BASES DE DONNEES AVANCEES

Dr BATOUMA NARKOY

Chapitre 9

Langage SQL

Sommaire

Introduction

Langage de définition des données;

- Création des tables, colonnes des tables
- Les contraintes sur les tables et les colonnes
- Les assertions
- Modification de la structure des tables et des colonnes;

Langage de Manipulation des données;

- Insertion des données;
- Modifications des données
- Suppression des données

Sommaire

Langage d'interrogation des données

- Operations relationnelles de base;
- Opération ensemblistes
- Requêtes avec agrégations;
- Les sous-requêtes

Sécurité des BDs

- sécurité et intégrité
- contrôle d'accès
- Privilèges sur les objets
- Droits associés aux rôles

Introduction

- Le langage SQL est la réalisation pratique des concepts de l'approche relationnelle. C'est une évolution du langage SEQUEL, lui-même dérivé du langage de recherche SQUARE.

- La première version du langage est normalisé par l'ISO depuis 1987. La deuxième version du langage, SQL2, a été adoptée en 1992. La troisième version du langage, SQL3, normalisée en 1999, ajoute essentiellement les fonctionnalités liées à l'utilisation de l'approche objet.

- La quasi-totalité des SGBD disposent d'une interface SQL même si aucun ne couvre l'ensemble de la norme. La norme SQL prévoit trois niveaux de conformité : le niveau d'entrée, le niveau intermédiaire et le niveau complet.

Introduction(suite)

- Le langage SQL manipule l'objet fondamental de l'approche relationnelle : la relation
- SQL est un langage dit « non procédural » ou « déclaratif »
- Il ne dispose pas d'instructions de structuration, telles que des boucles.
- Les instructions SQL sont alors intégrées dans le langage *via* une interface spécifique. Les résultats de la requête SQL sont alors stockés dans des structures de données propres au langage employé (par exemple un tableau) afin de pouvoir les manipuler.
- Le langage de programmation qui intègre le langage SQL est alors appelé **langage hôte**
- Un ensemble d'instructions SQL se nomme une requête. Une requête SQL se termine toujours par le caractère « ; ».

SQL: Trois Langages

- **Langage de définition de données (LDD/DDL)**
  - création de relations : **CREATE TABLE**
  - modification de relations: **ALTER TABLE**
  - suppression de relations: **DROP TABLE**
  - vues, index …. : CREATE VIEW ...

- **Langage de manipulation de données (LMD /DML)**
  - insertion de tuples: **INSERT**
  - mise à jour des tuples: **UPDATE**
  - suppression de tuples: **DELETE**

- **Langage de requêtes (LMD/DML)**
  - **SELECT ……. FROM ……. WHERE …….**

Terminologie

- **Relation** –> table
- **Tuple** –> ligne
- **Attribut** –> colonne (column)
- **Identifiant** –> clé primaire (primary key)
  clé secondaire (unique)
- **Identifiant externe**
  –> clé externe (external key)

Langage de Définition des Données (LDD)

Le LDD est la partie du langage SQL qui permet de créer de façon déclarative les objets composant une BD*.Il permet notamment la définition des schémas, des relations, des contraintes d'intégrité, des vues.

Langage de définition des Données

- **CREATE TABLE** : créer une table.

- **CREATE VIEW** : créer une vue particulière sur les données à partir d'un SELECT.

  Table dérivée.

- **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

Syntaxe:

```
CREATE TABLE nom-table
{ ( nom-col type-col [DEFAULT valeur]
[ [CONSTRAINT] contrainte-col] )*
[ [CONSTRAINT] contrainte-table ]*
| AS requête-SQL };
```

Légende :

- {a | b} : a ou b
- [option]
- * : applicable autant de fois que souhaité
- mot en capitale : mot clé

Create Table

- CREATE TABLE nom_table

  { (nom-col type-col [DEFAULT val] [[CONSTRAINT] contrainte-col])*

  [[CONSTRAINT] contrainte-table]* | AS requête-SQL };

Exemples :

```
create table CLIENT ( NCLI     char(10),
                      NOM      char(32),
                      ADRESSE  char(60),
                      LOCALITE char(30),
                      CAT      char(2),
                      COMPTE   decimal(9,2))
```

- CREATE TABLE Doctorant

  (nom VARCHAR(20),

  prénom VARCHAR(15),

  année_insc DECIMAL(4) DEFAULT 2003) ;

- CREATE TABLE Doctorant

  AS SELECT nom, prénom, année_inscr

  FROM Etudiant WHERE statut='Doctorant' ;

Colonnes et leurs types

- **SMALLINT** : entiers signés courts (p. ex. 16 bits);
- **INTEGER (ou INT)** : entiers signés longs (p. ex. 32 bits);
- **NUMERIC(p,q)** : nombres décimaux de p chiffres dont q après le point
- **décimal**; si elle n'est pas mentionnée, la valeur de q est 0;
- **DECIMAL(p,q)** : nombres décimaux d'au moins p chiffres dont q après le point décimal; si elle n'est pas mentionnée, la valeur de q est 0;
- **FLOAT(p) (ou FLOAT)** : nombres en virgule flottante d'au moins p bits significatifs;
- **CHARACTER(p) (ou CHAR)** : chaîne de longueur fixe de p caractères;

Colonnes et leurs types

- **CHARACTER VARYING (ou VARCHAR(p))** : chaîne de longueur variable d'au plus p caractères;
- **BIT(p)** : chaînes de longueur fixe de p bits;
- **BIT VARYING** : chaînes de longueur variable d'au plus p bits;
- **DATE** : dates (année, mois et jour);
- **TIME** : instants (heure, minute, seconde, éventuellement 1000ème de seconde);
- **TIMESTAMP** : date + temps,
- **INTERVAL** : intervalle en années/mois/jours entre dates ou en heures/minutes/secondes entre instants.

