

# 5. Le langage SQL

---

# Introduction

---

- **SQL** : **S**tructured **Q**uery **L**anguage
- Inventé chez IBM (centre de recherche d'Almaden en Californie), en 1974 par Astrahan & Chamberlin dans le cadre de System R
- Le langage SQL est normalisé
  - SQL2: adopté (SQL 92)
  - SQL3: adopté (SQL 99) → SQL-2008
- Standard d'accès aux bases de données relationnelles




# SQL : Trois langages en un

---

- SQL = Langage de définition de données (LDD)
  - CREATE TABLE
  - ALTER TABLE
  - DROP TABLE
- SQL = Langage de manipulation de données (LMD)
  - INSERT INTO
  - UPDATE
  - DELETE FROM
- SQL = Langage de requêtes (LMD) /ou LID Langage d'Intérogation de données
  - SELECT ... FROM ... WHERE ...
    - Sélection
    - Projection
    - Jointure
  - Les agrégats

# Terminologie

---

- Relation  Table
- Tuple  Ligne
- Attribut  Colonne

# SQL (LDD)

---

**Un langage de définition de données**

# Types de données

---

- Une base de données contient des **tables**
- Une table est organisée en **colonnes**
- Une colonne stocke des **données**
  
- Les données sont séparées en plusieurs **types** !

# Type des colonnes (en MySQL)

---

## ■ Numériques

- NUMERIC : idem DECIMAL //valeur exacte
- DECIMAL. Possibilité DECIMAL(M,D) M chiffre au total //valeur exacte
- INTEGER
  - TINYINT 1 octet (de -128 à 127)
  - SMALLINT 2 octets (de -32768 à 32767)
  - MEDIUMINT 3 octets (de -8388608 à 8388607)
  - INT 4 octets (de -2147483648 à 2147483647)
  - BIGINT 8 octets (de -9223372036854775808 à 9223372036854775807)
  - Possibilité de donner la taille de l'affichage : INT(6) => 674 s'affiche 000674
  - Possibilité de spécifier UNSIGNED
    - INT UNSIGNED => de 0 à 4294967296
- FLOAT : 4 octets par défaut. Possibilité d'écrire FLOAT(P) //valeur approchée
- REAL : 8 octets (4 octets dans d'autres SGBD) //valeur approchée
- DOUBLE : 8 octets //valeur approchée
  - 56,6789 → MySQL stockera 56,67890000000000000001

# Type des colonnes (en MySQL)

---

- Date et Heure
  - DATETIME
    - AAAA-MM-JJ HH:MM:SS
    - de 1000-01-01 00:00:00 à '9999-12-31 23:59:59
  - DATE
    - AAAA-MM-JJ
    - de 1000-01-01 à 9999-12-31
  - TIMESTAMP
    - Date sans séparateur AAAAMMMJJHHMMSS
  - TIME
    - HH:MM:SS (ou HHH:MM:SS)
    - de -838:59:59 à 838:59:59
  - YEAR
    - YYYY
    - de 1901 à 2155



# Type des colonnes (en MySQL)

## ■ Chaînes

- CHAR(n)  $1 \leq n \leq 255$
- VARCHAR(n)  $1 \leq n \leq 255$

Exemple :

	CHAR(4)		VARCHAR(4)	
Valeur	Stockée	Taille	Stockée	Taille
"	' '	4 octets	"	1 octets
'ab'	'ab '	4 octets	'ab'	3 octets
'abcd'	'abcd'	4 octets	'abcd'	5 octets
'abcdef'	'abcd'	4 octets	'abcd'	5 octets

# Type des colonnes (en MySQL)

---

## ■ Chaînes

- TINYBLOB Taille <  $2^8$  caractères
- BLOB Taille <  $2^8$  caractères
- MEDIUMBLOB Taille <  $2^{24}$  caractères
- LONGBLOB Taille <  $2^{32}$  caractères
  
- TINYTEXT Taille <  $2^8$  caractères
- TEXT Taille <  $2^8$  caractères
- MEDIUMTEXT Taille <  $2^{24}$  caractères
- LONGTEXT Taille <  $2^{32}$  caractères

Les tris faits sur les BLOB tiennent compte de la casse, contrairement aux tris faits sur les TEXT.

# Type des colonnes (en MySQL)

---

## ■ ENUM //valeur décimal

- Enumération
- ENUM("un", "deux", "trois")
- Valeurs possibles : "" , "un", "deux", "trois"
- Au plus 65535 éléments

## ■ SET //valeur binaire

- Ensemble
- SET("un", "deux")
- Valeurs possibles : "" , "un", "deux", "un,deux"
- Au plus 64 éléments

# Type des colonnes (en MySQL)

---

- Dans quelles situations faut-il utiliser ENUM ou SET ?

**JAMAIS !!**

- il faut toujours éviter autant que possible les fonctionnalités propres à un seul SGBD.

# Un langage de définition de données

---

Commandes pour Créer et supprimer une base de données:

- **CREATE DATABASE** *nom\_base*: créer une base de données,
  - **CREATE DATABASE** *bibliotheque* **CHARACTER SET** *'utf8'* : créer une base de données et encoder les tables en UTF-8
- 
- **DROP DATABASE** *bibliotheque* : supprimer la base de données,
  - **DROP DATABASE IF EXISTS** *bibliotheque* ;
- 

Utilisation d'une base de données

- **USE** *bibliotheque* ;

# Un langage de définition de données

---

Commandes pour créer, modifier et supprimer les éléments du schéma:

- **CREATE TABLE** : créer une table (une relation),
- **CREATE VIEW** : créer une vue particulière sur les données à partir d'un SELECT,
- **DROP {TABLE | VIEW}** : supprimer une table ou une vue,
- **ALTER {TABLE | VIEW}** : modifier une table ou une vue.

# CREATE TABLE

---

Commande créant une table en donnant son **nom**, ses **attributs** et ses **contraintes**:

```
CREATE TABLE [IF NOT EXISTS] nom_table (  
    colonne1 description_colonne1,  
    [colonne2 description_colonne2,  
    colonne3 description_colonne3,  
    ...,]  
    [PRIMARY KEY (colonne_clé_primaire)]  
)  
[ENGINE=moteur];
```

# Les moteurs de tables

---

- Les moteurs de tables sont une spécificité de MySQL. Ce sont des moteurs de stockage. Cela permet de gérer différemment les tables selon l'utilité qu'on en a.
- Les deux moteurs les plus connus sont **MyISAM** et **InnoDB**.
- **MyISAM** : C'est le moteur par défaut. Les commandes sont particulièrement rapides sur les tables utilisant ce moteur. Cependant, il ne gère pas certaines fonctionnalités importantes comme **les clés étrangères**.
- **InnoDB** : Plus lent et plus gourmand en ressources que MyISAM, ce moteur **gère les clés étrangères**



# CREATE TABLE

---

## Exemples:

```
CREATE TABLE Emprunteur(  
  id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,  
  nom VARCHAR(20) NOT NULL,  
  prenom VARCHAR(15) NOT NULL,  
  annee_insc YEAR DEFAULT 2021,  
  PRIMARY KEY (id)  
)  
ENGINE=INNODB;
```

# CREATE TABLE

---

## Exemples: Autre possibilité

```
CREATE TABLE Emprunteur(  
    id SMALLINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,  
    nom VARCHAR(20) NOT NULL,  
    prenom VARCHAR(15) NOT NULL,  
    annee_insc YEAR DEFAULT 2021,  
)  
ENGINE=INNODB;
```

# Vérifications

---

Deux commandes pour vérifier la création des tables :

- **SHOW TABLES;**
  - liste les tables de la base de données
- **DESCRIBE Emprunteur;**
  - liste les colonnes de la table avec leurs caractéristiques

# DROP TABLE

---

- **DROP TABLE** : Supprimer une table
  - supprime la table et tout son contenu
- **DROP TABLE** nom\_table [**CASCADE CONSTRAINTS**];
- **CASCADE CONSTRAINTS**
  - Supprime toutes les contraintes référençant une clé primaire (primary key) ou une clé unique (UNIQUE) de cette table
  - Si on cherche à détruire une table dont certains attributs sont référencés sans spécifier CASCADE CONSTRAINT, on a un message d'erreur.

