

# PostgreSQL

---

# Langage SQL

# Introduction

Ce module est centré sur le **langage SQL** (Structured Query Language), un **langage de requêtes** conçu pour :

- **consulter**,
- **insérer**,
- **modifier**,
- et **supprimer** des données stockées en base de données.

Son rôle est donc **d'interagir** avec les **bases de données**, et plus particulièrement les **bases de données relationnelles**.

# Structure des bases de données relationnelles

# Les bases de données relationnelles

Développées dès le début des années 70, les **bases de données relationnelles** (ou *relational databases*) sont l'un des modèles de base de données les plus utilisés.

Les BDDR sont fondées sur la **théorie relationnelle** développée par **Edgar F. Codd**, et s'appuient sur l'usage de **tables** (relations) composées de **lignes** (tuples) et de **colonnes** (attributs).

Cela signifie qu'on va essayer de regrouper nos données à travers des caractéristiques ou des étiquettes **communes**.

# Modéliser les données de manière logique

La **modélisation des données** consiste à concevoir :

- la **structure** des données qui seront stockées dans la base,
- ainsi que les **relations** entre ces données.

L'objectif est de répondre aux **besoins fonctionnels** d'une application, tout en garantissant la **cohérence** et **l'intégrité** des informations.

Pour cela, on définit des **contraintes d'intégrité** : ce sont des **règles** que les données doivent respecter lors de leur création, modification ou suppression.

La modélisation permet donc de **structurer les données** de manière **logique et cohérente**, afin d'assurer la fiabilité du système d'information.

Elle se déroule en plusieurs étapes clés, notamment :

- l'identification des **entités** (objets ou concepts à représenter) et de leurs **attributs**,
- la définition des **relations et associations** entre ces entités,
- la mise en place de **contraintes** pour encadrer le comportement des données.

