Data Types

🔹 Quick introduction

أنواع البيانات (**Data Types**) بتحدد شكل وقيمة البيانات اللي ممكن نخزنها في عمود داخل جدول. اختيار النوع الصح مهم للأداء، الدقة، وحجم التخزين.

الشرح التالي يركّز على الأنواع الشائعة في أنظمة **RDBMS** خاصةً **PostgreSQL / MySQL / SQL Server**، مع أمثلة SQL واقعية.

✅ INT (Integer)

- **الوصف:** عدد صحيح بدون كسور.
- **الأنواع الفرعية (common variants):**

- `SMALLINT` (2 bytes)
- `INT` / `INTEGER` (4 bytes)
- `BIGINT` (8 bytes)

- **متى تستخدمه:** للأرقام الصحيحة مثل IDs، Age، counts.
- **نقطة أداء:** اختيار نوع أصغر (SMALLINT) يوفر مساحة لكن حدود القيم أقل.

مثال:

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY, -- PostgreSQL auto-increment
  age INT
);
```

ملاحظة: في PostgreSQL يمكنك استخدام `SERIAL` أو الصيغة الحديثة `GENERATED AS IDENTITY` لإنشاء أعداد تلقائية.

✅ CHAR(n), VARCHAR(n)

CHAR(n)

- **الوصف:** سلسلة بطول ثابت `n` . لو النص أقصر، النظام يملأ بال (space-padding).
- **متى تستخدمه:** للحروف الثابتة الطول مثل رموز الدول `CountryCode CHAR(3)` .

VARCHAR(n)

- **الوصف:** سلسلة بطول متغير حتى `n`.
- **متى تستخدمه:** للأسماء، عناوين البريد، نصوص قصيرة متغيرة الطول.

ملاحظات مقارنة:

- ال `CHAR` أسرع قليلًا في بعض الحالات لأنها ثابتة الطول، لكن `VARCHAR` أكثر مرونة.
- في PostgreSQL، `TEXT` له سلوك مشابه لـ `VARCHAR` بدون حد.

مثال:

```
CREATE TABLE employees (
  code CHAR(5),
  name VARCHAR(100)
);

INSERT INTO employees (code, name) VALUES ('A123', 'Mohamed');
```

✅ TEXT

- **الوصف:** نص طويل غير محدد الطول (practically unlimited).
- **متى تستخدمه:** للمقالات، الأوصاف الطويلة، الملاحظات.
- **ملاحظة أداء:** النصوص الطويلة قد تكون أثقل في عمليات الـ indexing؛ لو هتعمل بحث نصي استخدم full-text indexes أو محركات بحث خارجية.

مثال:

```
CREATE TABLE articles (
  id SERIAL PRIMARY KEY,
  content TEXT
);
```

✅ DATE

- **الوصف:** يخزن التاريخ فقط (YYYY-MM-DD عادة).
- **متى تستخدمه:** تاريخ الميلاد، تاريخ حدث، تاريخ نشر.

مثال:

```sql
CREATE TABLE events (
  id SERIAL PRIMARY KEY,
  event_date DATE
);

INSERT INTO events (event_date) VALUES ('2025-08-22');
```

✅ DATETIME / TIMESTAMP (تاريخ + وقت)

- **الوصف:** يخزن التاريخ والوقت (`YYYY-MM-DD HH:MM:SS`).
- **الفرق المهم (PostgreSQL):**
  - ال `TIMESTAMP WITHOUT TIME ZONE` (توقيت محلي، لا يهتم بالـ timezone)
  - ال `TIMESTAMP WITH TIME ZONE` أو `timestamptz` (يحفظ لحظة زمنية عالمية)
- **متى تستخدم كل واحد:**
  - لو تطبيقك عالمي ويهمك توقيت موحّد، استخدم `timestamptz` .
  - لو بياناتك محلية وغير مرتبطة بالـ timezone، يكفي `timestamp` أو `datetime` .

مثال:

```sql
CREATE TABLE logins (
  id SERIAL PRIMARY KEY,
  user_id INT,
  logged_at TIMESTAMPTZ DEFAULT now()
);
```

✅ BOOLEAN

- **الوصف:** قيمة منطقية `TRUE` / `FALSE` (وأحيانًا `NULL`).
- **متى تستخدمه:** Flags مثل `is_admin` , `is_active` .

مثال:

```sql
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  is_active BOOLEAN DEFAULT TRUE
);
```

✅ FLOAT / REAL / DOUBLE PRECISION

- **الوصف:** أرقام عشرية تقريبية (floating-point). لها أخطاء دقيقة في التمثيل الثنائي.
- **الأنواع:**
- `REAL` / `FLOAT4` (32-bit)
- `DOUBLE PRECISION` / `FLOAT8` (64-bit)
- **متى تستخدمه:** للقياسات العلمية، السرعات، نسب تقريبية.
- **تحذير:** لا تستخدمها للأموال أو الحسابات التي تحتاج دقة عشرية مطلقة.

مثال:

```
CREATE TABLE measurements (
  id SERIAL PRIMARY KEY,
  temp DOUBLE PRECISION
);

INSERT INTO measurements (temp) VALUES (36.6);
```

✅ DECIMAL / NUMERIC(p, s)

- **الوصف:** رقم عشري دقيق ذو دقة محددة.
- `p` = precision (إجمالي عدد الأرقام)
- `s` = scale (عدد الأرقام بعد الفاصلة)
- **متى تستخدمه:** الحسابات المالية، الأسعار، الدرجات (مثل GPA) لأنك تحتاج دقة وعدم تقريب.

مثال:

```
CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  price DECIMAL(10,2) -- 12345678.90 مثال
);
```

🔹 أمثلة عملية مركّبة (Putting it all together)

```
CREATE TABLE Students (
  StudentID INT PRIMARY KEY,
  Name VARCHAR(50) NOT NULL,
  Age SMALLINT,
```

```
  GPA DECIMAL(3,2) -- دقة مناسبة لـ GPA (مثلاً 4.00)
);

CREATE TABLE Courses (
  CourseID INT PRIMARY KEY,
  Title VARCHAR(100)
);

CREATE TABLE StudentCourses (
  StudentID INT,
  CourseID INT,
  CourseGPA DECIMAL(3,2),
  PRIMARY KEY (StudentID, CourseID),
  FOREIGN KEY (StudentID) REFERENCES Students(StudentID) ON DELETE
CASCADE,
  FOREIGN KEY (CourseID) REFERENCES Courses(CourseID) ON DELETE CASCADE
);
```

🔹 نصائح عملية وBest Practices

- **اختيار الحجم المناسب:** استخدم `SMALLINT/INT/BIGINT` بحسب نطاق الأرقام لتوفير مساحة.
- **النصوص:** استخدم `VARCHAR(n)` للنصوص القصيرة، و `TEXT` للمحتوى الطويل. في PostgreSQL ال `TEXT` و `VARCHAR` متقاربين لكن `TEXT` أبسط.
- **الأموال:** استخدم `DECIMAL/NUMERIC` بدلاً من `FLOAT` لتفادي أخطاء التقريب.
- **الأوقات:** لو تطبيق موز ع/عالمي استخدم `TIMESTAMPTZ` لتخزين أوقات موحدة (Postgres).
- **المفاتيح التلقائية:** استخدم `SERIAL` أو `GENERATED AS IDENTITY` في PostgreSQL لسهولة توليد الـ IDs.
- **التحويلات (Casting):** التحويل بين الأنواع متاح بـ `type::` في PostgreSQL أو `CAST(... AS type)` في SQL القياسي.
  - مثال: `SELECT '123'::int;`
- **الفهرسة (Indexing):** فهرس الأعمدة المستخدمة في WHERE/JOIN لتحسين الأداء. فهرس النصوص الكبيرة قد يتطلب full-text index.
- **التحقق (CHECK):** استخدم `CHECK` للتحقق من الحدود (مثلاً `CHECK (gpa BETWEEN 0 AND 4)`).

مثال CHECK:

```
ALTER TABLE Students ADD CONSTRAINT chk_gpa_range CHECK (GPA >= 0 AND GPA
<= 4.0);
```

🔹 أخطاء شائعة تجنبها

- استخدام `FLOAT` للحسابات المالية → قد يؤدي لأخطاء تقريب.
- اختيار `VARCHAR` بقيمة صغيرة جدًا → يسبب قص الأحرف.
- تجاهل timezone عند تخزين توقيت الأحداث في أنظمة موزعة.
- عدم استخدام فهارس للأعمدة التي تظهر في شروط البحث (`WHERE`) و `JOIN` .