# ALTER TABLE

---

- Modifier la définition d'une table:
  - Changer le nom de la table  
mot clé : **RENAME**
  - Ajouter une colonne ou une contrainte  
mot clé : **ADD**
  - Modifier une colonne ou une contrainte  
mot clé : **MODIFY/CHANGE**
  - Supprimer une colonne ou une contrainte  
mot clé : **DROP**
  - renommer une colonne ou une contrainte  
mot clé : **RENAME**

# ALTER TABLE

---

## Syntaxe :

**ALTER TABLE** nom-table

```
{  RENAME TO nouveau-nom-table
  | ADD (( nom-col type-col [DEFAULT valeur] [contrainte-col])*
  | MODIFY (nom-col [type-col] [DEFAULT valeur] [contrainte-col])*
  | DROP COLUMN nom-col [CASCADE CONSTRAINTS]
  | RENAME COLUMN old-name TO new-name
};
```

# Ajout et suppression d'une colonne

---

**ALTER TABLE** nom\_table

**ADD** [COLUMN] nom\_colonne description\_colonne;

■ Exemple :

**ALTER TABLE** Emprunteur

**ADD COLUMN** date\_emprunt **DATE NOT NULL** ;

# Ajout et suppression d'une colonne

---

**ALTER TABLE** nom\_table

**DROP** [COLUMN] nom\_colonne;

■ Exemple :

**ALTER TABLE** Emprunteur

**DROP COLUMN** date\_emprunt ;



# Modification d'une colonne

---

```
ALTER TABLE nom_table
```

```
CHANGE ancien_nom nouveau_nom description_colonne;
```

■ Exemple :

```
ALTER TABLE Emprunteur
```

```
CHANGE nom nom_famille VARCHAR(10) NOT NULL ;
```

# Changement du type de données

---

**ALTER TABLE** nom\_table

**CHANGE** ancien\_nom nouveau\_nom description\_colonne;

Ou

**ALTER TABLE** nom\_table

**MODIFY** nom\_colonne description\_colonne;

# Des exemples pour illustrer :

**ALTER TABLE** Emprunteur

**CHANGE** nom nom\_famille **VARCHAR**(10) **NOT NULL** ;

→ Changement du type + changement du nom

**ALTER TABLE** Emprunteur

**CHANGE** id id **BIGINT** **NOT NULL** ;

→ Changement du type sans renommer

**ALTER TABLE** Emprunteur

**MODIFY** id **BIGINT** **NOT NULL** **AUTO\_INCREMENT**;

→ Ajout de l'auto-incrémentation

**ALTER TABLE** Emprunteur

**MODIFY** nom **VARCHAR**(30) **NOT NULL** **DEFAULT** 'Anonyme';

→ Changement de la description

```
CREATE TABLE Emprunteur(  
  id SMALLINT UNSIGNED  
    AUTO_INCREMENT PRIMARY KEY,  
  nom VARCHAR(20) NOT NULL,  
  prenom VARCHAR(15) NOT NULL,  
  annee_insc YEAR DEFAULT 2021,  
)  
ENGINE=INNODB;
```

# Renommer une table

---

- ... **RENAME TO** nouveau-nom-table

- Exemple :

**ALTER TABLE** Emprunteur **RENAME TO** Emprunteurs ;

# Les clé étrangères

---

```
CREATE TABLE [IF NOT EXISTS] Nom_table (  
    colonne1 description_colonne1,  
    [colonne2 description_colonne2,  
    colonne3 description_colonne3,  
    ...,]  
    [ [CONSTRAINT [symbole_contrainte]] FOREIGN KEY  
    (colonne(s)_clé_étrangère) REFERENCES table_référence  
    (colonne(s)_référence)  
]  
)  
[ENGINE=moteur];
```

# Exemple

---

- On imagine les tables **Client** et **Commande**,
- pour créer la table **Commande** avec une **clé étrangère** ayant pour **référence la colonne numero** de la table **Client**, on utilisera :

```
CREATE TABLE Commande (  
    numero INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,  
    client INT UNSIGNED NOT NULL,  
    produit VARCHAR(40),  
    quantite SMALLINT DEFAULT 1,  
    CONSTRAINT fk_client_numero          -- On donne un nom à notre clé  
    FOREIGN KEY (client)                  -- Colonne sur laquelle on crée la clé  
    REFERENCES Client(numero)            -- Colonne de référence  
)  
ENGINE=InnoDB;                          -- MyISAM interdit, je le rappelle encore une fois !
```

# Après création de la table

---

**ALTER TABLE** Commande

**ADD CONSTRAINT** fk\_client\_numero **FOREIGN KEY**  
(client) **REFERENCES** Client(numero);

Suppression d'une clé étrangère

**ALTER TABLE** nom\_table

**DROP FOREIGN KEY** symbole\_contrainte;

# Petit TP

---

Créer la base de données et les différentes tables de ce schéma relationnel:

- Personnes(PersonneID, Nom, Age, Adresse);
- Commandes(CommandeID, NumCommande, PersonneID);





# Petit TP

## ■ Solution:

```
CREATE DATABASE gestion;  
USE gestion;
```

```
CREATE TABLE Personnes (  
    PersonneID int AUTO_INCREMENT PRIMARY KEY,  
    Nom VARCHAR(20) NOT NULL,  
    Age int,  
    Adresse VARCHAR(100)  
);
```

```
CREATE TABLE Commandes (  
    CommandeID int AUTO_INCREMENT PRIMARY KEY,  
    NumCommande int NOT NULL,  
    PersonneID int,  
    FOREIGN KEY (PersonneID) REFERENCES Personnes(PersonneID)  
);
```

# SQL (LMD)

---

**Un langage de manipulation de données**

# Manipulation des données

---

- **INSERT INTO** : ajouter un tuple dans une table ou une vue
- **UPDATE** : changer les tuples d'une table ou d'une vue
- **DELETE FROM** : éliminer les tuples d'une table ou d'une vue

# INSERT INTO

---

- Syntaxe :

## INSERT INTO

```
{nom_table | nom_vue}  
[ (nom_col (, nom_col)*) ]  
{ VALUES (valeur (, valeur)*) | sous-requête  
};
```

# Insertion sans préciser les colonnes

- Nous travaillons toujours sur la table Emprunteur composée de 4 colonnes : id, nom, prenom, annee\_insc

**INSERT INTO** Emprunteur

**VALUES** (1, 'Buard', 'Jeremy', '2018');

**INSERT INTO** Emprunteur

**VALUES** (NULL, 'Zuckerberg', 'Mark', NULL);

→ Insert un tuple avec un id=2 et une année = NULL

```
CREATE TABLE Emprunteur(  
  id SMALLINT UNSIGNED AUTO_INCREMENT  
    PRIMARY KEY,  
  nom VARCHAR(20) NOT NULL,  
  prenom VARCHAR(15) NOT NULL,  
  annee_insc YEAR DEFAULT 2021,  
)  
ENGINE=INNODB;
```

## Insertion en précisant les colonnes

---

```
INSERT INTO Emprunteur (nom, prenom,  
annee_insc)  
VALUES ('Chan', 'Priscilla', '2018');
```

```
INSERT INTO Emprunteur (nom, prenom)  
VALUES ('Gates', 'Bill');
```

→ Insert un tuple avec une année = 2021

# Insertion multiple

---

```
INSERT INTO Emprunteur (nom, prenom,  
annee_insc)  
VALUES ('Jobes', 'Steve', '2010'),  
('Moskovitz', 'Dustin', '2011'),  
('Musk', 'Elon', '2013');
```

# UPDATE

---

- Syntaxe :

- **UPDATE** {nom\_table | nom\_vue}  
SET { (nom\_col)\* = (sous-requête)  
| nom\_col = { valeur | (sous-requête)} }\*  
WHERE condition;

- Exemples :

- **UPDATE** Emprunteur  
SET annee\_insc = '2019'  
WHERE nom = 'Musk'
- **UPDATE** Emprunteur  
SET annee\_insc = annee\_insc+2  
WHERE id < 3



# DELETE FROM

---

- Exemple :

- **DELETE FROM** Emprunteur  
WHERE annee\_insc < 2000

- Syntaxe :

- **DELETE FROM** {nom\_table | nom\_vue}  
WHERE condition;

# SQL

---

## Un langage de requêtes

# Structure générale d'une requête

---

- Structure d'une requête formée de trois clauses:
  - SELECT** <liste\_attributs>
  - FROM** <liste\_tables>
  - WHERE** <condition>
- **SELECT** définit le format du résultat cherché
- **FROM** définit à partir de quelles tables le résultat est calculé
- **WHERE** définit les prédicats de sélection du résultat