# La structure

**Quels sont les concepts présents dans vos données ?**

Formulez les concepts principaux, et leurs caractéristiques

Exemple : un individu

prenom : Jean

nom : Dupont

métier : garagiste

age : 25

➡ Ces concepts seront représentés par des tables



## Les données (data)

Je m'appelle Jean Dupont, je suis garagiste et j'ai 25ans

Je m'appelle Marie Dupuis, je suis chauffagiste et j'ai 33ans

Je m'appelle Eric François, je suis contrôleur et j'ai 51ans

## Les données (data) + relations

Je m'appelle Jean Dupont, je suis garagiste et j'ai 25ans

Individu 1

Je m'appelle Marie Dupuis, je suis chauffagiste et j'ai 33ans

Individu 2

Je m'appelle Eric François, je suis contrôleur et j'ai 51ans

Individu 3

## Les données (data) + relations

Je m'appelle Jean Dupont, je suis garagiste et j'ai 25ans

Je m'appelle Marie Dupuis, je suis chauffagiste et j'ai 33ans

Je m'appelle Eric François, je suis contrôleur et j'ai 51ans

Prénom

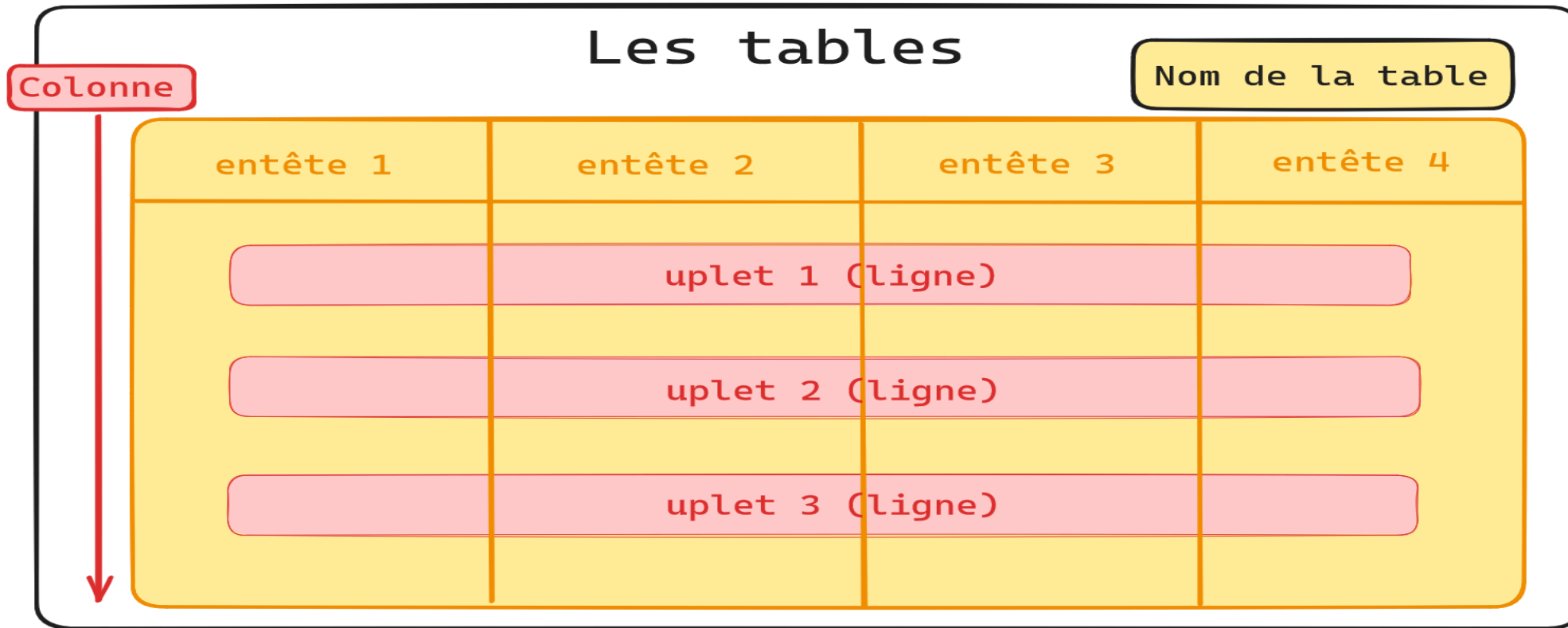
Nom

Métier

Age

# Qu'est-ce qu'une table ?

Une **table**, une **relation**, est un tableau à deux dimensions qui contient des données. Elle est composée de lignes et de colonnes.



# Les lignes

Les lignes d'une table (enregistrements, tuples ou n-uplets) représentent des instances concrètes de données et chaque ligne correspond à un enregistrement **unique** dans la table.

## ➔ Clé primaire (Primary Key, **PK**)

La clé primaire est une **colonne** (ou un **ensemble de colonnes**) qui **identifie de façon unique chaque ligne** dans la table.

- Elle permet de **distinguer** les enregistrements les uns des autres.
- Elle contient généralement un **identifiant unique**, souvent un **nombre entier**, qui ne peut être nulle.

# Les colonnes

Les colonnes d'une table représentent les **attributs** (ou **champs**) d'une entité. Chaque attribut possède :

- un **nom** (ex. : nom, date\_naissance),
- une **valeur** pour chaque enregistrement (ligne),
- un **type** de données (ex. : texte, entier, date...),
- un **domaine**, c'est-à-dire **l'ensemble des valeurs autorisées** pour cet attribut.

DATABASE (Base de données)

Table 1

C1	C2	C3

Table 2

C1	C2	C3

Table 3

C1	C2	C3

Exemple de base de données : Centre de formation

Stagiaires

Nom

Prenom

Age


Formateurs

Nom

Prenom

Age


Formations

Theme

Date

Lieu

--	--	--



## Attributs dérivés

Un **attribut dérivé** est un attribut que l'on peut **calculer à partir d'autres attributs** existants.

Exemples :

- `prix_TTC` peut être déduit de `prix_HT` et `TVA`,
  - `âge` peut être calculé à partir de la `date de naissance` et de la date du jour.
- ➔ Ces attributs ne sont généralement pas stockés dans la base de données, car cela créerait une redondance et un risque d'incohérence si les données sources changent.

# Les associations

Lorsque l'on modélise une base de données, il est essentiel d'identifier les **liens logiques** entre les entités : ce sont les **associations**.

- **One to one** : Chaque entité A est associée à **une seule** entité B, et réciproquement (ex: personne -> passeport).
- **One to many** : Une entité A peut être associée à **plusieurs** entités B, mais B n'est lié **qu'à une seule** entité A (ex: auteur -> livres).
- **Many to many** : Plusieurs entités A peuvent être associées à plusieurs entités B (ex: étudiants - cours).

## ➔ **Clé étrangère** (Foreign Key, **FK**)

La **clé étrangère** est une colonne qui permet de **lier deux tables** entre elles.

- Elle correspond à la **clé primaire** d'une autre table.
- Elle doit **obligatoirement** faire référence à une ligne existante dans une autre table (sinon les données sont incohérentes)

# Les tables d'association

Les tables d'association(ou **tables de jointure**) sont utilisées pour modéliser les relations **many-to-many** entre deux entités.

Elles permettent de lier deux tables en créant une **table intermédiaire** contenant les **clés étrangères** pointant vers les deux tables d'origine. Cette table contient généralement :

- une **clé primaire** (souvent composée des deux clés étrangères),
- les **clés étrangères** des tables liées,
- des **attributs supplémentaires** décrivant la relation (facultatif).

## Exercice 1

J'ai 4 individus avec certains de leurs attributs spécifiés.  
Classez leurs données sous forme de table:



- Julie, femme, 24ans, électricienne, 4ans d'expérience
- Françoise, comptable, 40ans, 18ans d'expérience, femme
- Romain, 30ans, développeur, homme, expérience non spécifiée
- Manuel, 12 ans d'expérience, 39ans, homme, professeur

## Exercice 2

Voici un tas de données et d'**entêtes** isolées, regroupez les avec cohérence, -de façon relationnelle-, sous forme de table:

chat      gros      chien      pigeon      **classe**

**taille**      autruche      petit      **nom**      éléphant

oiseau      moyen      herbivore      cheval      mammifere

baleine      carnivore      être humain      **régime alimentaire**      omnivore

# Le langage SQL

# Le langage SQL

Le langage SQL (*Structured Query Language*) est le langage de **requêtes** qui va nous permettre d'**interagir** avec notre base de données via le système de gestion de base de données.

Conçu au début des années 70 par IBM, il a été normé par l'American National Standards Institute (ANSI) **en 1986** puis mondialement par l'International Organization for Standardization (ISO) **en 1989**.

