson covers creating, querying, replacing and dropping views, what a view is and is not (the view vs table question), when you can write through one, `WITH CHECK OPTION`, how MySQL executes them, and materialized views in the databases that have them.

Creating and querying a view

```
CREATE VIEW customer_revenue AS
SELECT c.id   AS customer_id,
       c.name,
       COUNT(o.id)               AS orders,
       COALESCE(SUM(o.total), 0) AS revenue
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
GROUP BY c.id, c.name;
```

Now the report is a "table":

```
SELECT name, revenue
FROM customer_revenue
WHERE revenue > 1000
ORDER BY revenue DESC;
```

The view's columns are the SELECT's columns, named by their aliases, so alias every expression (an unaliased `COUNT(o.id)` would become a column literally named `COUNT(o.id)`). You can also name them in the header: `CREATE VIEW customer_revenue (customer_id, name, orders, revenue) AS SELECT ...`. A view may select from tables, other views, joins, subqueries and CTEs; filters, joins and sorts you add when querying it combine with its own.

A view stores a query, not data

Every time you select from a view, the database runs its definition against the **current** base tables. So a view is always up to date — change an order and `customer_revenue` reflects it immediately — and costs no storage, but it is only as fast as the query underneath. There is nothing to refresh and nothing to index; to speed a view up, index its base tables.

View vs table, the interview answer: a table holds data; a view holds a SELECT. A table's rows persist until changed; a view's rows are computed on each read. A table can be indexed; a (MySQL) view cannot. A table accepts any INSERT/UPDATE/DELETE; a view only when it is updatable. Dropping a view loses no data; dropping a table does.

Why use views

- **Simplicity** — a complex join written once, correctly, and reused; reports and analysts select from `customer_revenue` instead of re-deriving it.
- **Security** — grant a reporting user access to a view that leaves out salary or personal columns, and no access to the base table.
- **A stable interface** — when a table is split or renamed, a view with the old name and shape keeps old queries working while they are migrated.
- **Consistency** — "active customer" defined once in `active_customers`, instead of five slightly different WHERE clauses.

The cost: views on views on views become slow and hard to reason about, because every layer is expanded at query time. Keep them shallow.

Changing and removing views

```
CREATE OR REPLACE VIEW active_customers AS
SELECT id, name, email FROM customers WHERE is_active = 1 AND deleted_at IS NU

ALTER VIEW active_customers AS SELECT ...;     -- same effect in MySQL

DROP VIEW IF EXISTS active_customers;
```

`CREATE OR REPLACE VIEW` (MySQL, PostgreSQL, Oracle) swaps the definition in one statement. The portable form, which every database accepts and which this course's practice engine runs, is `DROP VIEW IF EXISTS` followed by `CREATE VIEW`. A view is bound to its base tables by name: drop or rename a table and the view fails when next queried

("references invalid table"). PostgreSQL additionally refuses `CREATE OR REPLACE` if the new definition removes or retypes existing columns.

One MySQL detail: `SELECT *` in a view definition is expanded to the column list **when the view is created**, so a column added to the table later does not appear in the view until it is recreated. Name the columns.

Writing through a view

Some views are **updatable**: an INSERT, UPDATE or DELETE on the view is applied to the base table. In MySQL a view is updatable when each of its rows maps to exactly one row of one base table — no aggregates, `DISTINCT`, `GROUP BY`, `HAVING`, `UNION`, `LIMIT` or subquery in the select list. A join view can be updated as long as one statement changes columns of only one of its tables.

```
CREATE VIEW cheap_products AS
SELECT id, name, price FROM products WHERE price < 10
WITH CHECK OPTION;

UPDATE cheap_products SET price = 8 WHERE id = 3;    -- fine: still under 10
UPDATE cheap_products SET price = 25 WHERE id = 3;   -- refused by CHECK OPTI
```

Without `WITH CHECK OPTION`, the second update would succeed and the row would vanish from the view — a confusing result. The option makes the view's WHERE a rule for writes through it. Updatable views are used to give restricted writers a filtered slice of a table; most views in practice are read-only reports.

