**📜 Who Made the First Database?**

The **first database model** — called the **hierarchical database model** — was developed by **IBM** in the **1960s**.

**🧠 1. First True Database System:**

**IBM Information Management System (IMS)**

* 🏢 Developed by: **IBM**
* 📅 Year: **1966–1968**
* 💡 Purpose: To help manage the **Apollo space program** parts list (for NASA)

IMS was the **first commercial database management system (DBMS)**. It used a **tree-like (hierarchical)** structure.

**🔁 2. Invention of the Relational Model:**

The **relational database model** — the one we use today with SQL — was invented by:

**👨‍🏫 Dr. Edgar F. Codd**

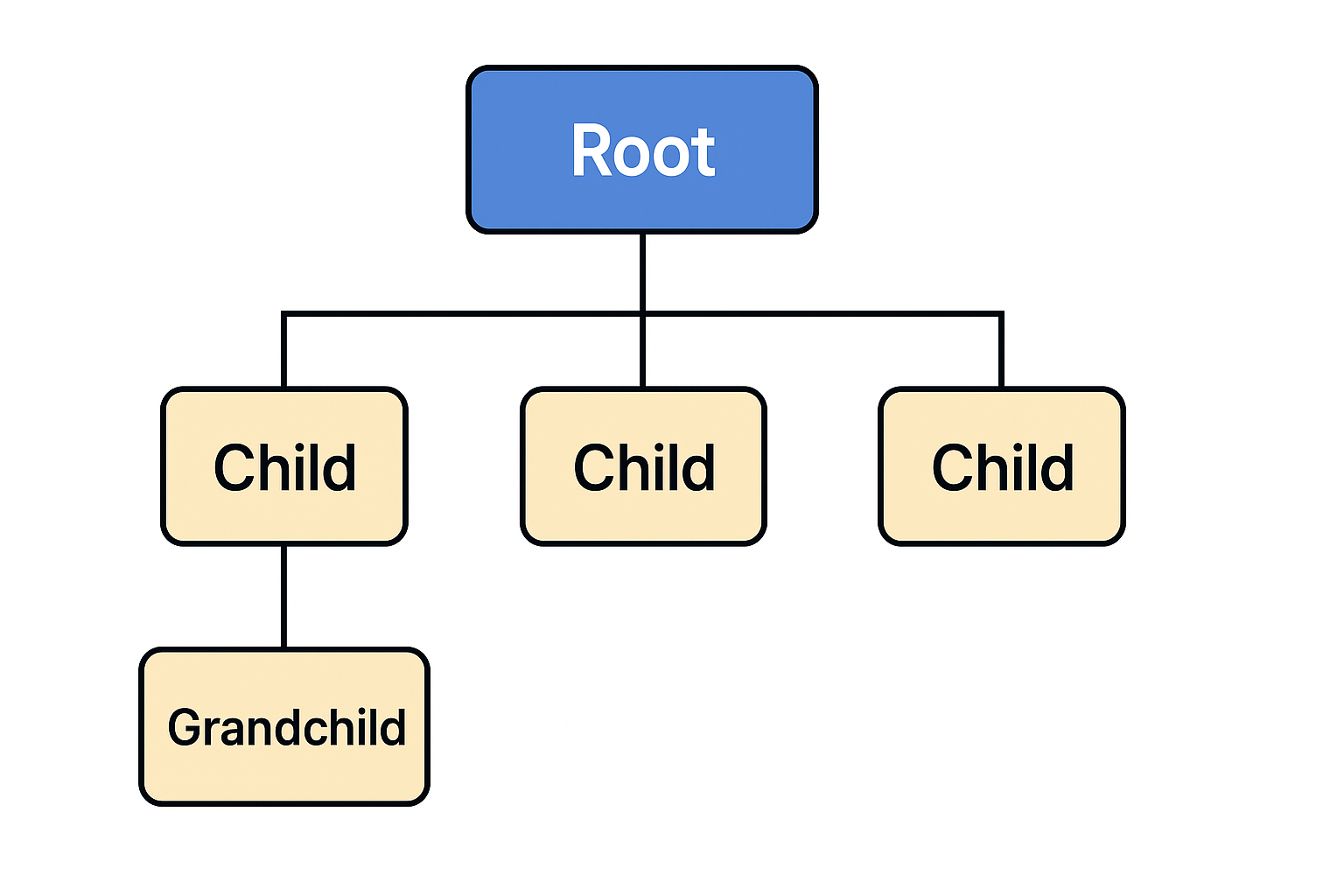
* 🏢 Worked at: IBM Research
* 📅 Year: **1970**
* 📄 Published: A paper titled **"A Relational Model of Data for Large Shared Data Banks"**