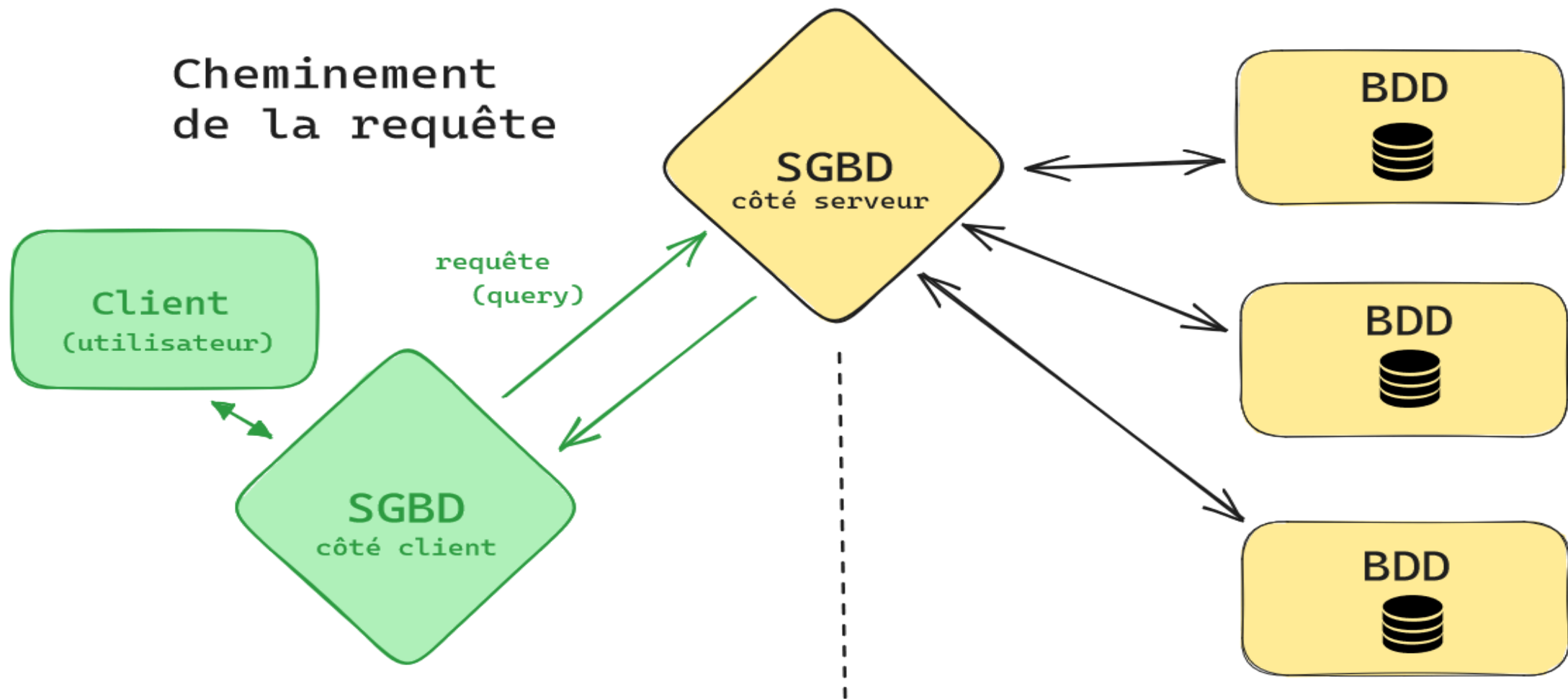
Il reçoit encore fréquemment des mises à jour (sa dernière date de 2023) et est le langage de requêtes le plus utilisé au monde.



Quelques mises à jours majeures:

- **SQL-92** (1992)  
Ajout des requêtes récursives et des déclencheurs
- **SQL:1999** (1999)  
Support des objets et des procédures stockées
- **SQL:2011** (1992)  
Ajout des tables temporaires et des opérations de fusion.
- **SQL:2023** (2023)  
Support des fichiers JSON

# Les requêtes SQL



# Les extensions de langage

Le choix de SGBDR n'est pas anodin pour l'apprentissage du SQL, car de nombreux SGBDR ont leur propre **extension du langage** SQL qui apportent des **fonctionnalités supplémentaires** et peuvent parfois modifier la syntaxe.

Attention donc, certaines syntaxes de requêtes qui sont valables dans un SGBDR auront des syntaxes différentes dans d'autres.

Cependant, la logique derrière ces différentes syntaxes reste la plupart du temps très similaire.

# Exemples d'extensions de langages

- **PL/SQL** (*Procedural Language*)

C'est l'extension créée par Oracle. Elle ajoute des fonctionnalités de programmation telles que les boucles, les fonctions...

- **PL/pgSQL**

C'est l'extension utilisée par le SGBD PostgreSQL. Sa logique est proche du PL/SQL.

