BASES DE DONNÉES

COURS 3.1

INSERTION MODIFICATION SUPPRESSION

INSERT INTO
UPDATE SET
DELETE FROM

INSÉRER DE LA DONNÉE

- Pour insérer de la donnée, on utilisera la clause INSERT:
  - Champs explicites :

```sql
INSERT INTO ma_table (champ_1, champ2, champ_3)
    VALUES (1, 'hello', 'foo’);
```

  - Champs implicites :
    - Attention, ajouter des valeurs de manière implicite nécessite de connaitre à l’avance l’ordre EXACT des champs de la table.

```sql
INSERT INTO ma_table VALUES (1, 'hello', 'world');
```

  - Insertion multiple :
    - Il suffit de chaîner les données avec une virgule

```sql
INSERT INTO ma_table (champ_1, champ2, champ_3) VALUES
    (1, 'hello', 'foo'),
    (2, 'world', 'bar');
```

INSÉRER DE LA DONNÉE

- On peut également insérer de la donnée grâce à une sous requête, pour peu que le type des champs retournés par la sous requêtes correspondent aux types des champs à insérer :

```sql
INSERT INTO utilisateur (id, nom, prenom)
    SELECT id, nom, prenom
    FROM tmp_utilisateur
    WHERE age < 100;
```

MODIFIER DE LA DONNÉE

- La clause UPDATE est utilisée pour modifier une donnée existante:
  - Attention, si on modifie une donnée sous contrainte, et que la modification ne respecte pas ces contraintes, la requêtes échouera.

```sql
UPDATE utilisateurs SET prenom = 'Florent' WHERE id = 1;

UPDATE utilisateurs
SET prenom = 'Florent', nom = 'Bejina'
WHERE id = 1;

UPDATE eleves
SET note = note - 1
WHERE id IN (SELECT DISTINCT id_eleve FROM absents);

UPDATE vendeurs
SET nb_ventes = (
    SELECT COUNT(*)
    FROM ventes
    WHERE id_vendeur = vendeurs.id
)
```

SUPPRIMER DE LA DONNÉE

- La clause DELETE est utilisée comme ceci :
  - Attention, si on supprime une donnée sous contrainte, et que le résultat de la requête fait invalide cette contrainte, la requêtes échouera.

```sql
DELETE FROM films;

DELETE FROM films WHERE id = 2;

DELETE FROM films WHERE type = 'musical' OR duration < 90;

DELETE FROM films WHERE producer_id = (
    SELECT id FROM producers WHERE producers.name = 'foo'
);

DELETE FROM films USING producers
  WHERE producer_id = producers.id AND producers.name =
'foo';
```

- Les deux dernières requêtes sont équivalentes.

RETOURNER LA DONNÉE DES LIGNES MODIFIÉE

- Dans le cas d'une modification de donnée, postgresql (comme d'autres SGBD) retournera par défaut le nombre de lignes insérées / modifiées / supprimées. Il se peut que quelques fois nous ayons besoin d'exploiter cette donnée modifiée directement depuis la requête en question.
- On utilise pour cela la clause RETURNING
  - Cette clause retourne une structure de données s'apparentant à une table **mais qui ne peut pas être utilisée dans une sous requête**.

```sql
INSERT INTO users (firstname, lastname)
  VALUES ('Joe', 'Cool')
  RETURNING id;

UPDATE products
  SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;

DELETE FROM products
  WHERE availability = 'unavailable'
  RETURNING *;
```

DES VARIABLES AVEC POSTGRESQL ?

- Avec postgresql il est possible de stocker le résultat dans un « genre » de variable afin de la réutiliser ensuite.
- Cela se fait avec la clause WITH
  - Exemples :

```
1 WITH tmp_table AS (
2     SELECT * FROM users WHERE height = 175
3 ) SELECT id FROM tmp_table;
4
5
6 WITH tmp_users AS (
7   SELECT * FROM users WHERE age < 30 AND age > 20
8 ), avg_height AS (
9   SELECT AVG(height) FROM users
10 )
11 SELECT *
12 FROM tmp_users
13 WHERE height > avg_height;
14
```