Exemples de domaines de valeurs

create domain MONTANT decimal(9,2)
create domain MATRICULE char(10)
create domain LIBELLE char(32)

create table CLIENT (NCLI **MATRICULE**,
NOM **LIBELLE**,
ADRESSE char(60),
LOCALITE **LIBELLE**,
CAT char(2),
COMPTE **MONTANT**)

create domain MONTANT decimal(9,2) **default 0.0**
CAT char(2) **default 'AA'**

Les contraintes

- **contrainte-col : contrainte sur une colonne**
  - NOT NULL
  - PRIMARY KEY
  - UNIQUE
  - REFERENCES nom-table [(nom-col)] [action]
  - CHECK (condition)

- **contrainte-table : contraintes sur une table**
  - PRIMARY KEY (nom-col*)
  - UNIQUE (nom-col*)
  - FOREIGN KEY (nom-col*) REFERENCES nom-table [(nom-col*)] [action]
  - CHECK (condition)

Attribut obligatoire: Not Null

- Contrainte sur une colonne

- CREATE TABLE Pays

  (nom VARCHAR(20) NOT NULL ,

  capitale VARCHAR(20) NOT NULL ,

  surface INTEGER,

  …);

- Par défaut (c'est-à-dire si on ne spécifie rien) toute colonne est facultative. Le caractère obligatoire d'une colonne se déclarera par la clause *not null*

Clé primaire/secondaire

- **PRIMARY KEY (A1, A2, …)**
  - La clé primaire (si elle existe) .
  - choisir l'identifiant le plus efficace
  - attribut référencé par défaut dans les identifiants externes
  - pas de valeur nulle possible c-à-d NOT NULL automatiquement

- **UNIQUE (A1, A2, …)**
  - une clé secondaire (s'il en existe)
  - contrainte d'intégrité pour les autres identifiants
  - valeur nulle permise (sauf si NOT NULL)

Exemples

```
create table CLIENT ( NCLI      char(10),
                      NOM       char(32),
                      ADRESSE   char(60),
                      LOCALITE  char(30),
                      CAT       char(2),
                      COMPTE    decimal (9,2),
                      primary key (NCLI) )

create table DETAIL ( NCOM      char(12),
                      NPRO      char(15),
                      QCOM      decimal(8),
                      primary key (NCOM,NPRO) )


create table ASSURE ( NUM_AFFIL char(10),
                      NUM_IDENT char(15),
                      NOM       char(35),
                      primary key (NUM_AFFIL),
                      unique (NUM_IDENT) )
```

Exemples

- CREATE TABLE Pays
  (nom VARCHAR(20) PRIMARY KEY ,
  capitale VARCHAR(20) …)

- CREATE TABLE Employé
  (nom VARCHAR(30) ,
  prénom VARCHAR(30) ,
  adresse VARCHAR(60) , …
  CONSTRAINT Pk_emp PRIMARY KEY (nom, prénom))

- contrainte de colonne et de table

Exemple: Unique

- CREATE TABLE Etudiant
  (AVS CHAR(11) PRIMARY KEY ,
  N°Etudiant CHAR(6) UNIQUE ,
  nom VARCHAR(20) ,
  prénom VARCHAR(30) , …
  CONSTRAINT UNIQUE (nom, prénom))

- Contrainte de colonne et de table

- PRIMARY KEY et UNIQUE sont incompatibles

Contrainte référentielle: foreign key

- CREATE TABLE Etudiant (N°E …)

- CREATE TABLE Cours (NomCours …)

- CREATE TABLE Suit

  (N°Etud CHAR(9) ,

  NomC VARCHAR(25) ,

  PRIMARY KEY (N°Etud , NomC) ,

  FOREIGN KEY (N°Etud) REFERENCES Etudiant ,

  FOREIGN KEY (NomCours) REFERENCES Cours)

Contrainte référentielle: foreign key

- Les clés externes référencent par défaut la clé primaire de la table référencée

- CREATE TABLE Employé
  (AVS CHAR(11) PRIMARY KEY,
  empN° CHAR(6) UNIQUE , …)
- CREATE TABLE Département
  (dpt_id VARCHAR(18) PRIMARY KEY,
  manager_id CHAR(11) REFERENCES Employé , …)
- **Une clé externe peut référencer une clé secondaire de la table référencée => à préciser**
- CREATE TABLE Département2
  (dpt_id VARCHAR(18) PRIMARY KEY,
  manager_id CHAR(6) REFERENCES Employé (empN°) , …)

Intégrité référentielle

- REFERENCES nom_table [(nom-col)] [action]
  - Qu'est ce qui se passe quand on **détruit/m.à.j. une clé primaire ou unique qui est référencée par un tuple (foreign key) d'une autre table?**
- CREATE TABLE Département
  - (dpt_id VARCHAR(18) PRIMARY KEY,
  - manager_id CHAR(11) **REFERENCES Employé**,…)
- Soit le tuple (dpt_id=Ventes, manager_id=12345,…) dans la table Département
  - **Que se passe-t-il si on détruit l'employé d'AVS 12345 dans la table Employé ?**

Referential triggered action

- **Deux circonstances**
  - ON DELETE
  - ON UPDATE
- **Trois options**
  - SET NULL
  - SET DEFAULT: valeur par défaut si elle existe, sinon NULL
  - CASCADE : on répercute la m.à.j.
- CREATE TABLE Département
  (dpt_id VARCHAR(18) PRIMARY KEY,
  manager_id CHAR(11) REFERENCES Employé (emp_id)
  ON DELETE SET NULL
  ON UPDATE CASCADE ,
  …)
- Si la clause n'existe pas : refus

Referential triggered action

Trigger ON DELETE:

- *ON DELETE* est suivi d'arguments entre accolades permettant de spécifier l'action à réaliser en cas d'effacement d'une ligne de la table faisant partie de la clé étrangère :
  - *CASCADE* indique la suppression en cascade des lignes de la table étrangère dont les clés étrangères correspondent aux clés primaires des lignes effacées
  - *RESTRICT* indique une erreur en cas d'effacement d'une valeur correspondant à la clé
  - *SET NULL* place la valeur NULL dans la ligne de la table étrangère en cas d'effacement d'une valeur correspondant à la clé
  - *SET DEFAULT* place la valeur par défaut (qui suit ce paramètre) dans la ligne de la table étrangère en cas d'effacement d'une valeur correspondant à la clé

Referential triggered action

Trigger ON UPDATE:

- *ON UPDATE* est suivi d'arguments entre accolades permettant de spécifier l'action à réaliser en cas de modification d'une ligne de la table faisant partie de la clé étrangère :
  - *CASCADE* indique la modification en cascade des lignes de la table étrangère dont les clés primaires correspondent aux clés étrangères des lignes modifiées
  - *RESTRICT* indique une erreur en cas de modification d'une valeur correspondant à la clé
  - *SET NULL* place la valeur NULL dans la ligne de la table étrangère en cas de modification d'une valeur correspondant à la clé
  - *SET DEFAULT* place la valeur par défaut (qui suit ce paramètre) dans la ligne de la table étrangère en cas de modification d'une valeur correspondant à la clé

Les assertions

Les assertions sont des expressions devant être satisfaites lors de la modification de données pour que celles-ci puissent être réalisées. Ainsi, elles permettent de garantir l'intégrité des données. Leur syntaxe est la suivante:

```
CREATE ASSERTION Nom_de_la_contrainte CHECK
(expression_conditionnelle)
```

La condition à remplir peut (et est généralement) être effectuée grâce à une clause *SELECT*.

Les assertions ne sont pas implémentées dans l'ensemble des SGBDR...