- **T-SQL** (*Transact-SQL*)

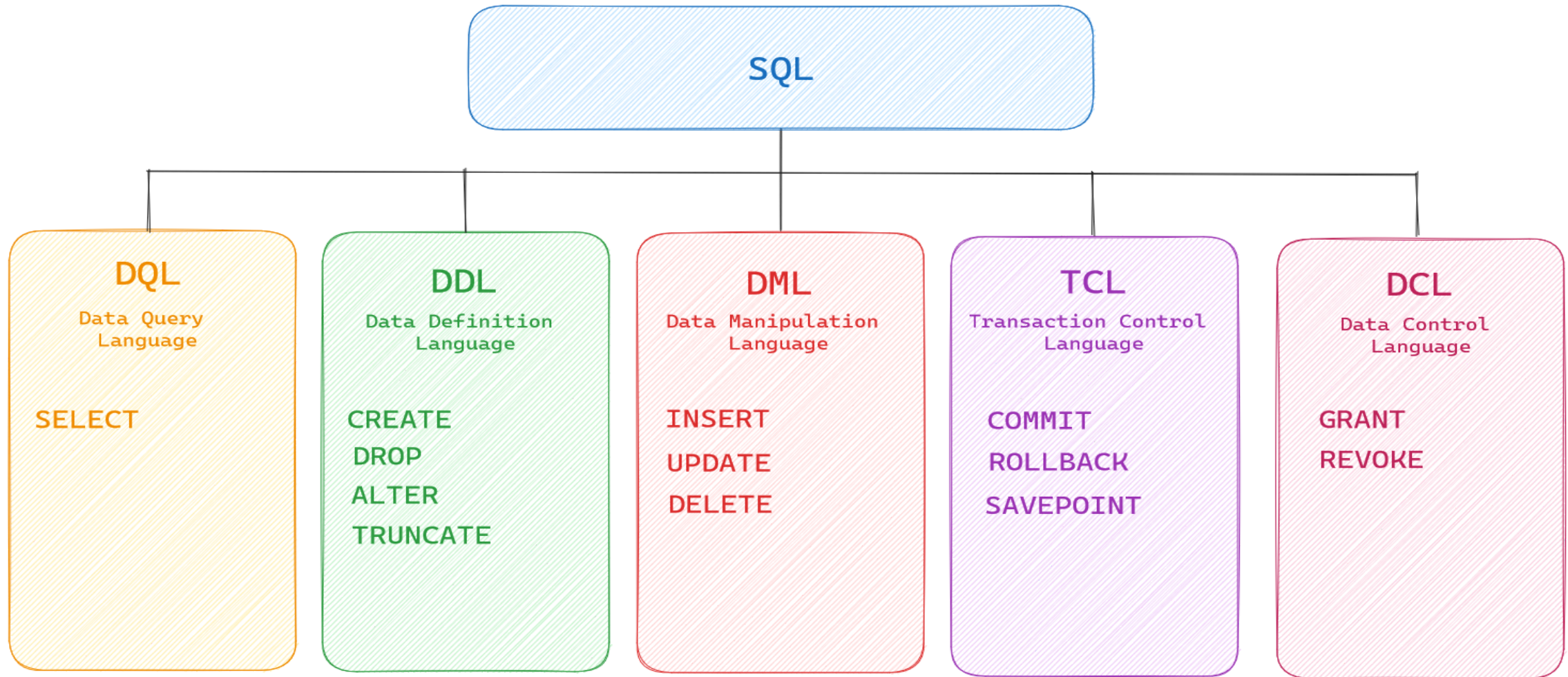
L'extension spécifique de Microsoft SQL Server et Sybase.

# Les sous-langages

Le SQL est un langage très étendu qui s'est vu augmenté de nombreuses fonctionnalités au fil des années. Il permet de faire diverses manipulations avec les bases de données relationnelles.

Par souci de clarté, il est souvent découpé en 5 grandes sous-catégories de **fonctionnalités spécifiques** qui sont appelées les **sous-langages SQL**.

# Les sous-langages





# Définition des sous-langages

- **Data Query Language (DQL):** Le langage de requête de données est utilisé pour interroger une BDD et en lire ses données.
- **Data Manipulation Language (DML):** Le langage de manipulation de données est utilisé pour modifier, ajouter ou supprimer des données au sein d'une base de données.
- **Data Definition Language (DDL):** Le langage de définition de données est utilisé pour définir la structure des objets de la base de données

# Définition des sous-langages

- **Data Control Language (DCL):** Le langage de contrôle de données est utilisé pour gérer les autorisations et les privilèges d'accès à la base de données.
- **Transaction Control Language (TCL):** Le langage de contrôle des transactions est utilisé pour gérer spécifiquement les transactions dans une base de données.



# Introduction à PostgreSQL

# Introduction à PostgreSQL

- PostgreSQL est un système de gestion de base de données relationnelle (SGBDR) open-source.
- Développé par le projet PostgreSQL Global Development Group.
- Connu pour sa robustesse, sa conformité aux standards et sa fiabilité.

# Histoire de PostgreSQL

- Développé à l'Université de Californie à Berkeley dans les années 1980.
- Basé sur le projet Ingres.
- Version initiale publiée en 1989.
- Renommé PostgreSQL en 1996 pour éviter des problèmes de marque déposée.

# Utilisation de PostgreSQL

1. Une base de données robuste dans la pile LAPP (Linux, Apache, PostgreSQL et PHP)
2. Base de données transactionnelle polyvalente
3. PostgreSQL prend en charge les bases de données géospatiales pour les systèmes d'information géographique (SIG)

# DDL (Data Definition Language)

# DDL dans PostgreSQL

- Permet de définir et de modifier la structure des données dans une base de données.
- Principales commandes DDL : CREATE, ALTER, DROP.