POUR ALLER PLUS LOIN

Que fait cette suite d'instructions ?
Quel est le retour final ?

```sql
CREATE TABLE users (
  id serial PRIMARY KEY,
  firstname text
);

CREATE TABLE settings (
  name text,
  value text
);

INSERT INTO settings (name, value)
    VALUES ('last_user_id', null);


WITH inserted_ids AS (
  INSERT INTO users (firstname)
  VALUES ('florent')
  RETURNING id
)
UPDATE settings
SET value = (
  SELECT id FROM inserted_ids LIMIT 1
)
WHERE name = 'last_user_id'
RETURNING *;
```

POUR ALLER PLUS LOIN

Que fait cette suite d'instructions ?
Quel est le retour final ?

name	value
last_user_id	1

```sql
CREATE TABLE users (
  id serial PRIMARY KEY,
  firstname text
);

CREATE TABLE settings (
  name text,
  value text
);

INSERT INTO settings (name, value)
    VALUES ('last_user_id', null);


WITH inserted_ids AS (
  INSERT INTO users (firstname)
  VALUES ('florent')
  RETURNING id
)
UPDATE settings
SET value = (
  SELECT id FROM inserted_ids LIMIT 1
)
WHERE name = 'last_user_id'
RETURNING *;
```

BONUS, L'UPSERT

- Un « upsert » tiens plus d'un concept.
- Upsert est une contraction d' « update » et d' « insert ». C'est une opération qui insère des lignes dans une table de base de données (si elles n'existent pas déjà) ou les met à jour (si elles existent).
- Il n'existe pas de clause UPSERT dans les SGBD, mais on peut réaliser cette opération avec la clause ON CONFLICT du INSERT (avec postgresql), et le EXCLUDED qui représente les valeurs rejetées.

```
1 INSERT INTO eleves (nom, prenom, email)
2   VALUES ('Le Méchant', 'Rastapopoulos','rastapopoulos@le-
  mechant.fr')
3 ON CONFLICT (email)
4 DO
5    UPDATE SET nom = EXCLUDED.nom, prenom = eleves.prenom;
```

Voir https://www.postgresql.org/docs/14/sql-insert.html pour plus d'informations sur la clause ON CONFLICT

UPSERT - EXEMPLE

- Quel est le résultat de cette suite d'instructions ?

```sql
CREATE TABLE visites (
  ip VARCHAR(15) UNIQUE,
  nb_visites INTEGER NOT NULL DFAULT 1,
  navigateur TEXT DEFAULT 'unknown'
);

INSERT INTO visites (ip, navigateur)
  VALUES ('111.111.111.111', 'chrome')
ON CONFLICT (ip) DO
  UPDATE SET
    nb_visites = visites.nb_visites + 1,
    navigateur = EXCLUDED.navigateur;

INSERT INTO visites (ip, navigateur)
  VALUES ('222.222.222.222', 'firefox')
ON CONFLICT (ip) DO
  UPDATE SET nb_visites = visites.nb_visites + 1,
    navigateur = EXCLUDED.navigateur;

INSERT INTO visites (ip, navigateur)
  VALUES ('111.111.111.111', 'firefox')
ON CONFLICT (ip) DO
  UPDATE SET nb_visites = visites.nb_visites + 1,
    navigateur = EXCLUDED.navigateur;

SELECT * FROM visites;
```

UPSERT - EXEMPLE

- Quel est le résultat de cette suite d'instructions ?

ip	nb_visites	navigateur
222.222.222.222	1	firefox
111.111.111.111	2	firefox

```sql
CREATE TABLE visites (
  ip VARCHAR(15) UNIQUE,
  nb_visites INTEGER NOT NULL DFAULT 1,
  navigateur TEXT DEFAULT 'unknown'
);

INSERT INTO visites (ip, navigateur)
  VALUES ('111.111.111.111', 'chrome')
ON CONFLICT (ip) DO
  UPDATE SET
    nb_visites = visites.nb_visites + 1,
    navigateur = EXCLUDED.navigateur;

INSERT INTO visites (ip, navigateur)
  VALUES ('222.222.222.222', 'firefox')
ON CONFLICT (ip) DO
  UPDATE SET nb_visites = visites.nb_visites + 1,
    navigateur = EXCLUDED.navigateur;

INSERT INTO visites (ip, navigateur)
  VALUES ('111.111.111.111', 'firefox')
ON CONFLICT (ip) DO
  UPDATE SET nb_visites = visites.nb_visites + 1,
    navigateur = EXCLUDED.navigateur;

SELECT * FROM visites;
```