# Exemple de requête

---

```
SELECT * FROM Emprunteur
```

→ Afficher **tous** les attributs de tous les tuples dans la table “**Emprunteur**”

# Opérateurs de comparaison

---

- = égal
  - WHERE id = 2
- <> différent
  - WHERE nom <> 'Ahmad'
- > plus grand que
  - WHERE annee\_insc > 2010
- >= plus grand ou égal
  - WHERE annee\_insc >= 2018
- < plus petit que
  - WHERE id < 3
- <= plus petit ou égal
  - WHERE id <= 2

# Opérateurs logiques

---

- **AND**

- **WHERE** annee\_insc < 2010 **AND** id<5

- **OR**

- **WHERE** annee\_insc < 2010 **OR** id<5

- Négation de la condition : **NOT**

- **SELECT** \*  
FROM Emprunteur  
WHERE nom = 'Badr'  
**AND NOT** annee\_insc = '2019' ;

# Expressions logiques

---

## Combinaisons:

WHERE

( ensoleillement > 80 **AND** pluviosité < 200 )  
**OR** température > 30

WHERE

ensoleillement > 80  
**AND** ( pluviosité < 200 **OR** température > 30 )

# Appartenance à un ensemble : IN

---

WHERE monnaie = 'Dollar'

OR monnaie = 'Dirham'

OR monnaie = 'Euro'

Équivalent à:

WHERE monnaie IN ('Dollar', 'Dirham', 'Euro')

**NOT IN:** non appartenance à un ensemble



# Comparaison à un ensemble : ALL

---

```
SELECT * FROM Employe  
WHERE salaire >= 1400  
AND salaire >= 3000 ;
```

Équivalent à:

```
SELECT * FROM Employe  
WHERE salaire >= ALL (1400, 3000);
```

## Valeur dans un intervalle : **BETWEEN**

---

WHERE population  $\geq$  50 **AND** population  $\leq$  60

Équivalent à:

WHERE population **BETWEEN** 50 **AND** 60

• **NOT BETWEEN**

# Conditions partielles (joker)

---

- % : un ou plusieurs caractères
  - WHERE nom LIKE '%med'
  - WHERE prenom LIKE '%rem%'
- \_ : exactement un caractère
  - WHERE nom LIKE 'B\_dr'
- **NOT LIKE**

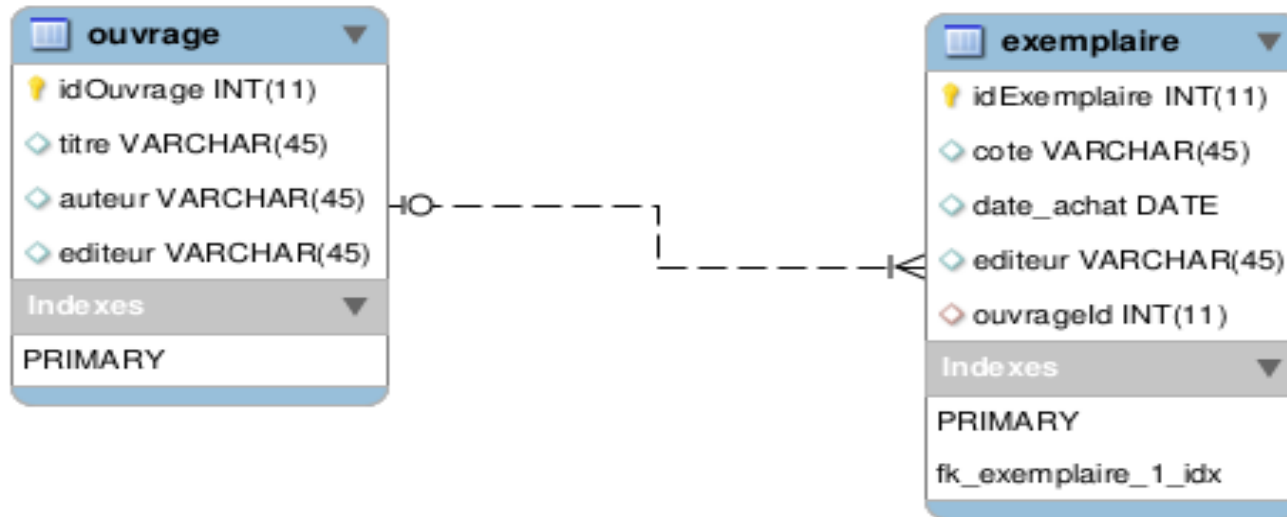
# Valeurs calculées

---

- `SELECT nom, population, surface, natalité`  
`FROM Pays`  
`WHERE (population * 1000 / surface) < 50`  
`AND (population * natalité / surface) > 0`
- `SELECT nom, (population * 1000 / surface )`  
`FROM Pays`

# Les jointures

- Principe :
  - Joindre plusieurs tables
  - On utilise les informations communes des tables



# Les jointures

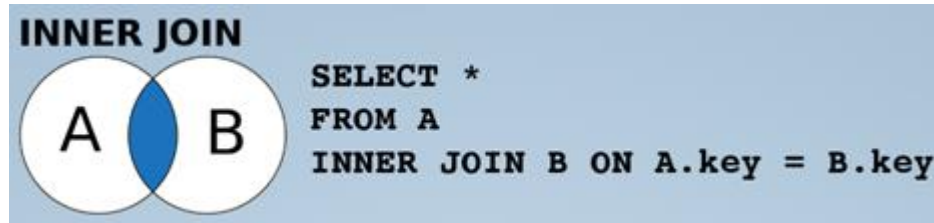
---

- Prenons pour exemple un **ouvrage** de **V. Hugo**
- Si l'on souhaite des informations sur **la cote** d'un **exemplaire** il faudrait le faire en 2 temps:
  - 1) je récupère l'id de l'ouvrage :  
**SELECT** id **FROM** ouvrage **where** auteur **LIKE** 'V. Hugo'
  - 2) Je récupère la ou les cote avec l'id récupéré  
**SELECT** cote **FROM** exemplaire **WHERE** ouvrageId = id\_récupéré

# Les jointures

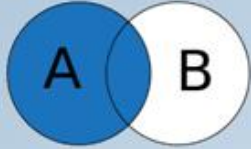
- On peut faire tout ça (et plus encore) en une seule requête
- C'est là que les jointures entrent en jeu:

```
SELECT exemple.cote  
FROM exemple  
INNER JOIN ouvrage  
    ON exemple.ouvrageId = ouvrage.idOuvrage  
WHERE ouvrage.auteur LIKE 'V. Hugo' ;
```



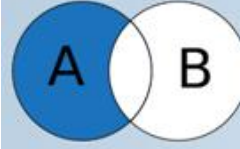
# Les jointures

## LEFT JOIN



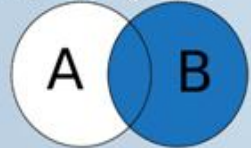
```
SELECT *  
FROM A  
LEFT JOIN B ON A.key = B.key
```

## LEFT JOIN (sans l'intersection de B)



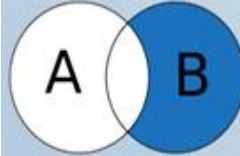
```
SELECT *  
FROM A  
LEFT JOIN B ON A.key = B.key  
WHERE B.key IS NULL
```

## RIGHT JOIN



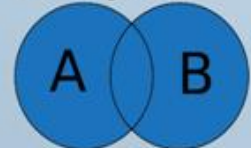
```
SELECT *  
FROM A  
RIGHT JOIN B ON A.key = B.key
```

## RIGHT JOIN (sans l'intersection de A)



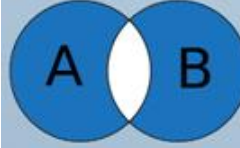
```
SELECT *  
FROM A  
RIGHT JOIN B ON A.key = B.key  
WHERE A.key IS NULL
```

## FULL JOIN



```
SELECT *  
FROM A  
FULL JOIN B ON A.key = B.key
```

## FULL JOIN (sans intersection)



```
SELECT *  
FROM A  
FULL JOIN B ON A.key = B.key  
WHERE A.key IS NULL  
OR B.key IS NULL
```