How MySQL runs a view

MySQL executes a view one of two ways. With the **MERGE** algorithm, it splices the view's definition into your query, so your WHERE is applied to the base tables directly and indexes are used as if you had written the join yourself. With **TEMPTABLE**, it first materializes the view's whole result into a temporary table and then applies your query to that — necessary for views with aggregates, `DISTINCT` or `UNION`, and slower, since your filter cannot reach the base tables' indexes. `EXPLAIN` shows which happened. The practical rule: a view that aggregates a large table is recomputed in full every time it is queried.

Materialized views

A **materialized view** stores its result as real rows, refreshed on demand or on a schedule. PostgreSQL has `CREATE MATERIALIZED VIEW ...` and `REFRESH MATERIALIZED VIEW [CONCURRENTLY] ...`; Oracle and SQL Server (indexed views) have their own. Reads are as fast as a table, at the price of staleness between refreshes. MySQL has no materialized views; the substitute is a summary table rebuilt by a scheduled job (`CREATE TABLE ... AS SELECT` into a new table and `RENAME TABLE` to swap it in) or maintained by the application.

Key takeaways

- `CREATE VIEW v AS SELECT ...` names a query; alias every column.
- A view stores the query, not rows: always current, no storage, as fast as its query, no indexes of its own.
- Use views for reuse, security, stable interfaces and one definition of a business rule; keep them shallow.
- `CREATE OR REPLACE VIEW` / `ALTER VIEW` in MySQL; `DROP VIEW IF EXISTS` + `CREATE VIEW` everywhere.
- Simple single-table views are updatable; `WITH CHECK OPTION` keeps writes inside the view.
- Materialized views store results (PostgreSQL, Oracle); MySQL uses summary tables instead.

EXERCISE 1

A live revenue view

Create a view `customer_revenue` with the columns `customer_id, name, orders, revenue`: every customer, the number of their orders, and the sum of their order totals (0, not NULL, for a customer with no orders).

The starter then records a new order of 50.00 for customer 1 and selects from the view — because a view runs its query on every read, the new order must show up. Leave those statements as they are; the result is `name, orders, revenue` ordered by `revenue` descending, then `name`.

`customers(id, name)`: (1, Asha), (2, Ben), (3, Chen). `orders(id, customer_id, total)`: (10, 1, 120.00), (11, 2, 80.50), (12, 1, 30.00).

```
name | orders | revenue
Asha | 3 | 200
Ben | 1 | 80.5
Chen | 0 | 0
```

INPUT 1

```
CREATE TABLE customers (
  id INT PRIMARY KEY,
  name VARCHAR(50) NOT NULL
);
CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer_id INT NOT NULL,
  total DECIMAL(10,2) NOT NULL
);
INSERT INTO customers VALUES (1,
'Asha'), (2, 'Ben'), (3, 'Chen');
INSERT INTO orders VALUES (10, 1,
120.00), (11, 2, 80.50), (12, 1, 30.00);
```

OUTPUT 1

```
name | orders | revenue
Asha | 3 | 200
Ben | 1 | 80.5
Chen | 0 | 0
```

REFERENCE SOLUTION

```sql
CREATE VIEW customer_revenue AS
SELECT c.id AS customer_id,
       c.name,
       COUNT(o.id) AS orders,
       COALESCE(SUM(o.total), 0) AS revenue
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
GROUP BY c.id, c.name;

INSERT INTO orders (id, customer_id, total) VALUES (1000, 1, 50.00);

SELECT name, orders, revenue
FROM customer_revenue
ORDER BY revenue DESC, name;
```

EXERCISE 2

Redefine a view

The setup already has a view `big_orders` showing `id, total` of orders of 100 or more. The business changed the rule: it should now show `id, customer, total` (in that order) of every order of **50 or more** that is **not cancelled**.

MySQL would do this with `CREATE OR REPLACE VIEW`; write the portable form, `DROP VIEW IF EXISTS` followed by `CREATE VIEW`. The starter's SELECT then reads the view, ordered by `total` descending, then `id`.