Les assertions

```
CREATE ASSERTION a_salaire CHECK (
   NOT EXISTS (SELECT * FROM employe
               WHERE SALAIRE > 4000 AND
               e.num_service <> (SELECT num_service
                                 FROM service
                                 WHERE nom='direction'
                                )
              )
);
```

Il ne doit pas exister de salaire > 4000 ailleurs que dans le service direction

- Les assertions sont des contraintes qui peuvent porter sur plusieurs tables. Elles doivent être vérifiées par le SGBD à chaque fois qu'une des tables mentionnées est modifiée
- CREATE ASSERTION <nom> CHECK (<condition>)
- La condition doit être vraie lors de la création, sinon l'assertion n'est pas créée.

Les assertions

- *Cours(NumC, Sem, Nb_inscrits)*
  Inscription(NumEt, NumC, Sem)

- On veut que *Nb_inscrits* reflète exactement le nombre d'inscrits

- ```
  CREATE ASSERTION NB_INSCR CHECK(
      NOT EXISTS(select * from Cours C where
              C.Nb_inscrits != (select COUNT(*)
                      FROM Inscription i
                      WHERE i.NumC=C.NumC AND
                            i.Sem=C.Sem
                      GROUP BY (i.NumC, i.Sem))
                  )
      );
  SET ASSERTION NB_INSCR DEFERRED;
  ```
  ```

Assertions et Trigger

- Assertions
  - Les assertions décrivent des contraintes qui doivent être satisfaites par la base à tout moment.
  - Une assertion est vérifiée par le SGBD à chaque fois qu'une des tables qu'elle mentionne est modifiée
  - Si une assertion est violée, alors la modification est rejetée
- Triggers
  - Les triggers spécifient explicitement à quel moment doivent-ils être vérifiés (i.e. **INSERT**, **DELETE**, **UPDATE**)
  - Les triggers sont des règles ECA : Evénement, Condition, Action
  - Quand E a lieu, si C est vérifiée alors A est exécutée

Contrainte CHECK(condition)

- Condition que chaque ligne de la table doit vérifier
- Contrainte de colonne et de table
- CREATE TABLE Employé

```
(       AVS CHAR(11) PRIMARY KEY ,
        nom VARCHAR(20) NOT NULL,
        prénoms VARCHAR(30) ,
        age NUMBER CHECK (age BETWEEN 16 AND 70) ,
        sexe CHAR CHECK (sexe IN ('M', 'F')) ,
        salaire NUMBER ,
        commission NUMBER ,
        CONSTRAINT check_sal
                CHECK (salaire * commission <= 7000) )
```

Create TRIGGER

- Contrainte d'intégrité
  - **simple** => clause CHECK dans CREATE TABLE
  - **complexe** => un TRIGGER

- CREATE TRIGGER
  - nouvelle instruction
  - QUAND événement
    - INSERT / DELETE /UPDATE

    SI condition-SQL

    ALORS action
    - refus / instructions SQL

Exemple de TRIGGER

```
Cours(NumC, Sem, Nb_inscrits)
Inscription(NumEt, NumC, Sem)
```

- Pour tout tuple de Prérequis <**nomC, nomCprérequis>,** le cycle de nomCprérequis dans Cours doit être inférieur ou égal à celui de nomC

QUAND : INSERT INTO Prérequis

SI : le cycle de nomCprérequis dans Cours est supérieur à celui de nomC

ALORS : refuser l'insertion

Supprimer une TABLE

- **DROP** : supprimer une table
  - supprime la table et tout son contenu
- DROP TABLE nom_table [CASCADE CONSTRAINTS]
- **CASCADE CONSTRAINTS**
  - Supprime toutes les contraintes de clé externe référençant cette table
  - Si on cherche à détruire une table dont certains attributs sont référencés sans spécifier CASCADE CONSTRAINT: refus

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**
- Supprimer une colonne ou une contrainte: mot clé : **DROP**
- renommer une colonne ou une contrainte: mot clé : **RENAME**

Format de ALTER TABLE

- **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

  }

Exemples

- **Ajout d’une colonne:**

  alter table PRODUIT

  add column POIDS smallint

- **Elimination d’une colonne:**

  alter table PRODUIT

  drop column PRIX

- **Modification d’une colonne:**

  alter table CLIENT

  alter column CAT set '00‘

- **Supprimer un domaine**

  drop domain MATRICULE

Exemples: Ajout et retrait de contraintes

- **Ajout d'un identifiant ou PRIMARY KEY**

  alter table CLIENT
  add primary key (NCLI)

- **Colonne UNIQUE**

  alter table CLIENT
  add unique (NOM,ADRESSE,LOCALITE)

- **Colonne obligatoire**

  alter table CLIENT
  modify CAT not null
  alter table CLIENT
  modify ADRESSE null

- **Clés étrangère:**

  alter table COMMANDE
  add foreign key (NCLI) references CLIENT

Exemples: Ajout et retrait de contraintes

```
create table DETAIL (NCOM char(12) not null,
                     NPRO char(15) constraint C1 not null,
                     QCOM decimal(8),
       constraint C2 primary key (NCOM,NPRO),
       constraint C3 foreign key (NPRO) references PRODUIT)

alter table CLIENT
add constraint C_CLI_U unique (NOM,ADRESSE,LOCALITE)