# Les jointures

## Ville

id_client	ville
1	Marseille
2	Paris
3	Lyon
7	Montpellier
8	Levallois

## Email

id_client	nom	prenom	email
1	Martin	Lucas	lucas.martin@outlook.fr
2	Clavier	Paul	NULL
3	Sauron	Benoit	NULL
4	Oron	Louis	louis.oron@yahoo.fr
5	Poiret	Michel	michel76@outlook.fr

### INNER JOIN

```
SELECT id_client, nom, prenom, ville
FROM ville
INNER JOIN email
ON ville.id_client = email.id_client
```

id_client	nom	prenom	email	ville
1	Martin	Lucas	lucas.martin@outlook.fr	Marseille
2	Clavier	Paul	NULL	Paris
3	Sauron	Benoit	NULL	Lyon

### LEFT JOIN

```
SELECT ID_CLIENT, NOM, PRENOM, VILLE
FROM EMAIL
LEFT JOIN VILLE
ON EMAIL.ID_CLIENT = VILLE.ID_CLIENT
```

id_client	nom	prenom	email	ville
1	Martin	Lucas	lucas.martin@outlook.fr	Marseille
2	Clavier	Paul	NULL	Paris
3	Sauron	Benoit	NULL	Lyon
4	Oron	Louis	louis.oron@yahoo.fr	NULL
5	Poiret	Michel	michel76@outlook.fr	NULL

### RIGHT JOIN

```
SELECT id_client, nom, prenom, ville
FROM email
RIGHT JOIN ville
ON email.id_client = ville.id_client
```

id_client	nom	prenom	email	ville
1	Martin	Lucas	lucas.martin@outlook.fr	Marseille
2	Clavier	Paul	NULL	Paris
3	Sauron	Benoit	NULL	Lyon
7	NULL	NULL	NULL	Montpellier
8	NULL	NULL	NULL	Levallois

# SQL

---

## Requêtes avec blocs emboîtés

# BD exemple

---

- **Produit**(np,nomp,couleur,poids,prix) *les produits*
- **Usine**(nu,nomu,ville,pays) *les usines*
- **Fournisseur**(nf,nomf,type,ville,pays) *les fournisseurs*
- **Livraison**(np,nu,nf,quantité) *les livraisons*
  - np référence *Produit*.np
  - nu référence *Usine*.nu
  - nf référence *Fournisseur*.nf

# Jointure par blocs emboîtés

*Requête: Nom et couleur des produits livrés par le fournisseur 1*

- Solution 1 : la jointure déclarative  
SELECT **nomp**, **couleur** FROM **Produit**, **Livraison**  
WHERE (**Livraison**.**np** = **Produit**.**np**) AND **nf** = 1 ;
- Solution 2 : la jointure procédurale (emboîtement)

***Nom et couleur des produits livrés par le fournisseur 1***

SELECT **nomp**, **couleur** FROM **Produit**  
WHERE **np** **IN**

(SELECT **np** FROM **Livraison** WHERE **nf** = 1) ;

***Numéros de produits livrés par le fournisseur 1***

# Jointure par blocs emboîtés

- `SELECT nomp, couleur FROM Produit  
WHERE np IN  
    ( SELECT np FROM Livraison  
      WHERE nf = 1) ;`

- **IN** compare chaque valeur de `np` avec l'ensemble (ou multi-ensemble) de valeurs retournés par la sous-requête

- **IN** peut aussi comparer un tuple de valeurs:

```
SELECT nu FROM Usine  
WHERE (ville, pays)  
      IN (SELECT ville, pays FROM Fournisseur);
```

# Composition de conditions

*Requête: Nom des fournisseurs qui approvisionnent une usine de Londres ou de Paris en un produit rouge*

```
SELECT nomf
FROM Livraison, Produit, Fournisseur, Usine
WHERE
    couleur = 'rouge'
    AND Livraison.np = Produit.np
    AND Livraison.nf = Fournisseur.nf
    AND Livraison.nu = Usine.nu
    AND (Usine.ville = 'Londres' OR Usine.ville = 'Paris');
```

# Composition de conditions

Requête: **Nom** des fournisseurs qui approvisionnent une usine de **Londres** ou de **Paris** en un produit **rouge**

```
SELECT nomf FROM Fournisseur  
WHERE nf IN
```

```
(SELECT nf FROM Livraison
```

```
  WHERE np IN (SELECT np FROM Produit  
                WHERE couleur = 'rouge')
```

```
  AND nu IN
```

```
(SELECT nu FROM Usine  
  WHERE ville = 'Londres' OR ville = 'Paris')
```

```
);
```

# Quantificateur ALL

*Requête: Numéros des fournisseurs qui ne fournissent que des produits rouges*

```
SELECT nf FROM Fournisseur  
WHERE 'rouge' = ALL  
  (SELECT couleur FROM Produit  
   WHERE np IN  
     (SELECT np FROM Livraison  
      WHERE Livraison.nf = Fournisseur.nf ) ) ;
```



- La requête imbriquée est ré-évaluée pour chaque tuple de la requête (ici pour chaque *nf*)
- **ALL**: tous les éléments de l'ensemble doivent vérifier la condition



# Condition sur des ensembles : EXISTS

- Test si l'ensemble n'est pas vide ( $E \neq \emptyset$ )

Exemple : *Noms des fournisseurs qui fournissent au moins un produit rouge*

SELECT nomf

FROM Fournisseur

WHERE EXISTS

( SELECT \*

FROM Livraison, Produit

WHERE Livraison.nf = Fournisseur.nf

AND Livraison.np = Produit.np

AND Produit.couleur = 'rouge' );

*ce fournisseur*

*Le produit fourni  
est rouge*

# Blocs emboîtés - récapitulatif

---

SELECT ...

FROM ...

WHERE ...

attr **IN** requête

attr **NOT IN** requête

attr opérateur **ALL** requête

**EXISTS** requête

**NOT EXISTS** requête

# SQL

---

## Traitement des résultat

# Fonctions sur les colonnes

---

- **Attributs calculés**

- Exemple : `SELECT nom, population*1000/surface FROM Pays`

- **Opérateurs sur attributs numériques**

- **SUM**: somme des valeurs des tuples sélectionnés
- **AVG**: moyenne

- **Opérateurs sur tous types d'attributs**

- **MIN**: minimum
- **MAX**: maximum
- **COUNT**: nombre de tuples sélectionnés

Opérateurs  
d'agrégation

# Opérateurs d'agrégation

pays

Nom	Capitale	Population	Surface	Continent
Irlande	Dublin	5	70	Europe
Autriche	Vienne	10	83	Europe
UK	Londres	50	244	Europe
Suisse	Berne	7	41	Europe
USA	Washington	350	441	Amérique

```
SELECT MIN(population), MAX(population), AVG(population),  
SUM(surface), COUNT(*)  
FROM Pays WHERE continent = 'Europe'
```

Donne le résultat :

MIN(population)	MAX(population)	AVG(population)	SUM(surface)	COUNT(*)
5	50	18	438	4

# DISTINCT

pays

Nom	Capitale	Population	Surface	Continent
Irlande	Dublin	5	70	Europe
Autriche	Vienne	10	83	Europe
UK	Londres	50	244	Europe
Suisse	Berne	7	41	Europe
USA	Washington	350	441	Amérique

Suppression des doubles

```
SELECT DISTINCT continent  
FROM Pays
```

Donne le résultat :

**Continent**

Europe

Amérique

# ORDER BY

---

## Tri des tuples du résultat

```
SELECT continent, nom, population
FROM Pays
WHERE surface > 60
ORDER BY continent, nom ASC
```

2 possibilités : **ASC** / **DESC**

Continent	Nom	Population
Amérique	USA	350
Europe	Autriche	10
Europe	Irlande	5
Europe	UK	50

# GROUP BY

Partition de l'ensemble des tuples en groupes homogènes:

```
SELECT continent, MIN(population), MAX(population),AVG(population),  
SUM(surface), COUNT(*)  
FROM Pays GROUP BY continent ;
```

Continent	MIN(population)	MAX(population)	AVG(population)	SUM(surface)	COUNT(*)
Europe	5	50	18	438	4
Amérique	350	350	350	441	1

**A noter :** cette commande doit toujours s'utiliser après la commande **WHERE** et avant la commande **HAVING**.



# HAVING

## Conditions sur les fonctions d'agrégation

- Il n'est pas possible d'utiliser la clause WHERE pour faire des conditions sur une fonction d'agrégation.
- Donc, si l'on veut afficher les pays dont on possède plus de 3 individus, **la requête suivante ne fonctionnera pas.**

```
SELECT continent, COUNT(*)  
FROM Pays  
WHERE COUNT(*) > 3  
GROUP BY continent ;
```

- Il faut utiliser HAVING qui se place juste après le GROUP BY

```
SELECT continent, COUNT(*)  
FROM Pays  
GROUP BY continent  
HAVING COUNT(*) > 3;
```

# Renommage des attributs : AS

```
SELECT MIN(population) AS min_pop,  
       MAX(population) AS max_pop,  
       AVG(population) AS avg_pop,  
       SUM(surface) AS sum_surface,  
       COUNT(*) AS count  
FROM Pays  
WHERE continent = 'Europe' ;
```

min_pop	max_pop	avg_pop	sum_surface	count
5	50	18	438	4

# Opérateurs ensemblistes en SQL

## 1. UNION

Syntaxe:

```
requête_SELECT  
UNION [ ALL] requête_SELECT  
[ UNION [ ALL] requête_SELECT ....]
```

L'opérateur UNION supprime les données redondantes par défaut, sauf si l'option ALL est explicité.

