ScopeTreasure Courses

SQL COURSES · LESSON DB.6 · CODE SCOPE SERIES

Primary Key, Foreign Key + Design Ke Usool

Give every row an identity and every table a safe link: PRIMARY KEY, FOREIGN KEY, the errors that guard them, and three design rules.

Introduction

Our tables so far trust us completely. Nothing stops two students sharing `id = 1`. Nothing stops a fee row pointing at student 99, who does not exist. Real databases do not run on trust — they run on **keys**: rules the database itself enforces, every time, even when the app code has a bug.

The **PRIMARY KEY** gives every row a unique identity. The **FOREIGN KEY** guards the links between tables — the same links your JOINs travel on. This lesson adds both to our school tables, watches the database refuse bad data with its own errors, and closes with three design rules that prevent most beginner schema pain.

(Roman Urdu: Key = database ka apna qaida. App bhool jaye, database nahi bhoolti — duplicate id aur nakli link andar hi nahi jayenge.)

Key points

- **PRIMARY KEY = unique + never empty:** one column (usually `id`) whose value identifies the row. Two rows can share a name, a class, a city — never a primary key. The database rejects duplicates itself.
- **AUTO_INCREMENT makes ids by itself:** with `AUTO_INCREMENT`, each INSERT gets the next number (1, 2, 3...) without you counting. The normal pattern for id columns: `id INT PRIMARY KEY AUTO_INCREMENT`.
- **FOREIGN KEY = a checked link:** `fees.student_id` is declared a FOREIGN KEY referencing `students(id)`. From then on, a fee can only point at a student who exists — and a student who *has* fees cannot simply be deleted out from under them.

  (Roman Urdu: Foreign key wohi rassi hai jo JOIN bandhta hai — bas ab database khud is ki hifazat karti hai.)
- **The errors are the feature:** `ERROR 1062: Duplicate entry` and `ERROR 1452: foreign key constraint fails` are not the database being difficult — they are the database catching a data bug at the door, before it poisons every report.
- **Design rule 1 — one table, one topic, one key:** every table gets its own primary key; every topic gets its own table (DB.1's rule, now enforced).
- **Design rule 2 — store each fact once:** a student's name lives in `students` only. `fees` stores the *id*, not a copy of the name. One fact, one home — so a name correction is made in exactly one place.
- **Design rule 3 — choose types like a designer:** money DECIMAL, days DATE, ids INT — DB.2's rules applied at design time, plus: if two tables will ever be asked together, plan the linking key column from day one.

Code examples

Example 1 — A table with a real identity

```
CREATE TABLE students (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(50),
  class VARCHAR(10),
  marks INT,
  city VARCHAR(50)
);

INSERT INTO students (name, class, marks, city)
VALUES ('Ali Raza', '10-A', 88, 'Sahiwal');

INSERT INTO students (id, name, class, marks, city)
VALUES (1, 'Copy Cat', '10-A', 50, 'Lahore');
```

What it does: The new `students` declares `id` as PRIMARY KEY with AUTO_INCREMENT — the first INSERT omits the id entirely and the database assigns 1. The second INSERT tries to force `id = 1` again.

What you will see:

```
Query OK, 1 row affected        -- first INSERT: id 1 assigned automatically

ERROR 1062 (23000): Duplicate entry '1' for key 'students.PRIMARY'
```

The refusal *is* the lesson: identity is now guaranteed by the database, not by careful typing. In a register, two students numbered 1 is a clerical mess; here it is simply impossible.

Example 2 — A foreign key guards the link

```
CREATE TABLE fees (
  id INT PRIMARY KEY AUTO_INCREMENT,
  student_id INT,
  amount DECIMAL(10,2),
  status VARCHAR(20),
  FOREIGN KEY (student_id) REFERENCES students(id)
);

INSERT INTO fees (student_id, amount, status)
VALUES (99, 5000, 'Paid');
```

What it does: `fees` declares that its `student_id` must be a real `students.id`. The INSERT tries to attach a fee to student 99 — who does not exist.