## Exemples de commandes DDL :

- `CREATE TABLE` : Crée une nouvelle table dans la base de données.
- `ALTER TABLE` : Modifie la structure d'une table existante.
- `DROP TABLE` : Supprime une table de la base de données.

# CREATE TABLE

La commande `CREATE TABLE` est utilisée pour créer une nouvelle table dans la base de données.

```
CREATE TABLE employees (  
    id SERIAL PRIMARY KEY,  
    name VARCHAR(100),  
    department VARCHAR(50),  
    salary NUMERIC(10, 2)  
);
```

Cette commande crée une table `employees` avec quatre colonnes : `id`, `name`, `department`, et `salary`. La colonne `id` est une clé primaire et est auto-incrémentée grâce à `SERIAL`.

# ALTER TABLE

La commande `ALTER TABLE` est utilisée pour modifier la structure d'une table existante.

```
ALTER TABLE employees  
ADD COLUMN email VARCHAR(100);
```

Cette commande ajoute une nouvelle colonne `email` à la table `employees`.



## DROP TABLE

La commande `DROP TABLE` est utilisée pour supprimer une table de la base de données.

```
DROP TABLE employees;
```

Cette commande supprime complètement la table `employees` de la base de données.

# CREATE INDEX

La commande `CREATE INDEX` est utilisée pour créer un index sur une ou plusieurs colonnes d'une table.

```
CREATE INDEX idx_department ON employees (department);
```

Cette commande crée un index nommé `idx_department` sur la colonne `department` de la table `employees`, ce qui accélère les opérations de recherche basées sur cette colonne.

# ALTER TABLE ADD CONSTRAINT

La commande `ALTER TABLE ADD CONSTRAINT` est utilisée pour ajouter une contrainte à une table existante.

```
ALTER TABLE employees  
ADD CONSTRAINT salary_check CHECK (salary >= 0);
```

Cette commande ajoute une contrainte de vérification (`CHECK`) pour s'assurer que la valeur de la colonne `salary` est toujours supérieure ou égale à zéro.

# ALTER TABLE DROP CONSTRAINT

La commande `ALTER TABLE DROP CONSTRAINT` est utilisée pour supprimer une contrainte d'une table existante.

```
ALTER TABLE employees  
DROP CONSTRAINT salary_check;
```

Cette commande supprime la contrainte `salary_check` de la table `employees`.

Ces commandes DDL sont essentielles pour la définition et la modification de la structure des données dans PostgreSQL. Utilisez-les avec précaution, car elles ont un impact direct sur la façon dont les données sont organisées et accessibles dans la base de données.

# Types de Données et Contraintes

# Types de Données Numériques

- `INTEGER` : Nombre entier.
- `BIGINT` : Nombre entier long.
- `NUMERIC(precision, scale)` : Nombre décimal précis.
- `FLOAT` : Nombre en virgule flottante simple précision.
- `DOUBLE PRECISION` : Nombre en virgule flottante double précision.

# Types de Données de Texte

- `CHAR(n)` : Chaîne de caractères fixe de longueur n.
- `VARCHAR(n)` : Chaîne de caractères variable de longueur maximale n.
- `TEXT` : Chaîne de caractères de longueur variable (illimitée).

# Types de Données Temporelles

- **DATE** : Date (année, mois, jour).
- **TIME** : Heure du jour.
- **TIMESTAMP** : Date et heure.
- **INTERVAL** : Intervalle de temps.



# Types de Données Booléennes

- **BOOLEAN** : Vrai ou faux (true/false).

# Types de Données Binaires

- **BYTEA** : Données binaires de longueur variable.

# PRIMARY KEY

Définit une colonne comme clé primaire, assurant son unicité et sa non-nullité.

```
CREATE TABLE students (  
    student_id SERIAL PRIMARY KEY,  
    name VARCHAR(100) NOT NULL,  
    email VARCHAR(100) UNIQUE  
);
```

Dans cet exemple, `student_id` est défini comme clé primaire, garantissant que chaque étudiant a un identifiant unique.

# Colonne identité

La version 10 de PostgreSQL a introduit une nouvelle contrainte `GENERATED AS IDENTITY` qui permet d'attribuer automatiquement un numéro unique à une colonne.

La contrainte `GENERATED AS IDENTITY` est la variante conforme au standard SQL de la bonne vieille colonne `SERIAL`.

```
CREATE TABLE color (  
    color_id INT GENERATED ALWAYS AS IDENTITY,  
    color_name VARCHAR NOT NULL  
);
```

## FOREIGN KEY

Établit une relation entre les données de deux tables en référençant la clé primaire d'une table dans une autre.

```
CREATE TABLE orders (  
    order_id SERIAL PRIMARY KEY,  
    product_id INTEGER REFERENCES products(product_id),  
    quantity INTEGER  
);
```

Dans cet exemple, `product_id` dans la table `orders` est une clé étrangère qui référence la clé primaire `product_id` de la table `products`.