**Exemple:** lister tous les clients appartenant aux tables CLIENT et CLIENT\_CASA.

```
SELECT idclient , nom, tel  
FROM Client  
UNION  
SELECT idclient , nom, tel  
FROM Client_CASA;
```

# Opérateurs ensemblistes en SQL

## 1. UNION ALL

Syntaxe:

```
SELECT * FROM table1  
UNION ALL  
SELECT * FROM table2
```

- cette commande permet d'inclure tous les enregistrements, même **les doublons**.
- Ainsi, si un même enregistrement est présents normalement dans les résultats des 2 requêtes concaténées, alors l'union des 2 avec **UNION ALL** retournera 2 fois ce même résultat.
- **Exemple:**

```
SELECT * FROM magasin1_client  
UNION ALL  
SELECT * FROM magasin2_client
```

# Opérateurs ensemblistes en SQL

## 1. UNION ALL

magasin1\_client

prenom	nom	ville	date_naissance	total_achat
Léon	Dupuis	Paris	1983-03-06	135
Marie	Bernard	Paris	1993-07-03	75
Sophie	Dupond	Marseille	1986-02-22	27
Marcel	Martin	Paris	1976-11-24	39

magasin2\_client

prenom	nom	ville	date_naissance	total_achat
Marion	Leroy	Lyon	1982-10-27	285
Paul	Moreau	Lyon	1976-04-19	133
Marie	Bernard	Paris	1993-07-03	75
Marcel	Martin	Paris	1976-11-24	39

Résultat:

prenom	nom	ville	date_naissance	total_achat
Léon	Dupuis	Paris	1983-03-06	135
Marie	Bernard	Paris	1993-07-03	75
Sophie	Dupond	Marseille	1986-02-22	27
Marcel	Martin	Paris	1976-11-24	39
Marion	Leroy	Lyon	1982-10-27	285
Paul	Moreau	Lyon	1976-04-19	133
Marie	Bernard	Paris	1993-07-03	75
Marcel	Martin	Paris	1976-11-24	39

# Opérateurs ensemblistes en SQL

## 2. INTERSECTION

Syntaxe:

```
requête_SELECT  
INTERSECT  
requête_SELECT
```

**Exemple:** afficher les livres dont le prix est supérieur à 500 et qui sont toujours empruntés.

```
SELECT noliv  
FROM livre l  
WHERE l.prix>500  
INTERSECT  
SELECT noliv  
FROM emprunt e  
WHERE e.retour IS NULL
```

# Opérateurs ensemblistes en SQL

---

## 3. DIFFERENCE

Syntaxe:

```
requête_SELECT  
MINUS  
requête_SELECT
```

**Exemple:** trouver les livres non encore empruntés

```
SELECT noliv  
FROM livre  
MINUS  
SELECT noliv  
FROM emprunt  
WHERE retour IS NULL
```

# La Requête TRUNCATE en SQL

---

- ❑ En SQL, la commande **TRUNCATE** permet de supprimer toutes les données d'une table sans supprimer la table en elle-même.
- ❑ Cette instruction diffère de la commande **DROP** qui a pour but de supprimer les données ainsi que la table qui les contient.
- ✓ **A noter :** l'instruction **TRUNCATE** est semblable à l'instruction **DELETE** sans utilisation de WHERE. Parmi les petite différences TRUNCATE est toutefois plus rapide et utilise moins de ressource.
- ✓ Ces gains en performance se justifie notamment parce que la requête n'indiquera pas le nombre d'enregistrement supprimés et qu'il n'y aura pas d'enregistrement des modifications dans le journal.



# La Requête TRUNCATE en SQL

---

## Syntaxe:

- ❑ Cette instruction s'utilise dans une requête SQL semblable à celle-ci :

```
TRUNCATE TABLE `table`
```

- ❑ Dans cet exemple, les données de la table “table” seront perdues une fois cette requête exécutée.

# La Requête TRUNCATE en SQL

## Exemple

Table “fourniture” :

id	nom	date_ajout
1	Ordinateur	2023-04-05
2	Chaise	2023-04-14
3	Bureau	2023-07-18
4	Lampe	2023-09-27

- ❑ Il est possible de supprimer toutes les données de cette table en utilisant la requête:

```
TRUNCATE TABLE `fourniture`
```

- ❑ Une fois la requête exécutée, la table ne contiendra plus aucun enregistrement. En d'autres mots, toutes les lignes du tableau présenté ci-dessus auront été supprimées.

# La Requête TRUNCATE en SQL

---

## Différence entre TRUNCATE et DELETE

- La commande **TRUNCATE** s'avère être similaire à la commande **DELETE**, lorsqu'elle est utilisée de la façon suivante :

**DELETE** FROM `fourniture`

- Il y a toutefois une différence notable : la commande **TRUNCATE** va **ré-initialiser la valeur de l'auto-incrément**, s'il y en a un.
- Dès lors, il faut potentiellement noter la valeur de l'auto-incrément avant de faire un **TRUNCATE**, surtout s'il faut ré-indiquer l'ancienne valeur après avoir écrasé toutes les données.

# SQL

---

## Les Vues

# Les Vues en SQL

---

- ❑ **Une vue** est une construction logique (**table virtuelle**) faites à partir de tables existantes (tables de base);
- ❑ elle ne contient aucune données en soit, **elle n'est qu'une représentation indirecte de données contenues dans d'autres tables;**
- ❑ les vues sont très utilisées pour simplifier et optimiser l'usage de structures intermédiaires souvent sollicitées;
- ❑ elles sont constamment mises à jour par le SGBD.

# Les Vues en SQL

---

Création par:

**CREATE VIEW** *nom* (*renommage facultatif des colonnes*) **AS**  
*requête SELECT*

Syntaxe générale:

**CREATE** [OR REPLACE]  
[FORCE | NOFORCE] **VIEW**  
*nom-de-vue [(attr1, ..., attrn)]*  
**AS** *requête*  
[WITH CHECK OPTION  
[CONSTRAINT *nom-contrainte*]]  
[WITH READ ONLY]

# Les Vues en SQL

---

- ❑ L'instruction ***OR REPLACE*** crée à nouveau la vue si elle existe déjà.
- ❑ Les clauses ***FORCE*** et ***NOFORCE*** indiquent respectivement que :
  - la vue est créée sans se soucier de l'existence des tables qu'elle référence ou des privilèges adéquats sur les tables,
  - la vue est créée seulement si les tables existent et si les permissions requises sont données.
- ❑ La clause ***WITH CHECK OPTION*** limite les insertions et les mises à jour exécutées par l'intermédiaire de la vue.
- ❑ La clause ***CONSTRAINT*** est un nom optionnel donné à la contrainte ***WITH CHECK OPTION***.