This was a **revolution** — instead of tree structures, he proposed **storing data in tables** (relations) that are linked using **keys**.

**🏆 Result:**

* SQL was created based on Dr. Codd’s model (by IBM in the 1970s).
* Almost every modern database (MySQL, Oracle, PostgreSQL, Azure SQL) is based on **Codd’s relational model**.

A diagram of a tree structure

AI-generated content may be incorrect.

Explore in **detailed and easy-to-understand language** how the **Hierarchical (Tree-like) Database Model** worked in real production environments back in the 1960s–1980s.

**🌳 What is the Hierarchical Database Model?**

A **Hierarchical Database** organizes data in a **tree-like structure**:

* **Each parent can have many children**
* But **each child can have only one parent**

This was how **IBM's IMS (Information Management System)** worked — one of the **first production database systems**.

**📦 Real-World Production Example: IBM IMS for NASA**

🛰️ **Use Case**: NASA used it to manage parts of the **Apollo Space Mission**  
🚀 Problem: They needed to track thousands of parts: bolts, engines, electronics, etc.  
📚 Solution: Store parts in a **parent-child hierarchy**

**🧭 How Hierarchical DB was used in Production:**

Imagine you're managing parts for a spaceship:

A diagram of a space craft

AI-generated content may be incorrect.

**🧠 Features of Hierarchical Databases in Production:**

| **Feature** | **Explanation** |
| --- | --- |
| ✅ **Fast Access** | Because the path is pre-defined (like a tree), it's super fast to retrieve child records |
| ❌ **Rigid Structure** | You must define parent-child relationships **in advance**; changing structure is hard |
| ❌ **Redundancy** | If multiple parents need the same child, data gets **duplicated** |
| ✅ **Used in Banking, Telecom** | Early banking systems, telcos, and airlines used this for batch and mainframe processing |
| ✅ **Batch Processing** | Worked well for **sequential data** operations in large systems (no real-time search needed) |

**🧩 How Data Was Stored (Example: Banking System)**

| **Parent Table (Customers)** | **→** | **Child Table (Accounts)** |
| --- | --- | --- |
| Customer\_ID: 101 | → | Account\_ID: 5001 |
| Name: Raj | → | Type: Savings |
|  | → | Balance: ₹50,000 |

 Each customer could have **multiple accounts**

 But an account **could not be shared** across customers (limitation)

**⚙️ Navigation: How Queries Worked**

Instead of **SQL**, they used **navigational commands**:

* GET NEXT
* GET FIRST CHILD
* MOVE TO PARENT

You had to **navigate step-by-step** through the tree — very different from today's flexible SQL.

**💡 Why It Was Revolutionary (Back Then)**

| **Then** | **Now** |
| --- | --- |
| No database systems | IMS was one of the first |
| Files were flat (no relationships) | IMS brought parent-child relations |
| Poor organization | Hierarchical model gave structure to data |

**📚 1. What is a Database (DB)?**

A **Database** is like a **digital cupboard or file system** where you store and organize information.

👉 Example:  
A school database might store:

* Student names
* Their marks
* Which class they belong to

So instead of paper files, you use **tables in software**.

**🧱 2. What is RDBMS (Relational Database Management System)?**

**RDBMS = Relational + Database + Management System**

**✅ It means:**

A software that helps you **store, manage, and access** data in a structured format using **tables** and **relationships**.