TRI ET LIMITES

ORDER BY
LIMIT OFFSET

TRIER LES DONNÉES

- Il est possible d'ordonner les résultats d'une requête avec la clause « **ORDER BY** ».
- ORDER BY prend en paramètre au moins nom de colonne auquel on ajoute (éventuellement) un ordre de tri :
  - Ascendant : ASC (a -> Z, 0 -> n)
  - Descendant : DESC (Z -> a, n -> 0)
- Si aucun ordre n'est spécifié, l'ordre ascendant est appliqué par défaut.
- Il est possible de trier selon plusieurs colonnes, il suffit de les chainer avec une virgule.

```sql
SELECT prenom, nom, age FROM eleves
ORDER BY prenom DESC;

SELECT prenom, nom age FROM eleves
ORDER BY nom, prenom, age DESC;
```

LIMITER LES RÉSULTATS

- Il est souvent conseillé de limiter le nombre de résultats à afficher (optimisation par exemple)
- Cette limite se fait avec les clauses « **LIMIT** » et « **OFFSET** ».
  - LIMIT permet de définir le nombre de résultats à afficher
  - OFFSET permet de définir à partir de quel ligne on affiche les données.
- LIMIT et OFFSET sont constamment suivi un nombre
- OFFSET n'est pas une clause obligatoire. Dans le cas où il n'est pas renseigné, sa valeur par défaut est 0.

```sql
-- les 20 premiers elèves triés par ordre alphabétique
SELECT prenom, nom, age FROM eleves
ORDER BY nom, prenom
LIMIT 20 OFFSET 0;

-- les 20 élèves suivants, triés par ordre alphabétique
SELECT prenom, nom, age FROM eleves
ORDER BY nom, prenom
LIMIT 20 OFFSET 20;
```

LES VUES

CREATE VIEW

CREATE MATERIALIZED VIEW

QU'EST CE QU'UNE VUE

- Une vue est une table (pré)calculée à partir d'une requête SELECT.
- Le résultat du SELECT est donc placé dans une vue, qui devient accessible comme une table.
- Cette vue peut être de deux natures :
  - Calculée à chaque accès
  - Calculée une fois, puis mise à jour « manuellement »
- On choisira la nature de la vue en fonction de la volumétrie de données.
  - Une vue de 10 millions de lignes ne devrait pas être recalculée à la volée à chaque accès
- Une vue peut être requêtée comme une table avec un SELECT.

```sql
SELECT * FROM ma_vue WHERE mon_champs = 'ma_valeur';
```

CRÉER UNE VUE - SYNTAXE

- CREATE VIEW définit une vue d'après une requête. La vue n'est pas matérialisée physiquement. Au lieu de cela, la requête est lancée chaque fois qu'une vue est utilisée dans une requête.
- CREATE **OR REPLACE** VIEW a la même finalité, mais **si une vue du même nom existe déjà, elle est remplacée**. La nouvelle requête doit générer les mêmes colonnes que celles de l'ancienne requête (c'est-à-dire les mêmes noms de colonnes dans le même ordre avec les mêmes types de données). Par contre, elle peut ajouter des colonnes supplémentaires en fin de liste.

```sql
CREATE VIEW view_comedies AS
    SELECT *
    FROM films
    WHERE genre = 'Comédie';

CREATE OR REPLACE VIEW view_comedies_acteurs AS
  SELECT f.id, f.titre, a.prenom, a.nom
  FROM films f
    JOIN films_acteurs fa ON f.id = fa.film_id
    JOIN acteurs a ON a.id = fa.acteur_id
  WHERE f.genre = 'Comédie';
```

CRÉER UNE VUE PERSISTANTE - SYNTAXE

- CREATE MATERIALIZED VIEW définit une vue matérialisée à partir d'une requête. La requête est exécutée et utilisée pour peupler la vue à l'exécution de la commande et peut être rafraichi plus tard en utilisant REFRESH MATERIALIZED VIEW.

```
1 CREATE MATERIALIZED VIEW view_comedies AS
2     SELECT *
3     FROM films
4     WHERE genre = 'Comédie';
5
6 REFRESH MATERIALIZED VIEW view_comedies;
7
8 REFRESH MATERIALIZED VIEW view_comedies WITH NO DATA;
```