# Les Vues en SQL

---

- ❑ **Les vues peuvent être créées à partir d'autres vues.**  
Pour cela il suffit de référencer les vues dans la clause *FROM* de l'instruction *select*.

```
CREATE VIEW nom_vue  
AS  
SELECT * FROM nom_vue2;
```



# Les Vues en SQL

## Exemples:

```
CREATE VIEW rbati
AS SELECT enum, ename
FROM employee
WHERE address="Rabat"
```

```
select * from rbati;
```

enum	ename
E5	Amina
E1	Ali

Avec renommage des attributs:

```
CREATE VIEW rbati (numero, nom)
AS SELECT enum, ename
FROM employee
WHERE address="Rabat"
```

```
select * from rbati;
```

numero	nom
E5	Amina
E1	Ali

# Les Vues en SQL

---

Interrogation avec:

```
SELECT ..  
FROM nom_de_la_vue  
WHERE ..
```

→ comme pour les tables de base.

Exemple:

```
select nom  
from rbatl  
where numero = 'E1';
```

Résultat:

nom
Ali

# Les Vues en SQL

---

- ❑ Suppression d'une vue

**DROP VIEW** *nom-de-vue*

→ La suppression d'une vue n'entraîne pas la suppression des données

- ❑ Renommer une vue

**RENAME** *ancien-nom* **TO** *nouveau-nom*

# Les Vues en SQL

---

## Exemples 1:

- **CREATE VIEW** `vue_personnel`  
**AS SELECT** sexe, nom, prenom, adresse, cp, ville  
**FROM** `tbl_personnel`  
**WHERE** service = 'commercial'
- **CREATE VIEW** `vue_pers_serv`  
**AS SELECT** v.sexe, v.nom, v.prenom, t.poste, t.service  
**FROM** `vue_personnel` v, `tbl_service` t  
**WHERE** code\_service = 'SC02354L' **GROUP BY** v.nom;

# Les Vues en SQL

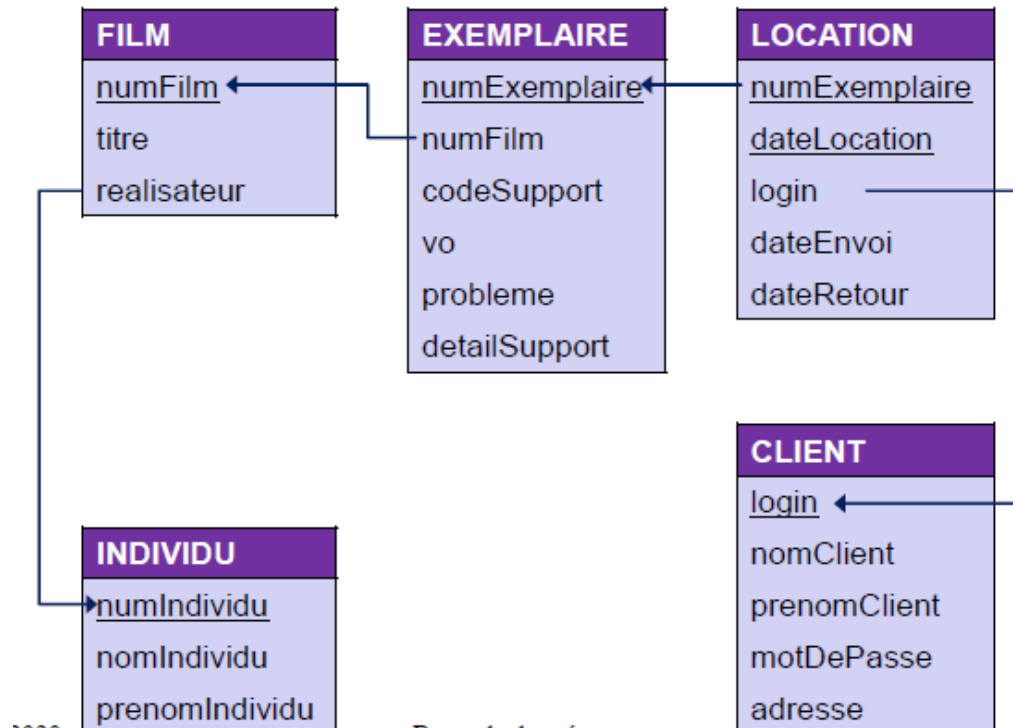
---

## Utilisation d'une vue :

- **UPDATE** vue\_pers\_serv  
**SET** sexe = 'masculin' **WHERE** nom = 'Frédérique' **AND** prenom = 'Jean'
- **SELECT** v1.nom, v1.prenom, v2.service  
**FROM** vue\_personnel AS v1, vue\_services AS v2  
**WHERE** v1.id\_service = v2.id\_service
- **CREATE View** tbl\_employes  
**AS SELECT** v1.id\_service, v2.service, v1.nom, v1.prenom, v1.adresse  
**FROM** vue\_personnel AS v1, vue\_services AS v2  
**WHERE** v1.id\_service = v2.id\_service

# Les Vues en SQL

## Exemple 2:



# Les Vues en SQL

---

## Création de la vue:

### ❑ **CREATE OR REPLACE VIEW**

**exemplairePlus** (num, titre, real, support)

**AS**

**SELECT** numExemplaire, titre, nomIndividu, codesupport

**FROM** **Exemplaire** E, **Film** F, **Individu**

**WHERE** E.numFilm = F.numFilm

**AND** realisateur = numIndividu

**AND** probleme IS NULL;

# Les Vues en SQL

---

## Sélection:

❑ **SELECT** num, titre, dateLocation, login  
**FROM** **exemplairePlus**, **Location**  
**WHERE** num = numExemplaire  
**AND** real = 'Scorces'  
**AND** dateRetour IS NULL;



# Les Vues en SQL

---

## Insertion:

❑ **INSERT INTO** **exemplairePlus** (num, support)  
**VALUES** (150346, 'DVD');

## Suppression:

❑ **DROP VIEW** **exemplairePlus**;

# Les Vues en SQL

---

## Contraintes d'intégrité (CHECK OPTION) :

❑ **CREATE VIEW** anciensExemplaires  
    **AS**   **SELECT** \* **FROM** Exemple  
          **WHERE** numExemple < 2000  
          **WITH CHECK OPTION;**

❑ **UPDATE** anciensExemplaires  
    **SET** numExemple = 3812  
    **WHERE** numExemple = 1318;

❑ **Sans** 'WITH CHECK OPTION', c'est possible.

❑ **Avec** 'WITH CHECK OPTION', c'est impossible.

# SQL

---

## Langage de Contrôle de Données

# Le Langage de Contrôle de Données

---

## Langage de contrôle de données (LCD):

- ❑ Les instructions de contrôle des données donnent à l'administrateur de base de données le pouvoir **de contrôler la sécurité** de la base.
- ❑ **Plusieurs personnes travaillent simultanément sur une base de données.**
- ❑ Certaines peuvent par exemple avoir besoin de **modifier ou supprimer des données** dans la table, pendant que d'autres ont seulement un besoin de **consultation** de données.
- ❑ Ainsi, il est possible de définir des **permissions** pour chaque personne en leur octroyant un **mot de passe**.

# Le Langage de Contrôle de Données

---

- ❑ Cette tâche est le rôle de l'**administrateur** de la base de données (en anglais *DBA, DataBase Administrator*).
- ❑ Il doit dans un premier temps **définir les besoins** de chacun, puis les appliquer à la base de données sous forme de **permissions**.
- ❑ Le LCD est composé de 4 commandes SQL :
  - ❑ **GRANT**
  - ❑ **REVOKE**
  - ❑ **COMMIT**
  - ❑ **ROLLBACK**

# Le Langage de Contrôle de Données

---

- ❑ **GRANT** et **REVOKE** sont utilisées pour exercer un contrôle sur l'accès aux données.
- ❑ **COMMIT** et **ROLLBACK** sont utilisées pour préserver l'intégrité des données.
- Pour utiliser une base de données, **l'utilisateur** doit passer par une procédure de **connexion**.
- Lors de cette procédure, il doit saisir un **login** (par exemple PDupont) et un **mot de passe** (par exemple xyz2V12).
- Ce **login** permet d'identifier chaque utilisateur et sert de repère pour lui **accorder** (ou lui enlever) **des droits** de manipulation de la base.