🔹 *Relational* = Data is stored in **tables** and connected using **relationships (keys)**  
🔹 *DBMS* = A system to manage, insert, delete, and retrieve data

**Popular RDBMS tools:**

* MySQL
* PostgreSQL
* Microsoft SQL Server
* Oracle Database
* Azure SQL Database
* Database

**🧾 3. What is a Table?**

A **table** is like an **Excel sheet** inside your database.

Each table stores data about **one type of thing**.

**Example: A Students table**

| **student\_id** | **name** | **age** | **city** |
| --- | --- | --- | --- |
| 101 | Raj | 20 | Mumbai |
| 102 | Priya | 22 | Chennai |

**🧍‍♂️ 4. What is a Row?**

A **row** is **one record** in the table — like a person’s full details.

Example:  
(101, Raj, 20, Mumbai) → This is one row → one student.

**🧍‍♂️ 4. What is a Row?**

A **row** is **one record** in the table — like a person’s full details.

Example:  
(101, Raj, 20, Mumbai) → This is one row → one student.

**📊 5. What is a Column?**

A **column** stores a **type of information** across all rows.

Example:

* student\_id column → stores unique IDs
* name column → stores names
* city column → stores cities

**🔑 6. What are Keys?**

**✅ Primary Key**

* A column (or set of columns) that **uniquely identifies each row**
* No two rows can have the same value in the primary key column

👉 Example: student\_id in the Students table is a **Primary Key**  
(You can’t have two students with same ID)

**✅ Foreign Key**

* A column that **links one table to another**.
* It refers to the **primary key** in another table.

👉 Example:

* You have a Courses table and a Enrollments table.
* Enrollments.student\_id is a **Foreign Key** pointing to Students.student\_id

**🔍 Visual Explanation**

**Students Table Enrollments Table**

* ┌─────────────┐ ┌─────────────┐
* │ student\_id │◄──────────────────┤ student\_id │
* │ name │ │ course\_name │
* └─────────────┘ └─────────────┘
* This arrow shows: Enrollments.student\_id → points to → Students.student\_id

🔧 Internally: How Does a Foreign Key Work?

✅ When you define a foreign key, the database engine does 3 things:

**1. Creates a Relationship (like a rule)**

When you write something like: FOREIGN KEY (student\_id) REFERENCES Students(student\_id).

➡ The database **creates a link** (like a contract) between the two tables:

* It **knows that** Enrollments.student\_id must match a value in Students.student\_id

**2. Adds a Constraint Behind the Scenes**

The foreign key becomes a **constraint** in the child table.

It does this:

* ✔️ **Allows only values** in Enrollments.student\_id **that already exist** in Students.student\_id
* ❌ **Rejects invalid inserts** automatically

💣 Example — What Happens When You Try to Insert Invalid Data

-- Let's say this student\_id = 999 doesn't exist in Students table

INSERT INTO Enrollments (course\_name, student\_id)

VALUES ('Biology', 999);

🏗️ DATABASE DESIGN – Step-by-Step Breakdown

1️⃣ **ER Diagrams (Entity Relationship Diagrams)**

**📌 ✅ What Is an ER Diagram?**

**An ER (Entity-Relationship) Diagram is a blueprint of how your database is structured before tables are even created.**

**ER Diagram shows a high-level design of tables (entities), their columns (attributes), and the relationships between them, before building actual tables**

An **Entity-Relationship Diagram (ERD)** is a **visual representation** of:

* 📦 **Entities** (tables)
* 🔗 **Relationships** (how tables are connected)
* 🔑 **Attributes** (columns)
* 💎 **Keys** (primary and foreign keys)

**🧠 Real-World Example — Student Enrolling in Courses**

**🎓 Entities:**

1. **Student**
2. **Course**
3. **Enrollment** (a relationship table between students and courses)

**🧱 ER Diagram Structure:**

Below is how the data flows logically:

A diagram of a program

AI-generated content may be incorrect.