# CHECK

Spécifie une condition qui doit être vraie pour chaque ligne de la table.

```
CREATE TABLE employees (  
    employee_id SERIAL PRIMARY KEY,  
    age INTEGER CHECK (age >= 18),  
    salary NUMERIC CHECK (salary > 0)  
);
```

Dans cet exemple, les contraintes **CHECK** garantissent que l'âge des employés est supérieur ou égal à 18 et que leur salaire est strictement positif.

# UNIQUE

Empêche l'insertion de valeurs en double dans une colonne ou un groupe de colonnes.

```
CREATE TABLE users (  
    username VARCHAR(50) PRIMARY KEY,  
    email VARCHAR(100) UNIQUE,  
    phone_number VARCHAR(15) UNIQUE  
);
```

Dans cet exemple, `email` et `phone_number` sont des contraintes d'unicité, garantissant que chaque adresse e-mail et numéro de téléphone est unique dans la table `users`.

# NOT NULL

Indique qu'une colonne ne peut pas contenir de valeurs NULL.

```
CREATE TABLE books (  
    book_id SERIAL PRIMARY KEY,  
    title VARCHAR(200) NOT NULL,  
    author VARCHAR(100) NOT NULL,  
    publication_year INTEGER  
);
```

Dans cet exemple, `title` et `author` sont des colonnes non-nullables, garantissant que chaque livre a un titre et un auteur spécifiés.



## DEFAULT

La contrainte **DEFAULT** est utilisée pour définir une valeur par défaut pour une colonne lorsque aucune valeur n'est spécifiée lors de l'insertion d'une ligne.

```
CREATE TABLE products (  
    product_id SERIAL PRIMARY KEY,  
    name VARCHAR(100) NOT NULL,  
    price NUMERIC(10, 2) DEFAULT 0.00,  
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);
```

# DELETE CASCADE

La contrainte `DELETE CASCADE` est utilisée pour garantir que lorsque la ligne référencée dans une table par une clé étrangère est supprimée, toutes les lignes qui font référence à cette clé étrangère sont également supprimées.

```
CREATE TABLE orders (  
    order_id SERIAL PRIMARY KEY,  
    customer_id INTEGER REFERENCES customers(customer_id) ON DELETE CASCADE,  
    order_date DATE NOT NULL  
);
```

## EXCLUDE

Permet de spécifier une condition d'exclusion qui ne peut pas être violée par plus d'une ligne dans la table.

```
CREATE TABLE reservations (  
    room_id INTEGER,  
    check_in DATE,  
    check_out DATE,  
    EXCLUDE USING GIST (room_id WITH =, daterange(check_in, check_out) WITH &&)  
);
```

Dans cet exemple, la contrainte d'exclusion garantit qu'aucune réservation ne chevauche une autre pour la même chambre dans le même intervalle de temps.

# DML (Data Manipulation Language)

# DML dans PostgreSQL

- Utilisé pour manipuler les données stockées dans la base de données.
- `INSERT INTO` : Insère une nouvelle ligne dans une table.
- `UPDATE` : Modifie les valeurs d'une ou plusieurs lignes dans une table.
- `DELETE FROM` : Supprime une ou plusieurs lignes d'une table.

# Commande INSERT

- Ajoute de nouvelles lignes dans une table.

```
INSERT INTO employees (name, department, salary)  
VALUES ('John Doe', 'IT', 50000);
```

# Insérer Plusieurs Lignes

- Ajoute plusieurs lignes en une seule commande.

```
INSERT INTO employees (name, department, salary)
VALUES
    ('Jane Smith', 'HR', 60000),
    ('Alice Johnson', 'Finance', 70000);
```

# Commande UPDATE

- Modifie les données existantes dans une table.

```
UPDATE employees  
SET salary = 55000  
WHERE name = 'John Doe';
```



# Mettre à Jour Plusieurs Colonnes

- Modifie plusieurs colonnes en une seule commande.

```
UPDATE employees  
SET department = 'Marketing', salary = 65000  
WHERE name = 'Jane Smith';
```

# Commande DELETE

- Supprime des lignes spécifiques d'une table.

```
DELETE FROM employees  
WHERE name = 'John Doe';
```

# Supprimer Toutes les Lignes

- Supprime toutes les lignes d'une table.

```
DELETE FROM employees;
```

# DQL (Data Query Language)

# DQL dans PostgreSQL

- Utilisé pour récupérer des données de la base de données.
- Principale commande DQL : SELECT.

```
SELECT * FROM employees WHERE department = 'IT';
```

# Commande **SELECT**

- Récupère des données d'une table.

```
SELECT * FROM employees;  
SELECT firstname, lastname, salary FROM employees;
```

# ALIAS

- Simplifie les noms de colonnes ou de tables dans une requête.

```
SELECT name AS employee_name, salary AS employee_salary  
FROM employees AS e;
```

## ORDER BY

- Trie les résultats par une ou plusieurs colonnes de manière croissant (**ASC**) ou décroissante (**DESC**)

```
SELECT name, salary  
FROM employees  
ORDER BY salary DESC;
```



# DISTINCT

- Retourne des résultats uniques par rapport à l'ensemble des colonnes sélectionnées

```
SELECT DISTINCT department  
FROM employees;
```

# WHERE

- Filtre les lignes selon une condition.

```
SELECT name, department  
FROM employees  
WHERE department = 'IT';
```