`orders(id, customer, total, status)`:

id	customer	total	status
1	Asha	120.00	paid
2	Ben	55.00	paid
3	Chen	75.00	cancelled
4	Dev	50.00	pending
5	Eve	20.00	paid

```
id | customer | total
1 | Asha | 120
2 | Ben | 55
4 | Dev | 50
```

INPUT 1

```
CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer VARCHAR(50) NOT NULL,
  total DECIMAL(10,2) NOT NULL,
  status VARCHAR(10) NOT NULL
);
CREATE VIEW big_orders AS
SELECT id, total FROM orders WHERE total
⩾ 100;
INSERT INTO orders VALUES (1, 'Asha',
120.00, 'paid'), (2, 'Ben', 55.00,
'paid'), (3, 'Chen', 75.00,
'cancelled'), (4, 'Dev', 50.00,
'pending'), (5, 'Eve', 20.00, 'paid');
```

OUTPUT 1

```
id | customer | total
1 | Asha | 120
2 | Ben | 55
4 | Dev | 50
```

REFERENCE SOLUTION

```sql
DROP VIEW IF EXISTS big_orders;

CREATE VIEW big_orders AS
SELECT id, customer, total
FROM orders
WHERE total >= 50
  AND status <> 'cancelled';

SELECT * FROM big_orders ORDER BY total DESC, id;
```

QUIZ

1. What does a (non-materialized) view store?

A. A copy of the rows its query returned when it was created

B. Its SELECT statement; rows are computed from the base tables on every read

C. An index on the base tables

D. Nothing at all — it is deleted after use

2. Which is the portable way to change a view's definition in every major database?

A. `ALTER TABLE v ...`

B. `DROP VIEW IF EXISTS v;` then `CREATE VIEW v AS ...`

C. `UPDATE VIEW v ...`

D. `REFRESH VIEW v`

3. `CREATE VIEW cheap AS SELECT id, price FROM products WHERE price < 10 WITH CHECK OPTION;` — what does `UPDATE cheap SET price = 25 WHERE id = 3` do?

A. Updates the product to 25 and it disappears from the view

B. Is refused, because the new row would no longer satisfy the view's WHERE

C. Updates only the view, not the table

D. Fails because views are read-only

4. Which view can MySQL update through?

A. `SELECT customer_id, SUM(total) FROM orders GROUP BY customer_id`

B. `SELECT DISTINCT city FROM customers`

C. `SELECT id, name, email FROM customers WHERE is_active = 1`

D. `SELECT id FROM a UNION SELECT id FROM b`

5. MySQL has no materialized views. What is the usual substitute?

A. An ordinary view

B. A summary table rebuilt on a schedule (or maintained by the application)

C. A temporary table

D. An index

ANSWER KEY · VIEWS — SAVED QUERIES AS TABLES

1. B A view is a named query. That is why it is always current and takes no storage, and why it is only as fast as its query.

2. B `CREATE OR REPLACE VIEW` and `ALTER VIEW` exist in MySQL (and OR REPLACE in PostgreSQL/Oracle), but DROP + CREATE works everywhere.

3. B WITH CHECK OPTION makes the view's condition a rule for writes through it. Without it, the update would succeed and the row would vanish from the view.

4. C Each row of a simple single-table filter maps to one base row. Aggregates, DISTINCT and UNION make a view read-only.

5. B A summary table stores precomputed results like a materialized view, refreshed by a job — e.g. CTAS into a new table and RENAME TABLE to swap it in.

CHECKPOINT · 25 MIN

Checkpoint — Schema design

This checkpoint covers the whole module: `CREATE TABLE` and choosing types (integers, `DECIMAL` for money, `VARCHAR` and `CHAR`, `DATE`, `DATETIME` and `TIMESTAMP`, `BOOLEAN`), `CREATE TABLE ... AS SELECT`; the constraints `NOT NULL`, `DEFAULT`, `PRIMARY KEY`, `UNIQUE`, `CHECK` and `FOREIGN KEY`; `ALTER TABLE` and `DROP`; normalization from 1NF to BCNF; one-to-many, many-to-many and self-referencing relationships with their `ON DELETE` actions; and views.

How it works. Twelve questions and three scripts. You need 70% on the questions and every script accepted to clear the module. You can retake it as often as you like; your best score counts. Each script builds or changes a schema and ends with the SELECT that shows the result.

Before you start, make sure you can answer these from memory:

- Why is money stored as `DECIMAL` and not `FLOAT`, and what does `BOOLEAN` really mean in MySQL?
- What does `CREATE TABLE ... AS SELECT` copy, and what does it leave behind?
- Primary key vs unique key: how many per table, and which may hold NULL?
- How do you add a NOT NULL column to a table that already has rows?
- What do 1NF, 2NF, 3NF and BCNF each forbid?
- Where does the foreign key go in a one-to-many relationship, and how is many-to-many stored?
- What is the difference between a view and a table, and how do you change a view's definition?