**🔗 How relationships are actually connected in the database engine internally.**

**🔧 ER Diagram → Table Structure (Correct Flow)**

| ER Concept | Real Table Structure Equivalent |
| --- | --- |
| Entity | A Table |
| Attribute | A Column in a table |
| Primary Key | Column that uniquely identifies rows |
| Relationship | Foreign Key or link to another table |
| Cardinality | One-to-One, One-to-Many, etc. (relationship rules) |
| **Are Relationship Rules (Cardinality) Defined in SQL?**  ➡️ **Short Answer**: **No**, SQL does **not explicitly define cardinality like "one-to-many" or "one-to-one" directly** in your table schema — but it's **implied** by how you design tables and keys (primary key + foreign key).  **✅ Cardinality: What It Means**  Cardinality describes **how many records in one table relate to records in another**:   | **Type** | **Meaning** | | --- | --- | | One-to-One | One record in Table A ↔ one record in Table B | | One-to-Many | One record in Table A ↔ many in Table B | | Many-to-Many | Many in Table A ↔ many in Table B (requires a link table) | | 🟩 You **don’t write the words** "One-to-Many" or "Cardinality" in SQL 🟩 But you **enforce those rules** by how you define:   * **Primary Keys** * **Foreign Keys** * **UNIQUE constraints**   So yes — **you are defining relationship rules**, just **indirectly** through table structure.  ✅ **Based on the keys (Primary Key, Foreign Key, and Unique constraints), we define the relationship rules (Cardinality) between different tables.**  **🧾 My Case:**  **✅ Table A:**   * Has **1 Primary Key** (e.g. id) * Has **multiple UNIQUE columns** (e.g. email, username)   **✅ Table B:**   * Has **1 Foreign Key** referencing **Table A’s Primary Key** * Has **1 UNIQUE column** of its own (e.g. passport\_number)   **🔗 Relationship Type (Cardinality)**  **The Foreign Key in Table B references the Primary Key in Table A.**  This means:   * Each row in **Table B** is **linked to exactly one row** in **Table A** (since it references the PK) * One row in **Table A** can be linked to **many rows** in **Table B**   ✅ **Therefore, the cardinality is:**  **👉 One-to-Many**  From: Table A (1) ➡ Table B (Many)  **❓ If Table B’s foreign key does *not* reference Table A’s primary key,**  then what happens to **cardinality / relationship rules**?  **1. Is It Still a Relationship?**   * **Technically, yes**, a foreign key *can* reference **any UNIQUE column** (not just the primary key). * So, Table B **can still have a valid foreign key relationship** with Table A — as long as the column it points to in Table A is **UNIQUE or PRIMARY**. |  | |  |

**🧠 Let’s Take an Example:**

**🧾 Table A: Users**

| **id (PK)** | **email (UNIQUE)** |
| --- | --- |
| **1** | **user1@mail.com** |
| **2** | **user2@mail.com** |

**🧾 Table B: Feedback**

| **feedback\_id (PK)** | **email (FK → Users.email)** | **comment** |
| --- | --- | --- |
| **101** | **user1@mail.com** | **...** |
| **102** | **user1@mail.com** | **...** |

