Lecture 6. Constraints

What Are Constraints?

> **Constraints** = rules applied to table columns that control what data can be stored.

- Enforce **data integrity and accuracy**
- Prevent **invalid or inconsistent data**
- If a value violates a constraint → PostgreSQL **raises an error** and the row is not inserted/updated.

Main types:

- `CHECK`
- `DEFAULT`
- `NOT NULL`
- `UNIQUE`
- `PRIMARY KEY` (and **composite** primary key)
- `FOREIGN KEY`

CHECK Constraint

Idea

> `CHECK` ensures each row satisfies a **boolean condition**.

- Used for:
  - numeric ranges
  - string patterns
  - logical relationships between columns
- If condition is `FALSE` → row is rejected.
- If condition is `TRUE` or `UNKNOWN` (because of NULL) → row is allowed.

Column-level syntax:

```
column_name data_type CHECK (condition)
```

Table-level syntax:

```
CHECK (condition_using_one_or_more_columns)
```

Examples

Numeric range:

```
age INT CHECK (age >= 0 AND age <= 120)
```

Multiple-column constraint:

```
CHECK (start_date <= end_date)
```

String pattern (email-like pattern):

Slides describe an email regex roughly like:

- `^[A-Za-z0-9._%+-]+` – allowed characters before `@`
- `@` – separator
- `[A-Za-z0-9.-]+` – domain
- `[A-Za-z]{2,}$` – top-level domain with at least 2 letters

Example:

```
email TEXT CHECK (
   email ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'
)
```

Here `~` is the PostgreSQL regex match operator.

Functions in CHECK

You can use functions in constraints, e.g.:

```
CHECK (lower(status) IN ('new', 'processing', 'done'))
```

or

```
CHECK (char_length(password) >= 8)
```

Adding / Dropping CHECK on Existing Table

Add:

```
ALTER TABLE users
ADD CONSTRAINT age_positive CHECK (age >= 0);
```

Drop:

```
ALTER TABLE users
DROP CONSTRAINT age_positive;
```

Multiple Constraints

A column or table can have **several constraints** at once:

```
salary NUMERIC
    CHECK (salary >= 0)
    CHECK (salary <= 1000000)
    NOT NULL;
```

All of them must be satisfied.

DEFAULT Constraint

Idea

`DEFAULT` specifies what value a column gets when **no value is provided** in `INSERT`.

- If value is omitted:

    - DB uses **default value**
- If no `DEFAULT`:
    - `NULL` (if column allows NULL)
    - error (if `NOT NULL` and no value given)

Slides emphasize: default can be a **constant, expression, or function**, depending on type.

Examples

Constant default:

```
status TEXT DEFAULT 'new'
```

Expression default:

```
discount NUMERIC(5,2) DEFAULT 0.0
```

Function default:

```
created_at TIMESTAMPTZ DEFAULT now()
```

Add / Drop DEFAULT

Add / change:

```
ALTER TABLE orders
ALTER COLUMN status SET DEFAULT 'pending';
```

Drop:

```
ALTER TABLE orders
ALTER COLUMN status DROP DEFAULT;
```

NULL and NOT NULL

Concept

- `NULL` = **no value / unknown**
  - Not the same as `0` or empty string.
- By default, PostgreSQL columns **allow NULL** unless you specify `NOT NULL`

NOT NULL

`NOT NULL` means the column **must always have a value**:

```
name TEXT NOT NULL
```

If you try to insert a row with `name = NULL`, you get an error.

Altering NOT NULL

Add NOT NULL:

```
ALTER TABLE users
ALTER COLUMN email SET NOT NULL;
```

(Only works if all existing rows already have non-NULL email.)

Drop NOT NULL:

```
ALTER TABLE users
ALTER COLUMN email DROP NOT NULL;
```

UNIQUE Constraint

Idea:

`UNIQUE` ensures that all non-NULL values in a column (or group) are **distinct**.