alter table COMMANDE
add constraint C5 foreign key (NCLI) references CLIENT


alter table DETAIL
drop constraint C2
```

Les structures physiques

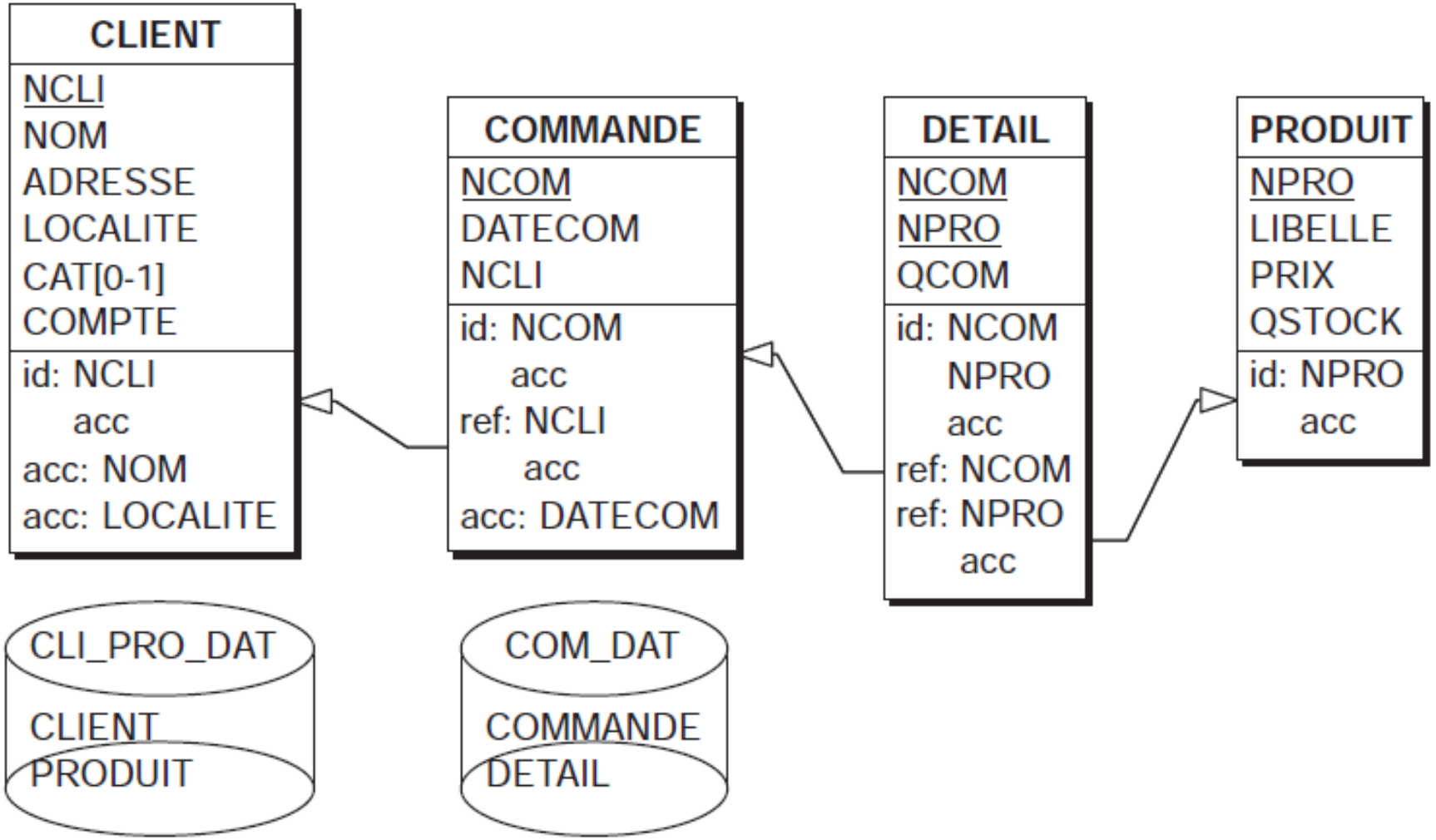


Schéma physique d'une base de données spécifiant les index et les espaces de stockage

Les structures physiques

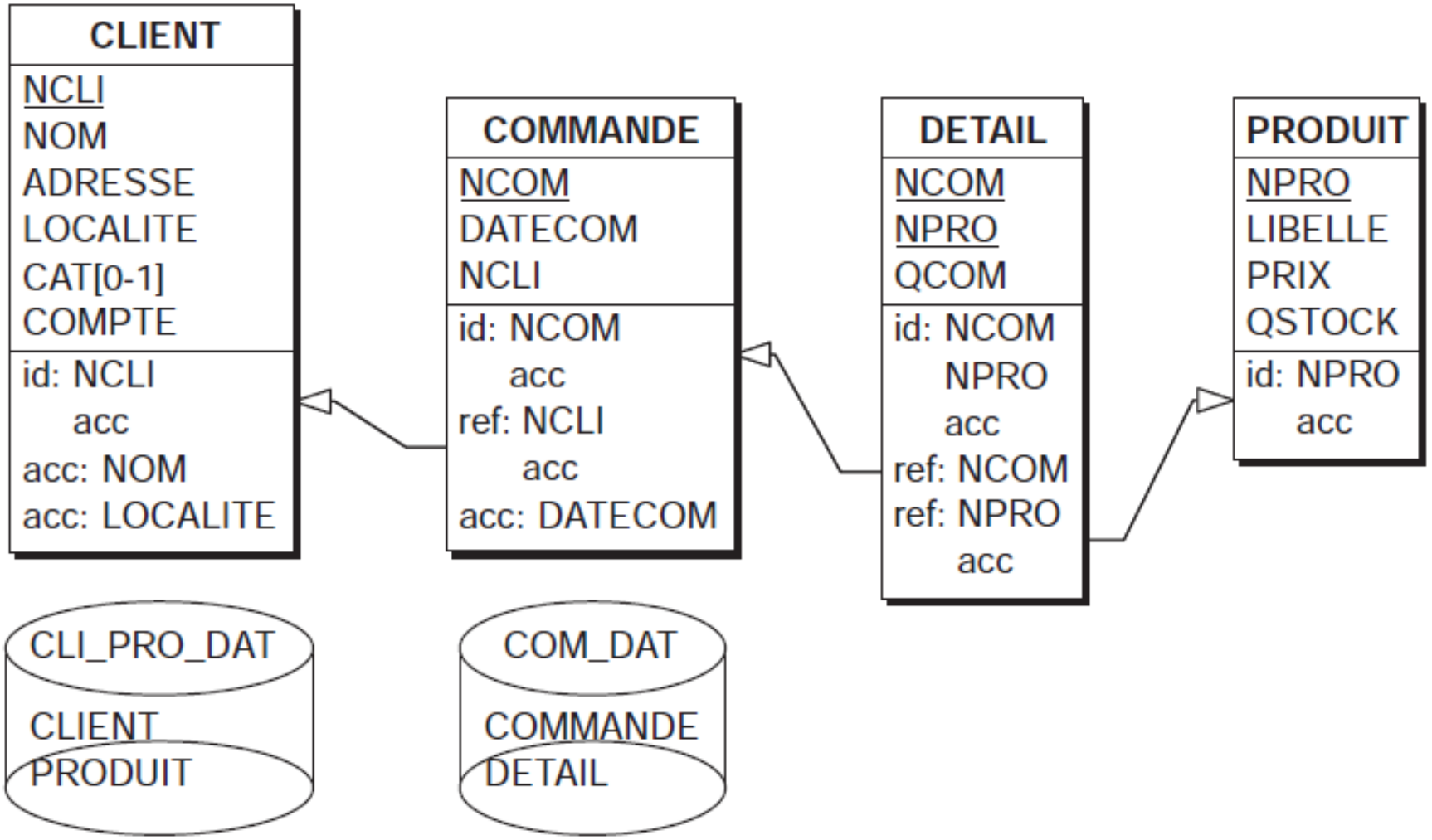


Schéma physique d'une base de données spécifiant les index et les espaces de stockage

Les structures physiques

- **Création d'un index**

  create index XCLILOC
  on CLIENT (LOCALITE)
- **Si les colonnes de l'index forme un identifiant:**

  create unique index XCLI_NCLI on CLIENT (NCLI desc)
  create unique index XDET1 on DETAIL (NCOM asc, NPRO desc)
- **Supprimer un index**

  drop index XDET1
- Un **espace de stockage** est créé comme tout objet SQL :

  create dbspace CLI_PRO_DAT
- On spécifiera ensuite pour chaque table dans quel espace ses lignes doivent être stockées :

  create table CLIENT (...) **in CLI_PRO_DAT**

Langage de Manipulation des Données (LMD)

Langage de Manipulation de Données

- On dispose classiquement de trois opérations : l'**insertion**, la **suppression** et la **mise à jour,** pour gérer les données d'une table.

- L'insertion se fait enregistrement par enregistrement (fastidieux!!!)

- Les opérations de suppression et de modification des données se font à partir de critères de sélection des enregistrements (lignes) à modifier ou à supprimer.

- Il est également possible d'utiliser le résultat d'une requête pour déterminer l'ensemble des valeurs d'une colonne afin d'effectuer cette sélection.

Insertion (INSERT INTO)

- La commande pour insérer des données est de la forme générale suivante :

INSERT INTO <nom de la table> [liste des colonnes] **VALUES** <liste des valeurs>

- Insertion d'un enregistrement dans la table 'voiture'

INSERT INTO voiture (NumVoit, Marque, Couleur)

VALUES (10,'Triumph','Bleue') ;

- Si certaines colonnes sont omises, elles prendront la valeur 'NULL'.

- **Si la liste des colonnes est omise, on considère qu'il s'agit de la liste de celles prises dans l'ordre défini lors de la création de la table.**

Insertion d'enregistrement(s) à partir du résultat d'une requête

- La table dans laquelle on insère les données doit avoir le même nombre de colonnes que la table « résultat » de la requête (et le même type).

- INSERT INTO voiture

SELECT NumVoit, Marque, Type, Couleur

FROM voiturebis

WHERE NumVoit>10;

- **Pour être insérées, les valeurs des colonnes doivent respecter les contraintes d'intégrité associées à la table.**

INSERT INTO

- **Il est possible d'ajouter plusieurs lignes à un tableau avec une seule requête**. Pour ce faire, il convient d'utiliser la syntaxe suivante :