**Here:**

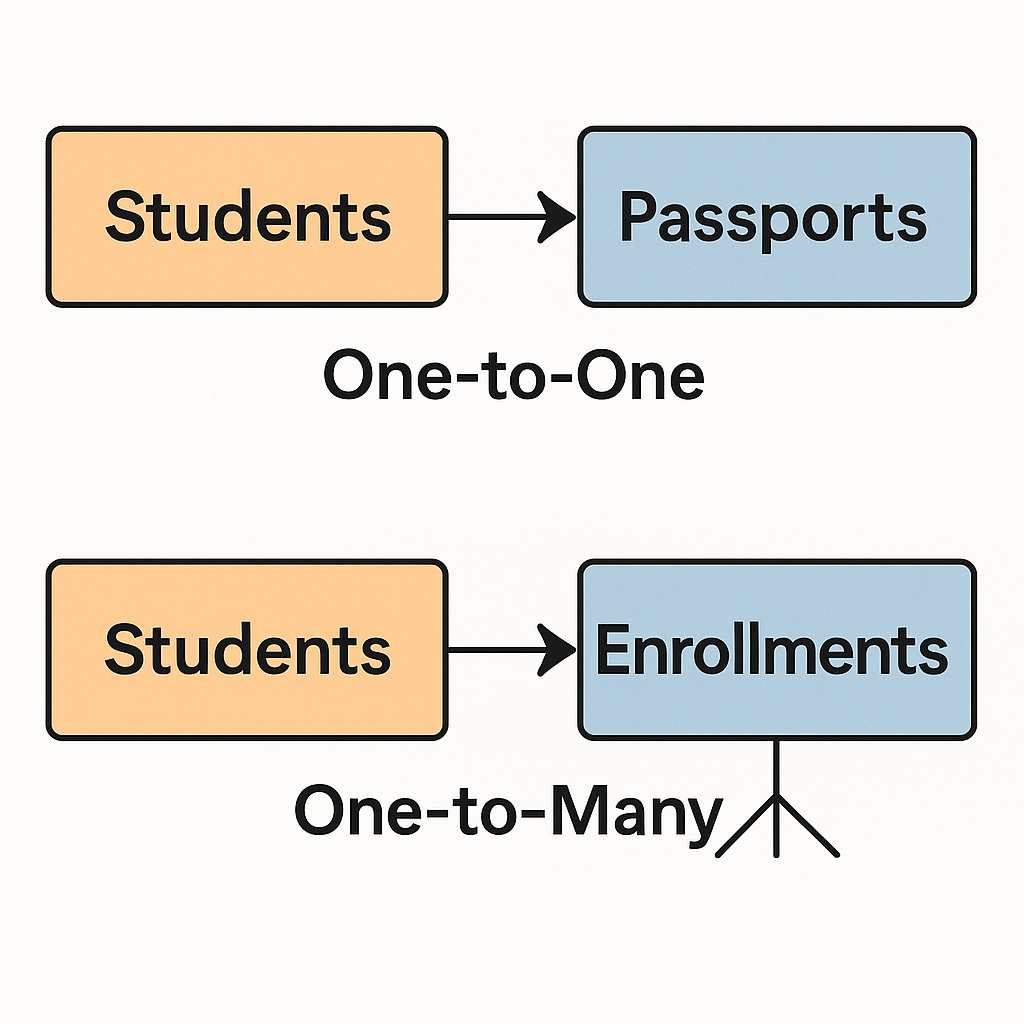
* **Foreign key is referencing Users.email, not the id.**
* **But Users.email is unique, so it's still valid.**

**🔗 So What Is the Cardinality Here?**

**It still behaves like a One-to-Many relationship:**

* **One user (unique email) → many feedback entries.**

**✅ Still: One-to-Many**



That is exactly a One-to-Many relationship.

🔍 Let’s confirm your thinking:

* Table A: Students  
  Each student has one record (with student\_id, name, etc.)
* Table B: Admissions  
  Each admission references a student via student\_id → and that student\_id appears multiple times (for different courses).

📊 So:

* One student → Many admissions  
  ✅ That’s One-to-Many

🧠 Why student\_id is not unique in Admissions?

Because:

One student can have many admission records, so student\_id appears repeatedly in the Admissions table.

Only the Primary Key in Admissions (like admission\_id) will be unique.

**🛠️ Internally, the database enforces:**

1. ✅ **Integrity Rule**:  
   It checks that every student\_id in Enrollments **must exist** in the Students table.

E.g., If someone tries to insert student\_id = 999 in Enrollments, it will fail if that ID isn’t in Students.

1. 🧮 **Index Lookups**:  
   The DB creates an **internal index** on student\_id in the Students table so it can **quickly verify** if the foreign key exists.
2. 🔗 **Pointer Mapping (Logical View)**:  
   Behind the scenes, the relationship looks like a **logical link** — imagine this:
3. Enrollments.student\_id ➡ Students.student\_id
4. (via pointer/index)
5. The DB engine doesn’t **duplicate data** but rather **links** using internal pointers or constraints.

📛 What if you break the relationship?

DELETE FROM Students WHERE student\_id = 101;

If the student is still in Enrollments, this will:

* ❌ **Throw an error** — unless ON DELETE CASCADE is set.
* 💡 **CASCADE tells DB**: if student is deleted, delete their enrollments too.