What you will see:

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

The fee is refused. Before keys, this row would have saved quietly and every fee report would have carried a ghost. (A valid insert — student_id 1 — succeeds normally, exactly like DB.5's data.) The same guard works in reverse: deleting a student who has fee rows is blocked, so history cannot be orphaned by one careless DELETE.

Example 3 — The design rules, applied in one schema

```
CREATE TABLE students (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(50) NOT NULL,
  class VARCHAR(10),
  marks INT,
  city VARCHAR(50)
);

CREATE TABLE fees (
  id INT PRIMARY KEY AUTO_INCREMENT,
  student_id INT NOT NULL,
  amount DECIMAL(10,2) NOT NULL,
  paid_on DATE,
  FOREIGN KEY (student_id) REFERENCES students(id)
);
```

What it does: The full school schema, designed by the three rules: each table one topic with its own AUTO_INCREMENT key; each fact stored once (the fee carries `student_id`, never a copied name); types chosen per job (DECIMAL money, DATE day). Two small promises also appear — `NOT NULL` on the columns that must never be empty; DB.7 is devoted to such promises (constraints).

What you will see: two `Query OK` lines — and a schema you can now trust. DESCRIBE either table and the `Key` column finally has entries: `PRI` on the id columns, `MUL` on `fees.student_id` (a key that allows multiples — many fees may point at one student). The JOIN from DB.5 now travels on a link the database itself protects.

Quick recap

- **PRIMARY KEY:** unique, never empty — the row's identity. Pair it with **AUTO_INCREMENT** for ids.
- **FOREIGN KEY:** a declared link the database checks — no ghost references, no orphaned children (errors 1062 / 1452 are guards, not nuisances).
- Design rules: **one topic per table, each fact stored once, types chosen deliberately** — and plan the linking key from day one.

Common mistakes

- **No primary key "for now":** without identity, updating or deleting "the" row is guesswork and duplicates creep in. Every table gets a key from its first CREATE — adding one later, to dirty data, is painful.
- **Copying facts into the child table:** storing the student's *name* inside `fees` "to save a join" creates two homes for one fact; when they disagree, nobody knows which is true. Store the id; JOIN for the name.
  (Roman Urdu: Fact ki ek hi ghar hona chahiye — copy banayi to kal dono mein farq aayega.)
- **Text ids with meaning:** ids like `'STU-2026-001'` invite format fights forever. Let ids be plain AUTO_INCREMENT numbers; keep meaning in real columns.
- **Deleting the parent first:** with a foreign key, deleting a student who has fees fails — correctly. Handle the children first (or choose an ON DELETE behaviour — DB.7 shows the options), never "fix" it by dropping the key.

Practice

1. Design the schema for a library: `books` and `issues` (which book is with which student, issue date, return date). Write both CREATE TABLEs with primary keys, one foreign key, and a line of reasoning for each type.
2. A fee row exists for student 3. Write down, in order, what the database will say if you (a) insert another fee for student 3, (b) insert a fee for student 30, (c) delete student 3. Which are allowed, which are refused — and why is each answer good news?
3. Find the design fault: a `marks` table stores student_id, student_name, subject and marks. Which rule does it break, and what does the fixed schema look like?

Practice quiz

1. What is a PRIMARY KEY?

- The first column of every table, whatever it holds
- One column whose value is unique and never empty, identifying the row; two rows can share a name or city, never a primary key
- A password for the table
- The column with the most data

Answer: One column whose value is unique and never empty, identifying the row; two rows can share a name or city, never a primary key

2. What does a FOREIGN KEY like fees.student_id referencing students(id) guarantee?

- That fees load faster
- That a fee can only point at a student who exists, and a student who has fees cannot simply be deleted out from under them
- That the two tables merge into one

- That student names are copied into fees

Answer: That a fee can only point at a student who exists, and a student who has fees cannot simply be deleted out from under them

3. Where should a student's name live, and why?

- In students and copied into fees, to save a join
- In students only; fees stores the id, not a copy of the name, so a name correction is made in exactly one place
- In fees only, students holds the id
- In both, kept in sync by hand

Answer: In students only; fees stores the id, not a copy of the name, so a name correction is made in exactly one place

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/sql-keys-design · More free tools and notes at https://scopetreasure.com