# Opérateurs de la clause WHERE

Signe	Description	Exemple
=	Égalité	WHERE age = 30
<>	Inégalité (différent de)	WHERE age <> 30
!=	Inégalité (différent de)	WHERE age != 30
>	Supérieur	WHERE salary > 50000
<	Inférieur	WHERE salary < 50000
>=	Supérieur ou égal	WHERE age >= 18
<=	Inférieur ou égal	WHERE age <= 65

# AND

- Combine plusieurs conditions (toutes doivent être vraies).

```
SELECT name, salary  
FROM employees  
WHERE department = 'IT' AND salary > 50000;
```

# OR

- Combine plusieurs conditions (au moins une doit être vraie).

```
SELECT name, department  
FROM employees  
WHERE department = 'IT' OR department = 'HR';
```

# IN

- Filtre selon plusieurs valeurs spécifiques.

```
SELECT name, department  
FROM employees  
WHERE department IN ('IT', 'HR', 'Finance');
```

## BETWEEN

- Filtre selon une plage de valeurs.

```
SELECT name, salary  
FROM employees  
WHERE salary BETWEEN 40000 AND 60000;
```

# LIKE

- Filtre selon un modèle.
- ○ `%` : Remplace zéro, un ou plusieurs caractères.
- ○ `_` : Remplace un seul caractère.

```
SELECT name  
FROM employees  
WHERE name LIKE 'J%';
```



# IS NULL

- Filtre les valeurs NULL.

```
SELECT name, email  
FROM employees  
WHERE email IS NULL;
```

# Fonction d'aggrégations et GROUP BY

- **Fonctions d'agrégation :**

- Effectuent des calculs sur un ensemble de valeurs
- Renvoient un résultat unique (un scalaire)
- Exemples : `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`, `STRING_AGG`

- **Clause `GROUP BY` :**

- Regroupe les lignes par valeurs communes dans une ou plusieurs colonnes
- Applique les fonctions d'agrégation à chaque groupe

# Fonction COUNT

- `COUNT(*)` : Compte toutes les lignes d'une table.

```
SELECT COUNT(*) AS total_employees  
FROM employees;
```

- `COUNT(column_name)` : Compte les lignes où la colonne spécifiée n'est pas NULL.

```
SELECT COUNT(department) AS total_departments  
FROM employees;
```

# Fonction SUM

- `SUM(column_name)` : Calcule la somme des valeurs d'une colonne numérique.

```
SELECT SUM(salary) AS total_salary  
FROM employees;
```

# Fonction AVG

- `AVG(column_name)` : Calcule la moyenne des valeurs d'une colonne numérique.

```
SELECT AVG(salary) AS average_salary  
FROM employees;
```

# Fonction MIN

- `MIN(column_name)` : Trouve la valeur minimale d'une colonne.

```
SELECT MIN(salary) AS minimum_salary  
FROM employees;
```

# Fonction MAX

- `MAX(column_name)` : Trouve la valeur maximale d'une colonne.

```
SELECT MAX(salary) AS maximum_salary  
FROM employees;
```

# Fonction **STRING\_AGG**

- `STRING_AGG(column_name, delimiter)` : Concatène les valeurs d'une colonne avec un délimiteur.

```
SELECT STRING_AGG(name, ', ') AS employee_names  
FROM employees;
```



## GROUP BY

- Groupe les résultats par une ou plusieurs colonnes en utilisant ou non une fonction d'agrégation

```
SELECT department, AVG(salary) AS avg_salary  
FROM employees  
GROUP BY department;
```

# HAVING

- Filtre les enregistrements après un regroupement selon une condition.

```
SELECT department, AVG(salary) AS avg_salary  
FROM employees  
GROUP BY department  
HAVING AVG(salary) > 50000;
```

# LIMIT

- Limite le nombre de résultats retournés.

```
SELECT name, salary  
FROM employees  
ORDER BY salary DESC  
LIMIT 5;
```

# FETCH

- Limite les lignes retournées après une certaine ligne.

```
SELECT name, salary  
FROM employees  
ORDER BY salary DESC  
OFFSET 10 ROWS  
FETCH NEXT 5 ROWS ONLY;
```

## INNER JOIN

- Récupère les lignes ayant des valeurs correspondantes dans les deux tables.

```
SELECT e.name, d.department_name  
FROM employees e  
INNER JOIN departments d ON e.department_id = d.department_id;
```

## LEFT JOIN

- Récupère toutes les lignes de la table de gauche et les lignes correspondantes de la table de droite.

```
SELECT e.name, d.department_name  
FROM employees e  
LEFT JOIN departments d ON e.department_id = d.department_id;
```

## RIGHT JOIN

- Récupère toutes les lignes de la table de droite et les lignes correspondantes de la table de gauche.

```
SELECT e.name, d.department_name  
FROM employees e  
RIGHT JOIN departments d ON e.department_id = d.department_id;
```

## FULL JOIN

- Récupère toutes les lignes lorsqu'il y a une correspondance dans l'une des tables.

```
SELECT e.name, d.department_name  
FROM employees e  
FULL JOIN departments d ON e.department_id = d.department_id;
```



## Sous-Requête

- Une sous-requête (ou requête imbriquée) est une requête à l'intérieur d'une autre requête SQL.

```
SELECT name, salary  
FROM employees  
WHERE salary > (SELECT AVG(salary) FROM employees);
```

- Cette requête sélectionne les employés dont le salaire est supérieur à la moyenne des salaires de tous les employés.