```
INSERT INTO client (prenom, nom, ville, age)
 VALUES
 ('Rébecca', 'Armand', 'Saint-Didier-des-Bois', 24),
 ('Aimée', 'Hebert', 'Marigny-le-Châtel', 36),
 ('Marielle', 'Ribeiro', 'Maillères', 27),
 ('Hilaire', 'Savary', 'Conie-Molitard', 58);
```

A noter : lorsque le champ à remplir est de type VARCHAR ou TEXT il faut indiquer le texte entre guillemet simple.
En revanche, lorsque la colonne est un numérique il n'y a pas besoin d'utiliser de guillemet, il suffit juste d'indiquer le nombre.

INSERT INTO: PostgreSQL

```
INSERT INTO nom_table [ ( nom_colonne [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | requête }
    [ RETURNING * | expression_sortie [ [ AS ] nom_sortie ] [, ...] ]
```

- ***nom_table***: Le nom de la table (éventuellement qualifié du nom du schéma).
- ***nom_colonne:*** Le nom d'une colonne de la table. Le nom de la colonne peut être qualifié avec un nom de sous-champ ou un indice de tableau, si nécessaire. (N'insérer que certains champs d'une colonne composite laisse les autres champs à NULL.)
- ***Default values***: Toutes les colonnes se voient attribuer leur valeur par défaut.
- ***Expression***: Une expression ou une valeur à affecter à la colonne correspondante.

- ***requête***: Une requête (instruction **SELECT**) dont le résultat fournit les lignes à insérer. La syntaxe complète de la commande est décrite dans la documentation de l'instruction
- ***expression_sortie***: Une expression à calculer et renvoyée par la commande **INSERT** après chaque insertion de ligne. L'expression peut utiliser tout nom de colonne de la table. Indiquez * pour que toutes les colonnes soient renvoyées.
- ***nom_sortie***: Un nom à utiliser pour une colonne renvoyée
- En cas de succès, la commande **INSERT** renvoie un code de la forme ***INSERT oid nombre*** (nombre correspond au nombre de lignes insérées)

INSERT INTO: Exemples

```
INSERT INTO films
        VALUES ('UA502', 'Bananas', 105, '1971-07-13', 'Comédie', '82 minutes');
```

la colonne `longueur` est omise et prend donc sa valeur par défaut

```
          INSERT INTO films (code, titre, did, date_prod, genre)
 VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drame');
```

utilise la clause `DEFAULT` pour les colonnes date plutôt qu'une valeur précise

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comédie', '82 minutes');
INSERT INTO films (code, titre, did, date_prod, genre)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drame');
```

```
INSERT INTO films DEFAULT VALUES;
```

Insérer une ligne constituée uniquement de valeurs par défaut

plusieurs lignes en utilisant la syntaxe multi-lignes **VALUES**

```
          INSERT INTO films (code, titre, did, date_prod, genre) VALUES
('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

Insérer dans la table `films` des lignes extraites de la table `tmp_films`

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

```
-- Créer un jeu de 3 cases sur 3
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');
-- Les indices de l'exemple ci-dessus ne sont pas vraiment nécessaires
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

Insérer dans des colonnes de type tableau :

SUPPRESSION (DELETE FROM)

- L'opération de suppression permet de supprimer un ensemble d'enregistrements (lignes) que l'on identifiera avec une expression identique aux conditions de sélection vues précédemment.

- Forme générale:

```
DELETE  FROM nom_de_table
   [WHERE predicat]
```

- DELETE FROM voiture

  WHERE Couleur='Rouge' ;

- **Attention**, si l'on ne spécifie aucune condition, tous les enregistrements sont supprimés.

- Exemple: DELETE FROM personne ;

SUPPRESSION EN CASCADE

- La présence d'une clé étrangère peut empêcher l'effacement d'un enregistrement.
- Il y a un moyen de remédier à cela, lors de la déclaration de cette clé étrangère.
- Si cette clé a été créée avec l'option **on cascade delete**, alors l'effacement est possible, et toutes les données qui référencent la donnée que l'on efface seront effacées aussi

- Ce mécanisme est très pratique à condition qu'il soit parfaitement maîtrisé.
- Dans le cas contraire, un effacement *a priori* anodin peut entrainer l'effacement de quantités d'autres données, sans qu'il soit réellement possible de prévoir lesquelles de façon simple.
- L'option **on cascade delete** est à manipuler avec les plus grandes précautions.

FONCTIONNEMENT DE LA SUPPRESSION

Exemple: **delete from** Marins **where** ddmort - ddnaissance
= (**select** avg(ddmort - ddnaissance) **from** Marins) ;

- Dans ce cas, la requête imbriquée est tout d'abord évaluée, et le résultat comparé à la soustraction. Ces deux opérations sont faites à chaque ligne.