The three scripts are a library schema built from a flat loan log, a migration that adds, backfills and renames columns, and a reporting view over a many-to-many relationship.

EXERCISE 1

A library schema from a loan log

`loan_log(id, member_email, member_name, isbn, title, borrowed_on)` is a flat record of every loan. Normalize it into three tables:

- `members(id INT AUTO_INCREMENT PRIMARY KEY, email VARCHAR(100) NOT NULL UNIQUE, name VARCHAR(50) NOT NULL)`;
- `books(isbn VARCHAR(13) PRIMARY KEY, title VARCHAR(100) NOT NULL)`;
- `loans(member_id INT NOT NULL, isbn VARCHAR(13) NOT NULL, borrowed_on DATE NOT NULL)` with a primary key on all three columns and foreign keys to `members` and `books`.

Fill `members` in **email order** (so ids are numbered alphabetically by email), then `books`, then `loans` (translating emails to member ids with a join). Print the members as `id, email` ordered by `id`, then each member's loan count as `name, loans` ordered by `loans` descending, then `name`.

id	member_email	member_name	isbn	title	borrowed_on
1	ravi@lib.io	Ravi	9780132350884	Clean Code	2024-03-01
2	asha@lib.io	Asha	9780596007126	Head First Design Patterns	2024-03-02
3	ravi@lib.io	Ravi	9780596007126	Head First Design Patterns	2024-03-10
4	meera@lib.io	Meera	9780132350884	Clean Code	2024-03-12
5	ravi@lib.io	Ravi	9780132350884	Clean Code	2024-04-01

```
id | email
1 | asha@lib.io
2 | meera@lib.io
3 | ravi@lib.io

name | loans
Ravi | 3
Asha | 1
Meera | 1
```

INPUT 1

```
CREATE TABLE loan_log (
  id INT PRIMARY KEY,
  member_email VARCHAR(100) NOT NULL,
  member_name VARCHAR(50) NOT NULL,
  isbn VARCHAR(13) NOT NULL,
  title VARCHAR(100) NOT NULL,
  borrowed_on DATE NOT NULL
);
INSERT INTO loan_log VALUES (1,
'ravi@lib.io', 'Ravi', '9780132350884',
'Clean Code', '2024-03-01'), (2,
'asha@lib.io', 'Asha', '9780596007126',
'Head First Design Patterns', '2024-03-
02'), (3, 'ravi@lib.io', 'Ravi',
'9780596007126', 'Head First Design
Patterns', '2024-03-10'), (4,
'meera@lib.io', 'Meera',
'9780132350884', 'Clean Code', '2024-03-
12'), (5, 'ravi@lib.io', 'Ravi',
'9780132350884', 'Clean Code', '2024-04-
01');
```

OUTPUT 1

```
id | email
1 | asha@lib.io
2 | meera@lib.io
3 | ravi@lib.io

name | loans
Ravi | 3
Asha | 1
Meera | 1
```

REFERENCE SOLUTION

```sql
CREATE TABLE members (
  id INT AUTO_INCREMENT PRIMARY KEY,
  email VARCHAR(100) NOT NULL UNIQUE,
  name VARCHAR(50) NOT NULL
);
CREATE TABLE books (
  isbn VARCHAR(13) PRIMARY KEY,
  title VARCHAR(100) NOT NULL
);
CREATE TABLE loans (
  member_id INT NOT NULL,
  isbn VARCHAR(13) NOT NULL,
  borrowed_on DATE NOT NULL,
  PRIMARY KEY (member_id, isbn, borrowed_on),
  FOREIGN KEY (member_id) REFERENCES members (id),
  FOREIGN KEY (isbn) REFERENCES books (isbn)
);

INSERT INTO members (email, name)
SELECT DISTINCT member_email, member_name
FROM loan_log
ORDER BY member_email;

INSERT INTO books (isbn, title)
SELECT DISTINCT isbn, title
FROM loan_log;

INSERT INTO loans (member_id, isbn, borrowed_on)
SELECT m.id, l.isbn, l.borrowed_on
FROM loan_log l
JOIN members m ON m.email = l.member_email;

SELECT id, email FROM members ORDER BY id;

SELECT m.name, COUNT(*) AS loans
FROM members m
JOIN loans l ON l.member_id = m.id
GROUP BY m.id, m.name
ORDER BY loans DESC, m.name;
```

EXERCISE 2

Migrate the price column

`products` stores prices as integer cents and carries a dead column. Migrate it, in order:

1. add `price DECIMAL(10,2)`;

2. backfill it as `price_cents / 100`;
3. drop `price_cents`;
4. rename `title` to `name`;
5. drop `old_sku`;
6. add `currency CHAR(3) NOT NULL DEFAULT 'INR'`.