## Sous-Requête Corrélée

- Une sous-requête corrélée est une sous-requête qui fait référence à des colonnes de la table de la requête principale.

```
SELECT e1.name, e1.salary  
FROM employees e1  
WHERE e1.salary > (SELECT AVG(e2.salary) FROM employees e2 WHERE e2.department = e1.department);
```

- Cette requête sélectionne les employés dont le salaire est supérieur à la moyenne des salaires de leur département.

# ANY

- L'opérateur **ANY** compare une valeur avec un ensemble de valeurs et retourne vrai si la comparaison est vraie pour au moins une des valeurs de l'ensemble.

```
SELECT name, salary  
FROM employees  
WHERE salary > ANY (SELECT salary FROM employees WHERE department = 'HR');
```

- Cette requête sélectionne les employés dont le salaire est supérieur à au moins un des salaires des employés du département des RH.

# ALL

- L'opérateur **ALL** compare une valeur avec un ensemble de valeurs et retourne vrai si la comparaison est vraie pour toutes les valeurs de l'ensemble.

```
SELECT name, salary  
FROM employees  
WHERE salary > ALL (SELECT salary FROM employees WHERE department = 'HR');
```

- Cette requête sélectionne les employés dont le salaire est supérieur à tous les salaires des employés du département des RH.

# EXISTS

- L'opérateur **EXISTS** vérifie l'existence de lignes retournées par une sous-requête et retourne vrai si la sous-requête retourne au moins une ligne.

```
SELECT name  
FROM employees e  
WHERE EXISTS (SELECT 1 FROM departments d WHERE d.manager_id = e.employee_id);
```

- Cette requête sélectionne les employés qui sont des gestionnaires de département.

# DCL (Data Control Language)

# DCL dans PostgreSQL

- Utilisé pour contrôler les autorisations d'accès aux données.
- Principales commandes DCL : GRANT, REVOKE.
- `GRANT SELECT ON employees TO user1;` : Accorde à l'utilisateur user1 le droit de sélectionner des données depuis la table employees.
- `REVOKE INSERT ON customers FROM user2;` : Révoque le droit d'insertion de données dans la table customers à l'utilisateur user2.

# Gestion d'utilisateur

- Créer un nouvel utilisateur avec un mot de passe.

```
CREATE USER john_doe WITH PASSWORD 'securepassword';
```

- Changer le mot de passe d'un utilisateur.

```
ALTER USER john_doe WITH PASSWORD 'newpassword';
```

- Supprimer un utilisateur existant.

```
DROP USER john_doe;
```



## Gestion des rôles

- Créer un nouveau rôle.

```
CREATE ROLE manager;
```

- Assigner un rôle à un utilisateur.

```
GRANT manager TO john_doe;
```

- Retirer un rôle d'un utilisateur.

```
REVOKE manager FROM john_doe;
```

# Attribution de privilèges

- Attribuer des privilèges spécifiques à un rôle ou un utilisateur sur une table.
- privilege\_list: SELECT, INSERT, UPDATE, DELETE, ALL

```
GRANT privilege_list | ALL  
ON table_name  
TO role_name;
```

```
GRANT SELECT, INSERT  
ON employees  
TO manager;
```

## Révocation de privilèges

- Retirer des privilèges spécifiques à un rôle ou un utilisateur sur une table.

```
REVOKE privilege_list | ALL  
ON table_name  
FROM role_name;
```

```
REVOKE SELECT, INSERT  
ON employees  
FROM manager;
```

# TCL (Transaction Control Language)

## Définition d'une Transaction

- Une transaction est une unité de travail qui se compose d'une ou plusieurs opérations SQL exécutées de manière atomique.
- Toutes les opérations dans une transaction doivent réussir ou échouer ensemble.
- PostgreSQL est conforme à la norme **ACID** depuis 2001

## Propriétés ACID

- **Atomicité:** Toutes les opérations réussissent ou échouent ensemble.
- **Cohérence:** La base de données passe d'un état valide à un autre état valide.
- **Isolation:** Les transactions concurrentes n'interfèrent pas les unes avec les autres.
- **Durabilité:** Une fois validées, les modifications sont permanentes même en cas de panne.

# Transactions Automatiques

- PostgreSQL exécute chaque commande SQL dans une transaction implicite par défaut.
- Pour les transactions explicites, utilisez `BEGIN`, `COMMIT`, et `ROLLBACK`.

# Utilisation des transactions

- Démarrer une nouvelle transaction avec l'instruction `BEGIN`

```
BEGIN;
```

- Valider toutes les opérations effectuées dans la transaction, rendant les changements permanents.

```
COMMIT;
```

- Annuler toutes les opérations effectuées dans la transaction, rétablissant l'état initial.

```
ROLLBACK;
```



## Points de Sauvegarde

- Créer un point de sauvegarde à partir duquel on peut revenir partiellement.

```
SAVEPOINT savepoint_name;
```

- Annuler les opérations jusqu'au point de sauvegarde spécifié sans annuler la transaction entière.

```
ROLLBACK TO SAVEPOINT savepoint_name;
```

- Libérer un point de sauvegarde.

```
RELEASE SAVEPOINT savepoint_name;
```

**Merci pour votre attention.**

**Des questions ?**