# Le Langage de Contrôle de Données

## Création d'un utilisateur de la base de données:

### Syntaxe sous MySQL :

**CREATE USER** username **IDENTIFIED BY** password;

### Paramètres:

- **username**: un nom d'utilisateur (login)
- **password**: un mot de passe

### Exemple:

```
CREATE USER 'nouveau_utilisateur'@'localhost' IDENTIFIED BY 'mot_de_passe';
```

- **localhost**: signifie que l'utilisateur va se connecter à partir de la machine locale. Sinon, il faut préciser **l'adresse IP** de l'hôte concerné, ou bien utiliser **%** pour qu'il se connecte de n'importe quel hôte
- ***DROP USER 'username'@'localhost' ; permet de supprimer un utilisateur***

# Le Langage de Contrôle de Données

## 1. La commande GRANT

- ❑ La commande **GRANT** permet **d'autoriser** un accès aux données de la base soit total, soit à des degrés limités.
- ❑ Ainsi, il est possible, par exemple, d'autoriser la consultation de certaines tables sans possibilité de les modifier.

### ❑ Syntaxe de la commande GRANT en SQL

```
GRANT ALL PRIVILEGES | accès_spécifique  
ON nom_table | nom_vue  
TO PUBLIC nom_authorized |  
[WITH GRANT OPTION];
```



# Le Langage de Contrôle de Données

---

## 1. La commande GRANT

Avec la convention suivante :

- Il est possible de donner tous les types de droits d'accès (**consultation, modification, suppression,...**) avec **ALL PRIVILEGES** ou d'accorder des privilèges spécifiques (accès\_spécifique).
- **nom\_autorisé** (ou **login**) : nom donné par l'utilisateur lors de sa connexion à la base de données (créé par l'administrateur)
- **PUBLIC** : tout le monde reçoit le privilège accordé (accès\_spécifique) ou tous les privilèges (ALL PRIVILEGES).
- **WITH GRANT OPTION** : celui qui reçoit le privilège peut lui-même l'accorder à un autre (opération dangereuse, à éviter).

# Le Langage de Contrôle de Données

## 1. La commande GRANT

Concernant les types de droits d'accès, si des tables de la base sont identifiées à l'aide du nom du créateur de la base, **cet utilisateur** peut (si on lui a accordé le privilège requis) **autoriser** l'accès à ses tables pour **d'autres utilisateurs**.

```
GRANT ALL PRIVILEGES  
ON VOITURE  
TO Martin  
WITH GRANT OPTION;
```

Cette instruction SQL permet à **l'administrateur** de la BD d'attribuer à **Martin tous les droits sur la table VOITURE**, il lui accorde également l'autorisation de **transférer** ces privilèges.

# Le Langage de Contrôle de Données

---

## Les droits d'accès:

- ☐ La gestion des droits d'accès aux tables est décentralisée : il n'existe pas d'administrateur global attribuant des droits.
- ☐ Seul le propriétaire (créateur) d'une table peut attribuer des droits sur celle-ci.

Les principaux droits d'accès spécifiques sont :

- ☐ sélection (**SELECT**)
- ☐ insertion (**INSERT**)
- ☐ suppression (**DELETE**)
- ☐ mise à jour (**UPDATE**)
- ☐ indexation (**INDEX**)
- ☐ référencer la table dans une contrainte (**REFERENCES**)

# Le Langage de Contrôle de Données

## Les droits d'accès:

- ❑ Le propriétaire peut ensuite passer ses droits sélectivement à d'autres utilisateurs ou à tout le monde (PUBLIC).
- ❑ Un droit peut être passé avec le droit de le transmettre (WITH GRANT OPTION) ou non.
- ❑ L'ensemble des droits d'accès (ALL PRIVILEGES) inclut les droits d'administration (changement de schéma et destruction de la relation).

```
GRANT SELECT, UPDATE  
ON VOITURE, ACHAT  
TO Smith;
```

- ❑ Cette commande permet de passer des droits de **consultation et de mise à jour** de la table **VOITURE** et de la table **ACHAT** à l'utilisateur **Smith**.

# Le Langage de Contrôle de Données

## Les droits d'accès:

```
GRANT ALL PRIVILEGES (Immatriculation, Prix)
ON VOITURE
TO Smith, Vandenbrouck, Dubois;
```

- ❑ Cette commande permet de passer l'ensemble des droits d'accès aux utilisateurs Smith, Vandenbrouck et Dubois uniquement sur les colonnes Immatriculation et Prix de la table VOITURE.

```
GRANT INSERT
ON PERSONNE
TO PUBLIC;
```

- ❑ Cette commande attribue le droit d'insérer de nouveaux enregistrements dans la table PERSONNE à tous ceux qui, dans la société, disposent d'une identification pour se connecter sur la BD.

# Le Langage de Contrôle de Données

---

## 2. La commande REVOKE

- ❑ La commande **REVOKE** permet de **retirer** l'accès, c'est la commande inverse de GRANT.

### ❑ Syntaxe de la commande REVOKE en SQL

```
REVOKE ALL PRIVILEGES | accès_spécifique  
ON nom_table | nom_vue  
FROM nom_utilisateur | PUBLIC;
```

# Le Langage de Contrôle de Données

## 2. La commande REVOKE

### Exemple

```
REVOKE SELECT, UPDATE  
ON VOITURE, ACHAT  
FROM Smith;
```

- ❑ Cette commande supprime les droits de consultation et de mise à jour de la table VOITURE et de la table ACHAT qui avait été accordés à l'utilisateur Smith.

```
REVOKE ALL PRIVILEGES  
ON VOITURE  
FROM Martin;
```

- ❑ Cette commande retire tous les privilèges accordés sur la table **VOITURE** à **Martin**.

# Le Langage de Contrôle de Données

---

## 3. La commande COMMIT

- La commande **COMMIT** permet à l'utilisateur de fixer le moment où les modifications en cours affecteront la base de données. Dans ce cadre, on utilise le concept de **transaction**.
- La transaction est une suite d'opérations telle que chaque opération de cette suite est nécessaire pour atteindre un résultat unitaire.
- C'est la raison pour laquelle SQL propose à l'utilisateur de n'enregistrer les modifications dans la base qu'au moment où la transaction est achevée grâce à la commande **COMMIT**.
- Avant l'exécution de l'instruction COMMIT, il est possible de restaurer la base par **ROLLBACK**, c'est-à-dire d'éliminer les modifications récentes.
- **Après l'enregistrement définitif d'une transaction par COMMIT, il n'est plus possible de restaurer l'état antérieur par ROLLBACK.**
- S'il apparaît après coup qu'une transaction doit être modifiée ou corrigée, on ne pourra effectuer cette modification qu'au moyen d'une **autre instruction SQL** comme UPDATE ou DELETE.



# Le Langage de Contrôle de Données

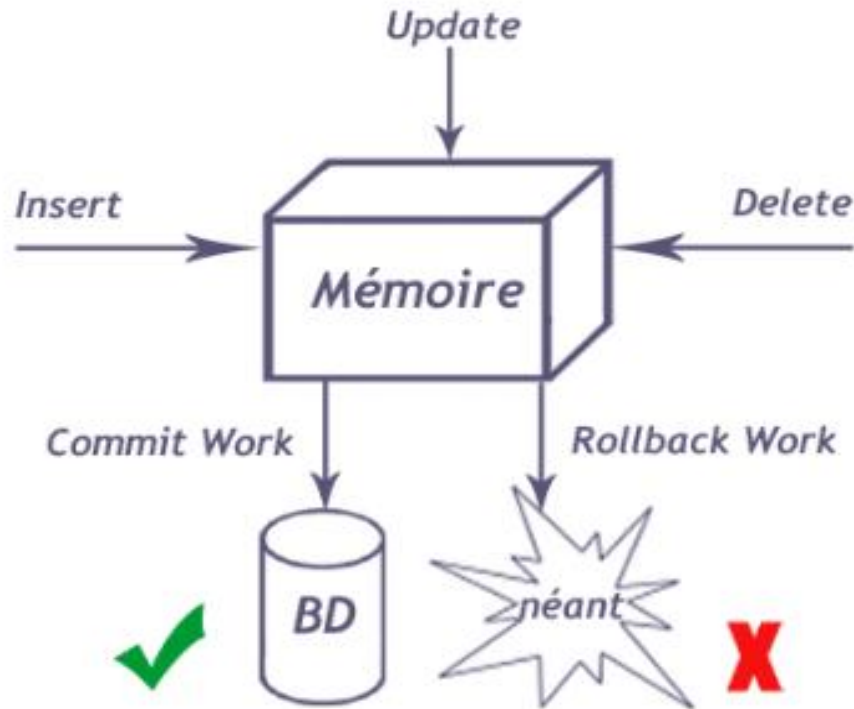
---

## 4. La commande ROLLBACK

- La commande ROLLBACK permet à l'utilisateur de **ne pas valider** les dernières modifications en cours dans la base de données.
- Par exemple, **si** au cours du déroulement d'une transaction, **l'utilisateur fait une erreur** ou si, pour une certaine raison, une transaction ne peut pas être achevée, l'utilisateur peut supprimer les modifications afin d'éviter des incohérences dans la base grâce à la commande **ROLLBACK**.
- Cette commande élimine tous les changements depuis la dernière validation.
- Dans le cas d'une défaillance du système, l'intégrité de la base peut être préservée par une option **ROLLBACK** automatique qui élimine les transactions inachevées et empêche donc qu'elles soient introduites dans la base.