Then print `SELECT * FROM products ORDER BY id` — the columns are `id, name, price, currency`.

id	title	price_cents	old_sku
1	Pen	150	X1
2	Notebook	1999	X2
3	Bag	120000	NULL

```
id | name | price | currency
1 | Pen | 1.5 | INR
2 | Notebook | 19.99 | INR
3 | Bag | 1200 | INR
```

INPUT 1

```
CREATE TABLE products (
  id INT PRIMARY KEY,
  title VARCHAR(80) NOT NULL,
  price_cents INT NOT NULL,
  old_sku VARCHAR(10)
);
INSERT INTO products VALUES (1, 'Pen',
150, 'X1'), (2, 'Notebook', 1999, 'X2'),
(3, 'Bag', 120000, NULL);
```

OUTPUT 1

```
id | name | price | currency
1 | Pen | 1.5 | INR
2 | Notebook | 19.99 | INR
3 | Bag | 1200 | INR
```

REFERENCE SOLUTION

```sql
ALTER TABLE products ADD COLUMN price DECIMAL(10,2);
UPDATE products SET price = price_cents / 100;
ALTER TABLE products DROP COLUMN price_cents;
ALTER TABLE products RENAME COLUMN title TO name;
ALTER TABLE products DROP COLUMN old_sku;
ALTER TABLE products ADD COLUMN currency CHAR(3) NOT NULL DEFAULT 'INR';

SELECT * FROM products ORDER BY id;
```

EXERCISE 3

A course roster view

Create a view `course_roster` with the columns `code, student, enrolled_on` — one row per enrolment, joining `enrolments` to `courses` and `students`.

The starter then enrols Ben (student 2) in course 1 and reads the view twice: the roster of course `DB101` as `student` ordered by name, and the number of students per course as `code, students` ordered by `code`. Leave those statements as they are.

`students`: (1, Asha), (2, Ben), (3, Chen). `courses`: (1, DB101), (2, OS201). `enrolments(student_id, course_id, enrolled_on)`: (1, 1, 2024-01-10), (3, 1, 2024-01-12), (1, 2, 2024-01-15), (2, 2, 2024-01-20).

```
student
Asha
Ben
Chen

code | students
DB101 | 3
OS201 | 2
```

INPUT 1

```
CREATE TABLE students (
  id INT PRIMARY KEY,
  name VARCHAR(50) NOT NULL
);
CREATE TABLE courses (
  id INT PRIMARY KEY,
  code VARCHAR(10) NOT NULL UNIQUE
);
CREATE TABLE enrolments (
  student_id INT NOT NULL,
  course_id INT NOT NULL,
  enrolled_on DATE NOT NULL,
  PRIMARY KEY (student_id, course_id),
  FOREIGN KEY (student_id) REFERENCES
students (id),
  FOREIGN KEY (course_id) REFERENCES
courses (id)
);
INSERT INTO students VALUES (1, 'Asha'),
(2, 'Ben'), (3, 'Chen');
INSERT INTO courses VALUES (1, 'DB101'),
(2, 'OS201');
INSERT INTO enrolments VALUES (1, 1,
'2024-01-10'), (3, 1, '2024-01-12'), (1,
2, '2024-01-15'), (2, 2, '2024-01-20');
```

OUTPUT 1

```
student
Asha
Ben
Chen

code | students
DB101 | 3
OS201 | 2
```

REFERENCE SOLUTION

```sql
CREATE VIEW course_roster AS
SELECT c.code, s.name AS student, e.enrolled_on
FROM enrolments e
JOIN courses c ON c.id = e.course_id
JOIN students s ON s.id = e.student_id;

INSERT INTO enrolments (student_id, course_id, enrolled_on) VALUES (2, 1, '2024-02-0

SELECT student FROM course_roster WHERE code = 'DB101' ORDER BY student;

SELECT code, COUNT(*) AS students FROM course_roster GROUP BY code ORDER BY code;
```