- Used on:
  - single column
  - multiple columns (composite unique key)

Basic syntax:

```
email TEXT UNIQUE
```

or table-level:

```
CONSTRAINT unique_email UNIQUE (email)
```

UNIQUE vs PRIMARY KEY

- **UNIQUE**:
  - allows **multiple NULLs** (each NULL is treated as distinct)
  - a table can have **many UNIQUE** constraints
- **PRIMARY KEY**:
  - implies `UNIQUE` **and** `NOT NULL`
  - a table can have **only one** primary key

Add UNIQUE to Existing Table

```
ALTER TABLE users
ADD CONSTRAINT users_email_key UNIQUE (email);
```

Composite UNIQUE

```
ALTER TABLE enrollments
ADD CONSTRAINT unique_student_course UNIQUE (student_id, course_i
d);
```

PRIMARY KEY & Composite Primary Key

PRIMARY KEY

A `PRIMARY KEY`:

- uniquely identifies each row

- is both:
  - `UNIQUE`
  - `NOT NULL`
- there is **only one** primary key per table (but it can be made of several columns).

Column-level:

```
id SERIAL PRIMARY KEY
```

Table-level:

```
CONSTRAINT orders_pkey PRIMARY KEY (order_id)
```

Composite Primary Key

A Composite Primary Key spans **more than one column**.
Lecture6

- Uniqueness is enforced on the **combination** of values.
- No two rows can have the same combination of those columns.

Example:

```
CREATE TABLE enrollment (
    student_id INT,
    course_id  INT,
    enrolled_at DATE,
    CONSTRAINT enrollment_pkey PRIMARY KEY (student_id, course_id)
);
```

Use when:

- no single column uniquely identifies rows
- but combination does (like `(student_id, course_id)`).

FOREIGN KEY

Idea

`FOREIGN KEY` creates a **relationship** between two tables by referencing the **primary key** (or unique key) of another table.

- Enforces **referential integrity**:
  - you cannot insert a row in the child table if the referenced row does not exist in the parent table.
- Links tables logically (parent–child).

Basic Syntax

```
-- parent table
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    name        TEXT NOT NULL
);

-- child table
CREATE TABLE orders (
    order_id    SERIAL PRIMARY KEY,
    customer_id INT,
    amount      NUMERIC(10,2),
    CONSTRAINT orders_customer_fk
        FOREIGN KEY (customer_id)
        REFERENCES customers (customer_id)
);
```

Now every `orders.customer_id` must refer to an existing `customers.customer_id` (or be NULL if allowed).

ON DELETE and ON UPDATE Actions

ON DELETE CASCADE

If a parent row is deleted → **child rows are automatically deleted**.

```
FOREIGN KEY (customer_id)
REFERENCES customers (customer_id)
ON DELETE CASCADE
```

Use when child rows have no meaning without parent.

ON UPDATE CASCADE

If the parent key changes → the **foreign key in child rows is updated** automatically.

```
ON UPDATE CASCADE
```

Use when parent key values can change (less common in practice).

ON DELETE SET NULL

On parent delete → child's foreign key becomes `NULL`:

```
ON DELETE SET NULL
```

Use when child row can exist without parent, but we still want to keep it, just unlink.

ON DELETE RESTRICT

Prevents deletion of a parent row if it is still referenced:

```
ON DELETE RESTRICT
```

- Database will **reject** deletion of parent if any child rows exist.
- Good for protecting important data.

Multiple Constraints on a Column / Table

PostgreSQL allows multiple constraints on same column or table.

Example combining everything:

```
CREATE TABLE users (
    id         SERIAL PRIMARY KEY,
    email      TEXT NOT NULL
               UNIQUE
               CHECK (email ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-
z]{2,}$'),
    created_at TIMESTAMPTZ NOT NULL DEFAULT now(),
    age        INT CHECK (age >= 0)
);
```

Here on one table we have:

- `PRIMARY KEY`
- `NOT NULL`
- `UNIQUE`
- `DEFAULT`
- multiple `CHECK`s

All constraints must be satisfied for each row.