# Le Langage de Contrôle de Données

## Terminaison d'une transaction



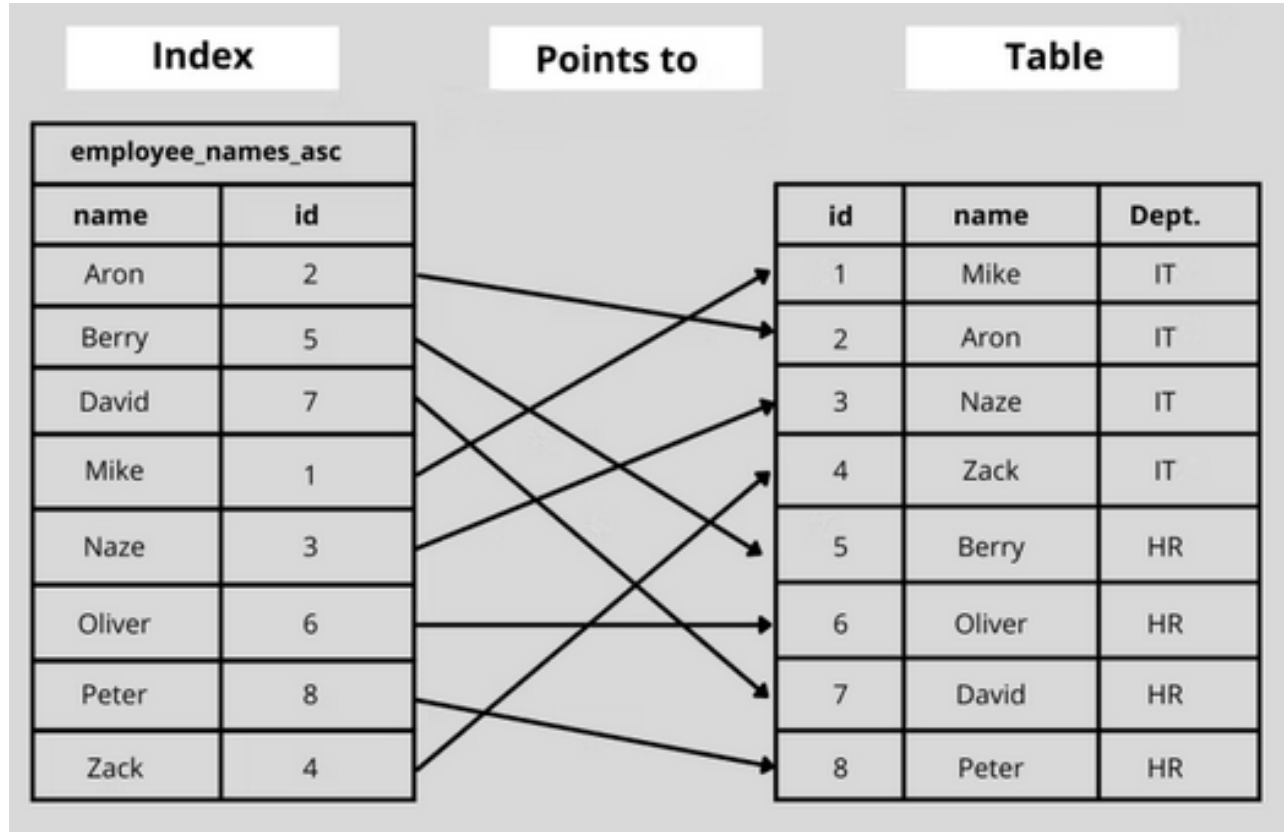
# Les INDEX en SQL

---

- ❑ En SQL, **les index** sont des ressources très utiles qui permettent **d'accéder plus rapidement** aux données.
- ❑ Dans un livre, **un index permet de retrouver toutes les pages liées à un mot important spécifique.**
- ❑ **Un index**, dans **une base de données** se base sur le même principe qu'un index dans un livre.
- ❑ Avec **un index** placé sur une ou plusieurs **colonnes** le système d'une base de données peut rechercher les données d'abord sur l'index et s'il trouve ce qu'il cherche il saura plus rapidement où se trouve les enregistrements concernés.

# Les INDEX en SQL

## Exemple 1:



# Les INDEX en SQL

## Exemple 3:

Id	Id	Espèce	Sexe	Date de naissance	Nom	Commentaires	Date de naissance
1	2	chat	NULL	2010-03-24 02:23:00	Roucky	NULL	2008-09-11 15:38:00
2	1	chien	male	2010-04-05 13:43:00	Rox	Mordille beaucoup	2008-12-06 05:18:00
3	3	chat	femelle	2010-09-13 15:02:00	Schtroumpfette	NULL	2009-06-13 08:17:00
4	6	tortue	femelle	2009-06-13 08:17:00	Bobosse	Carapace bizarre	2009-08-03 05:12:00
5	9	tortue	NULL	2010-08-23 05:18:00	NULL	NULL	2010-03-24 02:23:00
6	4	tortue	femelle	2009-08-03 05:12:00	NULL	NULL	2010-04-05 13:43:00
7	7	chien	femelle	2008-12-06 05:18:00	Caroline	NULL	2010-08-23 05:18:00
8	8	chat	male	2008-09-11 15:38:00	Baghera	NULL	2010-09-13 15:02:00
9	5	chat	NULL	2010-10-03 16:44:00	Choupi	Né sans oreille gauche	2010-10-03 16:44:00

# Les INDEX en SQL

---

## Syntaxe: Créer un index ordinaire

- ❑ La syntaxe basique pour créer un index est la suivante :

**CREATE INDEX** `index\_nom` **ON** `table`;

- ❑ Il est également possible de créer un index sur une seule colonne en précisant la colonne sur laquelle doit s'appliquer l'index :

**CREATE INDEX** `index\_nom` **ON** `table` (`colonne1`);

- L'exemple ci-dessus va donc insérer l'index intitulé "**index\_nom**" sur la table nommée "**table**" uniquement sur la colonne "**colonne1**".
- ❑ Pour insérer un index sur plusieurs colonnes il est possible d'utiliser la syntaxe suivante:

**CREATE INDEX** `index\_nom` **ON** `table` (`colonne1`, `colonne2`);

- L'exemple ci-dessus permet d'insérer un index les 2 colonnes : **colonne1** et **colonne2**.

# Les INDEX en SQL

---

## Créer un index unique:

- ❑ Un index unique permet de spécifier qu'une ou plusieurs colonnes doivent contenir des valeurs uniques à chaque enregistrement.
- ❑ Le système de base de données retournera une erreur si une requête tente d'insérer des données qui feront doublons sur la clé d'unicité.

## ❑ **Syntaxe:**

```
CREATE UNIQUE INDEX `index_nom` ON `table` (`colonne1`);
```

- ❑ créer un index d'unicité sur 2 colonnes:

```
CREATE UNIQUE INDEX `index_nom` ON `table` (`colonne1`, `colonne2`);
```

# Les INDEX en SQL

---

## Convention de nommage:

- ❑ Il n'existe pas de convention de nommage spécifique sur le nom des index, juste des suggestions de quelques développeurs et administrateurs de bases de données.
  
- ❑ Voici une liste de suggestions de préfixes à utiliser pour nommer un index :
  - Préfixe “**PK\_**” pour **P**rietary **K**ey (traduction : clé primaire)
  - Préfixe “**FK\_**” pour **F**oreign **K**ey (traduction : clé étrangère)
  - Préfixe “**UK\_**” pour **U**nique **K**ey (traduction : clé unique)
  - Préfixe “**UX\_**” pour **U**nique **I**ndex (traduction : index unique)
  - Préfixe “**IX\_**” pour chaque autre **I**ndex



# Bases de données

---

**Fin du cours**