CHECKPOINT QUESTIONS

1. Which type is right for an account balance in rupees and paise?

A. `FLOAT`

B. `DOUBLE`

C. `DECIMAL(12,2)`

D. `INT`

2. How many primary keys and unique keys can a MySQL table have?

A. One primary key; any number of unique keys

B. Any number of each

C. One of each

D. One primary key; no unique keys alongside it

3. Which DDL statement keeps the table but removes all its rows?

A. `DROP TABLE t`

B. `TRUNCATE TABLE t`

C. `ALTER TABLE t DROP COLUMN c`

D. `DROP VIEW t`

4. What is the safe way to add a required column whose value differs per existing row?

A. Add it NOT NULL with no default

B. Add it nullable, backfill it with UPDATE, then MODIFY it to NOT NULL

C. Drop and recreate the table

D. Add it with DEFAULT NULL NOT NULL

5. What does `CHECK (age ⩾ 18)` do with a row whose age is NULL?

A. Rejects it

B. Accepts it

C. Sets age to 18

D. Converts it to 0

6. `enrolments(student_id, course_id, student_name)` with key `(student_id, course_id)`. Which normal form is broken?

A. 1NF

B. 2NF

C. 3NF only

D. BCNF only

7. The rule "every non-key column depends on the key, the whole key, and nothing but the key" summarises which normal forms?

A. 1NF only

B. 1NF to 3NF

C. BCNF and 4NF

D. 5NF

8. Where is a many-to-many relationship between `authors` and `books` stored?

A. `books.author_id`

B. `authors.book_ids` as a comma-separated list

C. A junction table `book_authors(book_id, author_id)` with a composite key

D. It cannot be stored relationally

9. Which `ON DELETE` action keeps the child rows but clears their reference to a deleted parent?

A. `CASCADE`

B. `RESTRICT`

C. `SET NULL`

D. `NO ACTION`

10. Which statement about views is true?

A. A view is refreshed nightly

B. A view can be indexed like a table in MySQL

C. Selecting from a view runs its query against the current base tables

D. Dropping a view deletes the base tables' rows

11. A table's id is `INT` and the business expects billions of events. What is the risk and the fix?

A. No risk; INT is unlimited

B. INT tops out near 2.1 billion (4.29 billion unsigned); use BIGINT from the start

C. Use VARCHAR ids instead

D. Use DECIMAL ids

12. In MySQL, why can't `UPDATE employees SET manager_id = (SELECT id FROM employees m WHERE m.name = employees.manager_name)` run?

A. Self-references are not allowed in MySQL

B. MySQL refuses an UPDATE whose subquery reads the table being updated (error 1093); use a join, a derived table or a helper table

C. The subquery returns more than one column

D. manager_id must be declared NOT NULL first

ANSWER KEY · CHECKPOINT — SCHEMA DESIGN

1. C DECIMAL is exact. (Integer paise in a BIGINT is the other correct design; a plain INT of rupees loses the paise.)

2. A Exactly one primary key (possibly composite); unique keys are unlimited in practice.

3. B TRUNCATE empties the table and keeps its structure; DROP removes the table itself.

4. B Add → backfill → constrain. The final MODIFY also proves the backfill missed no row.

5. B The condition is unknown, and only a false condition violates a CHECK. Add NOT NULL to require a value.

6. B `student_name` depends on `student_id` alone, part of the composite key — a partial dependency.

7. B "The key" (1NF, a key exists and values are atomic), "the whole key" (2NF), "nothing but the key" (3NF).

8. C Each side is one-to-many with the junction table, whose composite key prevents duplicate pairings.

9. C SET NULL needs a nullable foreign-key column. CASCADE deletes children; RESTRICT/NO ACTION refuse the delete.

10. C A view stores only its query, so it is always current; dropping it loses no data and it has no indexes of its own.

11. B Running out of INT ids stops inserts; widening a huge table later is a long migration. BIGINT costs 4 more bytes per row.

12. B Error 1093: you can't specify the target table for update in FROM. A multi-table UPDATE with a self join, or a copied helper table, avoids it.

CodeKairo · SQL study plan · Module 12: Schema design. Solve the exercises online to track your progress and earn XP.