**🔄 What is ON DELETE CASCADE?**

It's a **rule you apply on a foreign key** that tells the database:

“If a record in the parent table is deleted, **automatically delete** the matching records in the child table too.”

**🧨 What is CASCADE in SQL?**

When you use **CASCADE**, you're telling the database:

**“If a parent row is deleted or updated, automatically apply the same change to all child rows that depend on it.”**

**🔄 Types of CASCADE:**

| **Type** | **Meaning** |
| --- | --- |
| ON DELETE CASCADE | If the parent row is deleted → delete all related child rows |
| ON UPDATE CASCADE | If the parent key is updated → update the foreign keys in child table |

**🧠 Real-Life Analogy:**

Imagine:

* 🧑 A **student** is admitted (parent).
* 📝 He has **enrollment records** (child).

Now if the student **leaves the college** (you delete the student):

* Do you want to keep their old course enrollments?
* Probably not: ON DELETE CASCADE

Which means:

If the **student is deleted**, their **enrollments will also be deleted automatically**.

-----------------------------------------------------------------------------------------------------------------------------------------------------

**🔍 Without CASCADE:**

If you didn’t use ON DELETE CASCADE, the database would **give an error**:

❌ Cannot delete or update a parent row: a foreign key constraint fails

This protects **data integrity**.

**❓ Is CASCADE enabled by default?**

**🟥 No — CASCADE is *not enabled by default*.**

You must **explicitly define it** when creating the foreign key constraint.

**🚫 Default behavior (if CASCADE not defined):**

* If you try to delete a parent row that has dependent child rows, the database will **throw an error**.
* Why? Because it’s protecting **referential integrity**.

**✅ To enable CASCADE:**

You must define it **manually** like this:

**FOREIGN KEY (student\_id)**

**REFERENCES students(student\_id)**

**ON DELETE CASCADE**

**ON UPDATE CASCADE.**

**✅ When to Define the Foreign Key & Tables**

**🔸 If you're creating both tables from scratch:**

You **must create the parent table first** (the one with the **Primary Key**) and then the child table with the **Foreign Key**.

✅ Example: Correct Order

-- Step 1: Create the parent table first

CREATE TABLE students (

student\_id INT PRIMARY KEY,

name VARCHAR(100)

);

-- Step 2: Create the child table with foreign key referencing students

CREATE TABLE enrollments (

enrollment\_id INT PRIMARY KEY,

student\_id INT,

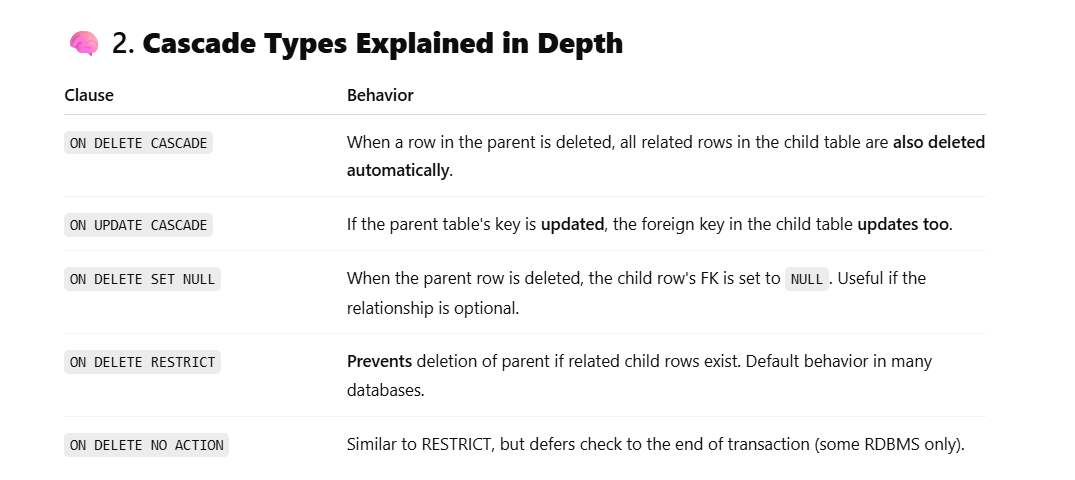
course\_name VARCHAR(100),

FOREIGN KEY (student\_id)

REFERENCES students(student\_id)

**ON DELETE CASCADE**

);



**🔒 What is ON DELETE RESTRICT?**

It means:

❌ You **cannot delete** a row from the **parent table** if **related rows** exist in the **child table**.