- Si l'effacement était immédiat, le résultat du calcul de la moyenne changerait à chaque ligne, le résultat final de l'opération changerait suivant l'ordre dans lequel les lignes seraient traitées.

Comme le calcul se fait en deux passes, le processus fonctionne comme attendu. Dans un premier temps, les lignes à effacer sont marquées, mais sont toujours prises en compte dans le calcul de la moyenne. Une fois toutes les lignes examinées, les effacements sont effectués.

SUPPRESSION (PostgreSQL 11.9)

```
DELETE FROM [ ONLY ] nom_table [ * ] [ [ AS ] alias ]
    [ USING liste_using ]
    [ WHERE condition | WHERE CURRENT OF nom_curseur ]
    [ RETURNING * | expression_sortie [ [ AS ] output_name ] [, ...] ]
```

- ***nom_table*** Le nom (éventuellement qualifié du nom du schéma) de la table dans laquelle il faut supprimer des lignes.
- Si **ONLY** est indiqué avant le nom de la table, les lignes supprimées ne concernent que la table nommée. Si ONLY n'est pas indiquée, les lignes supprimées font partie de la table nommée et de ses tables filles. En option, * peut être ajouté après le nom de la table pour indiquer explicitement que les tables filles doivent être inclues.
- ***alias*** Un nom de substitution pour la table cible. Quand un alias est fourni, il cache complètement le nom réel de la table. Par exemple, avec DELETE FROM foo AS f, le reste de l'instruction **DELETE** doit référencer la table avec f et non plus foo.
- ***liste_using*** Une liste d'expressions de table, qui permet de faire apparaître des colonnes d'autres tables dans la condition WHERE.

SUPPRESSION (PostgreSQL 11.9)

```
DELETE FROM [ ONLY ] nom_table [ * ] [ [ AS ] alias ]
    [ USING liste_using ]
    [ WHERE condition | WHERE CURRENT OF nom_curseur ]
    [ RETURNING * | expression_sortie [ [ AS ] output_name ] [, ...] ]
```

condition Une expression retournant une valeur de type boolean. Seules les lignes pour lesquelles cette expression renvoie true seront supprimées.

nom_curseur Le nom du curseur à utiliser dans une condition WHERE CURRENT OF. La ligne à supprimer est la dernière ligne récupérée avec ce curseur. Le curseur doit être une requête sans regroupement sur la table cible du **DELETE**. Notez que WHERE CURRENT OF ne peut pas se voir ajouter de condition booléenne.

expression_sortie Une expression à calculer et renvoyée par la commande **DELETE** après chaque suppression de ligne. L'expression peut utiliser tout nom de colonne de la table nommée *nom_table* ou des tables listées dans la clause USING. Indiquez * pour que toutes les colonnes soient renvoyées.

nom_sortie Un nom à utiliser pour une colonne renvoyée.

DELETE FROM: Exemples

```
DELETE FROM films USING producteurs
  WHERE id_producteur = producteurs.id AND producteurs.nom = 'foo';
```

PostgreSQL™ autorise les références à des colonnes d'autres tables dans la condition WHERE par la spécification des autres tables dans la clause USING. Par exemple, pour supprimer tous les films produits par un producteur donné

```
DELETE FROM films
  WHERE id_producteur IN (SELECT id FROM producteur WHERE nom = 'foo');
```

Pour l'essentiel, une jointure est établie entre films et producteurs avec toutes les lignes jointes marquées pour suppression. Cette syntaxe n'est pas standard. Une façon plus standard de procéder consiste à utiliser une sous-selection

```
DELETE FROM films WHERE genre <> 'Comédie musicale';
```

Supprimer tous les films qui ne sont pas des films musicaux

```
DELETE FROM films;
```

Effacer toutes les lignes de la table films

```
DELETE FROM taches WHERE statut = 'DONE' RETURNING *;
```

Supprimer les tâches terminées tout en renvoyant le détail complet des lignes supprimées

```
DELETE FROM taches WHERE CURRENT OF c_taches;
```

Supprimer la ligne de taches sur lequel est positionné le curseur c_taches

MODIFICATION (UPDATE)

- La commande UPDATE permet de modifier les valeurs d'une ou plusieurs colonnes, dans une ou plusieurs lignes existantes d'une table

- Pour cette opération, il faut préciser :
  - la (les) colonne(s) concernée(s) ;
  - la (les) nouvelle(s) valeur(s) ;
  - les enregistrements pour lesquels on modifiera ces valeurs.

- **Exemple:**

UPDATE personne

SET Ville='Faya'

WHERE Ville='NDjamena' ;

forme générale de la commande update:

```
UPDATE nom_table
SET nom_col_1 = {expression_1 | ( SELECT ...) },
    nom_col_2 = {expression_2 | ( SELECT ...) },
    ...
    nom_col_n = {expression_n | ( SELECT ...) }
WHERE predicat
```

MODIFICATION (UPDATE)

```
UPDATE nom_table
SET nom_col_1 = {expression_1 | ( SELECT ...) },
    nom_col_2 = {expression_2 | ( SELECT ...) },
    ...
    nom_col_n = {expression_n | ( SELECT ...) }
WHERE predicat
```

- Les valeurs des colonnes ***nom_col_1, nom_col_2,… nom_col_n*** sont modifiées dans toutes les lignes qui satisfont le prédicat predicat.
- En l'absence d'une clause **WHERE**, toutes les lignes sont mises à jour.
- Les expressions ***expression_1, expression_2,… expression_n*** peuvent faire référence aux anciennes valeurs de la ligne.
- Exemple:

```
update une_table  set a = b, b = a ;
```

UPDATE AVEC REQUETE IMBRIQUEE

```
-- création de la table Marins
 create  table Marins (
   id  int  primary  key,
   nom  varchar(30),
   commune_naissance  varchar(30)
) ;

 -- création de la table Communes
 create  table Communes (
   id  int  primary  key,
   nom  varchar(30),
) ;

 -- mise à jour de la table Marins
 alter  table Marins
 add  column id_commune  int ;

 -- mise à jour de la colonne id_commune
 update Marins
 set id_commune =
   (select id  from Communes  where Communes.nom =
    Marins.commune_naissance) ;
```

Il est également possible de mettre à jour les valeurs d'une table avec des valeurs lues dans cette même table ou une autre table. En d'autres termes, des valeurs lues par un select

Cet exemple montre comment le résultat d'une requête de type select peut être utilisé pour mettre à jour une colonne. Tout se passe comme si, pour chaque ligne de la table Marins, la requête imbriquée était exécutée, avec les paramètres de la ligne courante. Bien sûr, il ne faut pas que le résultat de cette requête fasse plus d'une ligne, sans quoi une erreur sera

générée.