SUPPRIMER UNE VUE

- On utilisera la clause DROP VIEW ou DROP MATERIALIZED VIEW comme ceci :

```
1 DROP MATERIALIZED VIEW view_1;
2 DROP MATERIALIZED VIEW IF EXISTS view_2;
3 DROP VIEW view_3;
4 DROP VIEW IF EXISTS view_4;
```

LES AGRÉGATS

GROUP BY
HAVING

DÉFINITION

- L'agrégation, c'est l'action de regrouper des éléments.
- En SQL, lorsque les données sont agrégées, on peut leur appliquer un certain nombre de fonctions d'agrégation pour en tirer des résultats comme des moyennes, des maximum, etc…
- On peut alors répondre à des questions comme
  - « quelle est la moyenne d'âge des étudiants en SIE » ou,
  - « Quelle est la note maximum obtenue en cours de base de données des étudiants en GB »

SYNTAXE

- Une agrégation se définit avec le clause « **GROUP BY** ».
- Cette clause est suivie d'une liste d'au moins un nom de colonne. Cette liste défini l'ordre d'agrégation.

- Par exemple, ces deux requêtes donneront un résultat différent.

```sql
SELECT * FROM eleves GROUP BY age, specialisation;
-- est différent de
SELECT * FROM eleves GROUP BY specialisation, age;
```

FONCTIONNEMENT

- Lorsqu'on réalise une agrégation, la base fait des « paquets » de données.

- Exemple

eleves

id	prenom	spe	age
1	Johan	SIEE	21
2	Anne Claire	GSI	21
3	Sabrina	GSI	20
4	Laurine	SIEE	23
5	Sami	SIEE	22
6	Xavier	SIEE	22

```
SELECT *
FROM eleves
GROUP BY age
```

?

FONCTIONNEMENT

- Lorsqu'on réalise une agrégation, la base fait des « paquets » de données.

- Réponse

eleves

id	prenom	spe	age
1	Johan	SIEE	21
2	Anne Claire	GSI	21
3	Sabrina	GSI	20
4	Laurine	SIEE	23
5	Sami	SIEE	22
6	Xavier	SIEE	22

```
SELECT *
FROM eleves
GROUP BY age
```

id	prenom	spe	age
3	Sabrina	GSI	20
1, 2	Anne Claire, Johan	GSI, SIEE	21
5, 6	Sami, Xavier	SIEE, SIEE	22
4	Laurine	SIEE	23

FONCTIONNEMENT

- Ces données agrégées ne sont pas exploitables telles quelles. En effet, quelle valeur pourrait choisir la base de données si on effectuait la requête

```
SELECT prenom, age FROM eleves GROUP BY age
```

eleves

id	prenom	spe	age
1	Johan	SIEE	21
2	Anne Claire	GSI	21
3	Sabrina	GSI	20
4	Laurine	SIEE	23
5	Sami	SIEE	22
6	Xavier	SIEE	22

prenom	age
Sabrina	20
????	21
????	22
Laurine	23

LES FONCTIONS D'AGRÉGATS

- Pour pouvoir manipuler des données agrégées, il faut leur appliquer des fonctions d'agrégats, telles que COUNT, AVG, MIN, MAX, SUM, etc…
  - Liste exhaustive ici : https://docs.postgresql.fr/11/functions-aggregate.html

eleves			
id	**prenom**	**spe**	**age**
1	Johan	SIEE	21
2	Anne Claire	GSI	21
3	Sabrina	GSI	20
4	Laurine	SIEE	23
5	Sami	SIEE	22
6	Xavier	SIEE	22

```
SELECT spe, avg(age) AS moy_age
FROM eleves
GROUP BY spe
```

?