It **protects your data** — so you don't break the connection between parent and child tables.

**✅ What is ON DELETE SET NULL?**

It means:

If a row in the **parent table** is deleted, the **foreign key column** in the **child table** is automatically set to NULL.

So, the child record stays, but its link to the parent is removed safely by turning the foreign key to NULL.

**✅ Default Index = Primary Key Index**

When you create a table with a **Primary Key**, the database **automatically creates an index** on that primary key column.

👉 Here, student\_id is the **primary key**.

🧠 Internally, the database will **automatically** create:

CREATE UNIQUE INDEX idx\_students\_student\_id ON Students(student\_id);

You don’t need to write this yourself — it happens behind the scenes!

A diagram of a table

AI-generated content may be incorrect.

**✅ What is a Data Type in a Database?**

**Data Type** defines **what kind of data** can be stored in a column.

Just like in real life you don’t store a name inside a calculator 📟, a **database** needs to know:

* Is this value a number? 🧮
* A date? 📅
* A word or sentence? ✍️

**🧠 Why Do We Need Data Types?**

* To save **space efficiently**
* To avoid **wrong data entry** (e.g., you can’t put "Paresh" in an age column)
* To let the database **optimize** searches and sorting

**✅ Explanation:**

* student\_id: Whole number like 1, 2, 3
* name: Up to 50 letters (like "Paresh")
* date\_of\_birth: e.g., '2000-01-01'
* height: Allows decimal (e.g., 165.4 cm)
* is\_active: TRUE if currently enrolled

**🧩 Common SQL Data Types (Simple Examples):**

| **Data Type** | **Description** | **Example** |
| --- | --- | --- |
| INT | Integer numbers (no decimals) | 25, 2025 |
| VARCHAR(n) | Text (n = max letters) | 'Paresh', 'ODISHA' |
| DATE | Calendar date | '2025-07-18' |
| DECIMAL(5,2) | Numbers with decimals | 99.99, 123.45 |
| BOOLEAN | True/False | TRUE, FALSE |
| TEXT | Long form text (paragraphs) | 'This is a story…' |
| CHAR(n) | Fixed-length string (n letters) | 'Y', 'M', '1234' |
| TIME | Time only | '14:30:00' |
| DATETIME | Full date and time | '2025-07-18 14:30' |

**✅ What Are Constraints in SQL?**

Constraints are used to **enforce rules** at the column or table level.  
They **protect your data** from mistakes or invalid entries.

**✅ SQL Constraints – Full List**

| **Constraint** | **Description** |
| --- | --- |
| PRIMARY KEY | -Uniquely identifies each row in a table. Combines NOT NULL + UNIQUE. |
| FOREIGN KEY | -Links rows in one table to rows in another. Ensures **referential integrity**. |
| NOT NULL | -Column **must have a value**; no missing (null) values allowed. |
| UNIQUE | -All values in the column must be **different**. No duplicates. |
| CHECK | -Ensures the value meets a specific condition (like age > 18). |
| DEFAULT | -Sets a **default value** for a column when no value is provided. |
| INDEX | -Not a constraint exactly, but helps in **fast searching** of data. |
| AUTO\_INCREMENT | -Automatically generates sequential numbers (e.g. IDs). |
| ON DELETE CASCADE | -If a parent row is deleted, child rows are also deleted. |
| ON DELETE SET NULL | -When parent is deleted, child column is set to NULL. |
| ON DELETE RESTRICT | -Prevents deletion of a parent if child rows still exist. |
| ON UPDATE CASCADE | -If primary key is updated, foreign keys auto-update too. |
| ENUM (MySQL) | -Restricts value to a set of predefined strings. |
| CHECK CONSTRAINT (Advanced) | -Applies to table-level or multiple columns at once. |