MODIFICATION (UPDATE)

SGBD: PostgreSQL:

```
UPDATE [ ONLY ] nom_table [ * ] [ [ AS ] alias ]
    SET { nom_colonne = { expression | DEFAULT } |
          ( nom_colonne [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
    [ FROM liste_from ]
    [ WHERE condition | WHERE CURRENT OF nom_curseur ]
    [ RETURNING * | expression_sortie [ [ AS ] nom_sortie ] [, ...] ]
```

- ***nom_table***: Le nom de la table à mettre à jour (éventuellement qualifié du nom du schéma).
- Si ***only*** est indiqué avant le nom de la table, les lignes modifiées ne concernent que la table nommée.
- En option, * peut être ajouté après le nom de la table pour indiquer explicitement que les tables filles doivent être inclues.
- **alias:** Un nom de substitution pour la table cible. Quand un alias est fourni, il cache complètement le nom réel de la table.

Par exemple, avec ***UPDATE foo AS*** f, le reste de l'instruction **UPDATE** doit référencer la table avec f et non plus foo.

MODIFICATION (UPDATE)

SGBD: PostgreSQL:

nom_colonne: Le nom d'une colonne dans la table. Le nom de la colonne peut être qualifié avec un nom de sous-champ ou un indice de tableau, si nécessaire. Ne pas inclure le nom de la table dans la spécification d'une colonne cible

condition Une expression qui renvoie une valeur de type boolean. Seules les lignes pour lesquelles cette expression renvoie true sont mises à jour.

nom_curseur Le nom du curseur à utiliser dans une condition WHERE CURRENT OF. La ligne à mettre à jour est la dernière récupérée à partir de ce curseur. Le curseur doit être une requête sans regroupement sur la table cible de l'**UPDATE**. Notez que WHERE CURRENT OF ne peut pas être spécifié avec une condition booléenne.

expression_sortie Une expression à calculer et renvoyée par la commande **UPDATE** après chaque mise à jour de ligne. L'expression peut utiliser tout nom de colonne de la table nommée *nom_table* ou des tables listées dans le FROM. Indiquez * pour que toutes les colonnes soient renvoyées.

nom_sortie Un nom à utiliser pour une colonne renvoyée.

UPDATE: Exemples (PostgreSQL)

```
UPDATE films SET genre = 'Dramatique' WHERE genre = 'Drame';
```

```
UPDATE temps SET temp_basse = temp_basse+1, temp_haute = temp_basse+15, prcp = DEFAULT
WHERE ville = 'San Francisco' AND date = '2005-07-03';
```

```
UPDATE temps SET temp_basse = temp_basse+1, temp_haute = temp_basse+15, prcp = DEFAULT
WHERE ville = 'San Francisco' AND date = '2003-07-03'
RETURNING temp_basse, temp_haute, prcp;
```

```
UPDATE temps SET (temp_basse, temp_haute, prcp) = (temp_basse+1, temp_basse+15, DEFAULT)
WHERE ville = 'San Francisco' AND date = '2003-07-03';
```

```
UPDATE employes SET total_ventes = total_ventes + 1 FROM comptes
WHERE compte.nom = 'Acme Corporation'
AND employes.id = compte.vendeur;
```

```
UPDATE employes SET total_ventes = total_ventes + 1 WHERE id =
(SELECT vendeur FROM comptes WHERE nom = 'Acme Corporation');
```

Incrémenter le total des ventes de la personne qui gère le compte d'Acme Corporation, à l'aide de la clause FROM.
Idem mais en utilisant une sous-requête dans la clause WHERE.

UPDATE: Exemples (PostgreSQL)

```
BEGIN;
-- autres opérations
SAVEPOINT sp1;
INSERT INTO vins VALUES('Chateau Lafite 2003', '24');
-- A supposer que l'instruction ci-dessus échoue du fait d'une violation de clé
-- unique, les commandes suivantes sont exécutées :
ROLLBACK TO sp1;
UPDATE vins SET stock = stock + 24 WHERE nomvin = 'Chateau Lafite 2003';
-- continuer avec les autres opérations, et finir
COMMIT;
```

Tenter d'insérer un nouvel élément dans le stock avec sa quantité. Si l'élément existe déjà, mettre à jour le total du stock de l'élément. Les points de sauvegarde sont utilisés pour ne pas avoir à annuler l'intégralité de la transaction en cas d'erreur

```
UPDATE films SET genre = 'Dramatic' WHERE CURRENT OF c_films;
```

Modifier la colonne *genre* de la table films dans la ligne où le curseur c_films est actuellement positionné :

Langage d'interrogation des Données

Base de données exemple

Voiture

NumVoit	Marque	Type	Type
1	Peugeot	404	Rouge
2	Citroen	SM	Noire
3	Opel	GT	Blanche
4	Peugeot	403	Blanche
5	Renault	Alpine A310	Rose
6	Renault	Floride	Bleue

Personne

NumAch	Nom	Age	Ville	Sexe
1	Nestor	96	Paris	M
2	Irma	20	Lille	F
3	Henri	45	Paris	M
4	Josette	34	Lyon	F
5	Jacques	50	Bordeaux	M

Vente

DateVente	Prix	NumVoit	NumAch
1985-12-03	10 000	1	1
1996-03-30	70 000	2	4
1998-06-14	30 000	4	1
2000-04-02	45 000	5	2

La projection

- **SELECT Nom, Ville**
  FROM personne ;

Nom	Ville
Nestor	Paris
Irma	Lille
Henri	Paris
Josette	Lyon
Jacques	Bordeaux

- **SELECT * FROM personne ;**

NumAch	Nom	Age	Ville	Sexe
1	Nestor	96	Paris	M
2	Irma	20	Lille	F
3	Henri	45	Paris	M
4	Josette	34	Lyon	F
5	Jacques	50	Bordeaux	M

- Les colonnes de la table « résultat » peuvent être renommées par le mot clé AS.

SELECT Ville AS City FROM personne ;

- **SELECT DISTINCT Marque**
  FROM voiture ;

Marque
Peugeot
Citroen
Opel
Renault

- Opérateurs d'expressions de SQL.