LES FONCTIONS D'AGRÉGATS

- Pour pouvoir manipuler des données agrégées, il faut leur appliquer des fonctions d'agrégats, telles que COUNT, AVG, MIN, MAX, SUM, etc…
  - Liste exhaustive ici : https://docs.postgresql.fr/11/functions-aggregate.html

eleves

id	prenom	spe	age
1	Johan	SIEE	21
2	Anne Claire	GSI	21
3	Sabrina	GSI	20
4	Laurine	SIEE	23
5	Sami	SIEE	22
6	Xavier	SIEE	22

```
SELECT spe, avg(age) AS moy_age
FROM eleves
GROUP BY spe
```

moy_age	spe
20,5	SIEE
22	GSI

AGRÉGER PLUSIEURS COLONNES

- Dans le cas ou l'agrégation se fait sur plusieurs colonnes, les agrégats de données sont réalisés pour chaque sous ensemble d'agrégat.

eleves				
id	**prenom**	**spe**	**age**	**note**
1	Johan	SIEE	21	5
2	Anne Claire	GSI	21	12
3	Sabrina	GSI	20	13
4	Laurine	SIEE	23	15
5	Sami	SIEE	22	12
6	Xavier	SIEE	22	16
7	Loubna	GSI	21	19
8	Daniil	SIEE	20	11
9	Simon	SIEE	21	9

```
SELECT * FROM eleves GROUP BY spe, age
```

spe	age	id	prenom	note
SIEE	21	1, 9	Johan, Simon	5, 9
	20	8	Daniil	11
	22	5, 6	Sami, Xavier	16, 12
	23	4	Laurine	15
GSI	21	7, 2	Loubna, AC	13, 19
	20	3	Sabrina	12

PRÉCISIONS

- Lors d'une requête avec un GROUP BY, seuls les instructions suivantes sont autorisés dans le SELECT :
  - Les champs nommés dans le GROUP BY
  - Les autres champs pour peu qu'une fonction d'agrégation leur soit appliqués
- Cela pour effet que la requête suivante n'est pas « exécutable » et produira une erreur :

```
SELECT * FROM eleves GROUP BY spe, age
```

- Mais que la requête suivante fonctionnera

```
SELECT age FROM eleves GROUP BY spe, age
```

FILTRER SUR DES AGRÉGATS

- Il est possible de filtrer des résultats sur des données agrégées avec la clause **HAVING**, suivi d'une ou plusieurs condition logiques.
- La clause HAVING est similaire à la clause FROM, sauf qu'elle ne traite que des agrégats.
- La clause HAVING accepte, comme le SELECT
  - Les champs nommés dans le GROUP BY
  - Les autres champs pour peu qu'une fonction d'agrégation leur soit appliqués

```sql
SELECT spe, age, AVG(note) AS avg_note
FROM eleves
GROUP BY spe, age
HAVING AVG(note) > 10 AND spe = 'SIEE';
```

ATTENTION !

- HAVING et WHERE sont deux clauses complètement différentes,
  - HAVING s'occupe de filtrer des agrégats
  - WHERE s'occupe de filtrer des lignes qui seront ensuite incluses dans les agrégats
- Ces deux requêtes donnent donc un résultat différent

```sql
SELECT spe, age, AVG(note) AS avg_note
FROM eleves
GROUP BY spe, age
HAVING AVG(note) > 10 AND AVG(note) < 19;
```

spe	age	avg_note
SIEE	23	15
SIEE	20	11
SIEE	22	14
GSI	20	13
GSI	21	15,5

```sql
SELECT spe, age, AVG(note) AS avg_note
FROM eleves
WHERE note > 10 AND note < 19
GROUP BY spe, age;
```

spe	age	avg_note
GSI	20	13
GSI	21	12
SIEE	20	11
SIEE	22	14
SIEE	23	15

ATTENTION ! (2)

Dans le cadre de cette modélisation :

```sql
CREATE TABLE categories (
  id serial PRIMARY KEY,
  name text
);
CREATE TABLE products (
  id serial PRIMARY KEY,
  category_id integer,
  price integer,
  name text,
  CONSTRAINT fk_test FOREIGN KEY (category_id) REFERENCES categories(id)
);

```

Comment afficher le prix moyen des produits, par catégorie, ainsi que le nom de la catégorie ?

ATTENTION ! (2)

Dans le cadre de cette modélisation :

```sql
CREATE TABLE categories (
  id serial PRIMARY KEY,
  name text
);
CREATE TABLE products (
  id serial PRIMARY KEY,
  category_id integer,
  price integer,
  name text,
  CONSTRAINT fk_test FOREIGN KEY (category_id) REFERENCES categories(id)
);

```

Comment afficher le prix moyen des produits, par catégorie, ainsi que le nom de la catégorie ?

```sql
SELECT categories.name, AVG(price) AS avg_price
FROM categories JOIN products
    ON categories.id = products.category_id
GROUP BY categories.id;

```