+	**Addition**
-	Soustraction
*	Multiplication
/	Division
%	Modulo

- **SELECT Prix, DateVente, (Prix / 6.5596) AS Prix_Euros FROM vente ;**

Prix	DateVente	Prix_Euros
10 000	1985-12-03	1 524.483 200
70 000	1996-03-30	10 671.382 401
30 000	1998-06-14	4 573.449 601
45 000	2000-04-02	6 860.174 401

La projection

- **SELECT UPPER(Nom)**
  AS NomMajuscule
  FROM personne ;

NomMajuscule
NESTOR
IRMA
HENRI
JOSETTE
JACQUES

- **SELECT MONTH(DateVente)**
  AS Mois
  FROM vente ;

Mois
12
3
6
4

- liste (non exhaustive) des opérateurs statistiques de SQL

COUNT	Comptage du nombre d'éléments (lignes) de la table
MAX	Maximum des éléments d'une colonne
MIN	Minimum des éléments d'une colonne
AVG	Moyenne des éléments d'une colonne
SUM	Somme des éléments d'une colonne

- Les fonctions statistiques s'appliquent à l'ensemble des données d'une colonne (sauf pour la fonction COUNT qui s'applique aux lignes de la table entière). Pour toutes ces opérations, la table « résultat » contiendra une seule ligne et souvent une seule colonne.

- **SELECT AVG(Prix)**
  AS Prix_Moyen
  FROM vente ;

Prix_Moyen
38 750.000 0

- **SELECT COUNT(*)**
  AS Nombre_Personne
  FROM personne ;

Nombre_Personne
5

La sélection ou restriction (where)

- Opérateurs de comparaison *classiques* permettant de constituer les expressions en SQL

=	Égal
<>	Différent
<	Inférieur

>	Supérieur
<=	Inférieur ou égal
>=	Supérieur ou égal

- Opérateurs de comparaison spécifiques à SQL

BETWEEN <valeur> AND <valeur>	Appartient à un intervalle
IN <liste de valeurs>	Appartient à un ensemble de valeurs
IS NULL	Teste si la colonne n'est pas renseignée
LIKE	Compare des chaînes de caractères

- Opérateurs et connecteurs logiques

AND	Et : les deux conditions sont vraies simultanément
OR	Ou : l'une des deux conditions est vraie
NOT	Inversion de la condition

- **SELECT * FROM vente WHERE Prix > 50 000 ;**

DateVente	Prix	NumVoit	NumAch
1996-03-30	70 000	2	4

- **SELECT * FROM voiture WHERE Couleur IN ("Blanc","Rouge") ;**

NumVoit	Marque	Type	Couleur
1	Peugeot	404	Rouge

- **SELECT * FROM personne WHERE Age BETWEEN 40 AND 60;**

NumAch	Nom	Age	Ville	Sexe
3	Henri	45	Paris	M
5	Jacques	50	Bordeaux	M

- **SELECT * FROM voiture WHERE Couleur="Blanche" OR Marque="Peugeot" ;**

NumVoit	Marque	Type	Couleur
1	Peugeot	404	Rouge
3	Opel	GT	Blanche
4	Peugeot	403	Blanche

La sélection ou restriction (where)

- **SELECT * FROM personne WHERE NOT (Ville='Paris');**

NumAch	Nom	Age	Ville	Sexe
2	Irma	20	Lille	F
4	Josette	34	Lyon	F
5	Jacques	50	Bordeaux	M

- **SELECT Marque**

  FROM voiture

  GROUP BY Marque ;

Marque
Citroen
Opel
Peugeot
Renault

- **SELECT Marque, COUNT(*) AS Compte FROM voiture GROUP BY Marque ;**

Marque	Compte
Citroen	1
Opel	1
Peugeot	2
Renault	2

- **SELECT Ville, AVG(Âge) AS Moyenne_Age FROM personne GROUP BY Ville ;**

Ville	Moyenne_Age
Bordeaux	50.0000
Lille	20.0000
Lyon	34.0000
Paris	70.5000

- Le résultat de l'opération de groupage peut lui-même être filtré : c'est-à-dire que l'on effectue une sélection des lignes par rapport au contenu des colonnes obtenues dans la table « résultat » précédente. En pratique, on filtre sur le résultat des opérations statistiques appliquées aux sous-ensembles définis par le groupage.

SELECT Marque, COUNT(*) AS Compte FROM voiture

GROUP BY Marque HAVING Compte > 1;

Marque	Compte
Peugeot	2
Renault	2

La sélection ou restriction (where)

- Le mot clé **HAVING** permet d'effectuer une sélection sur le résultat de l'opération de groupage.

- Le mot clé **WHERE** opère une sélection sur les éléments (lignes) de la table avant l'opération de groupage.

- Supposons que l'on veuille éliminer les voitures rouges de notre calcul

```
SELECT Marque, COUNT(*) AS Compte
        FROM voiture
                WHERE NOT (Couleur='Rouge')
                        GROUP BY Marque;
```

Marque	Compte
Citroen	1
Opel	1
Peugeot	1
Renault	2

Requêtes sur plusieurs tables

- Qualification des attributs par leur table d'appartenance.

- **SELECT voiture.Marque, voiture.Couleur FROM voiture ;**

Marque	Couleur
Peugeot	Rouge
Citroen	Noire
Opel	Blanche
Peugeot	Blanche
Renault	Rose
Renault	Bleue

- Cette notation peut devenir rapidement fastidieuse si le nombre de tables est élevé et si leurs noms sont longs.

- Dans ce cas, on désigne la table par un alias plus commode, qui peut être réduit à une simple lettre, plutôt que par son nom complet. **L'alias est indiqué simplement à la suite du nom de la table ou à l'aide du mot clé AS qui est optionnel.**

- **SELECT Vo.Marque, Vo.Couleur**

  FROM voiture

  AS Vo;

Marque	Couleur
Peugeot	Rouge
Citroen	Noire
Opel	Blanche
Peugeot	Blanche
Renault	Rose
Renault	Bleue

Le produit cartésien

- **SELECT ***

  FROM personne, voiture ;

NumAch	Nom	Age	Ville	Sexe	Num Voit	Marque	Type	Couleur
1	Nestor	96	Paris	M	1	Peugeot	404	Rouge
2	Irma	20	Lille	F	1	Peugeot	404	Rouge
3	Henri	45	Paris	M	1	Peugeot	404	Rouge
4	Josette	34	Lyon	F	1	Peugeot	404	Rouge
5	Jacques	50	Bordeaux	M	1	Peugeot	404	Rouge
1	Nestor	96	Paris	M	2	Citroen	SM	Noire
2	Irma	20	Lille	F	2	Citroen	SM	Noire
3	Henri	45	Paris	M	2	Citroen	SM	Noire
4	Josette	34	Lyon	F	2	Citroen	SM	Noire
5	Jacques	50	Bordeaux	M	2	Citroen	SM	Noire
1	Nestor	96	Paris	M	3	Opel	GT	Blanche
2	Irma	20	Lille	F	3	Opel	GT	Blanche
3	Henri	45	Paris	M	3	Opel	GT	Blanche
4	Josette	34	Lyon	F	3	Opel	GT	Blanche
5	Jacques	50	Bordeaux	M	3	Opel	GT	Blanche

•••

Jointure interne (INNER JOIN)

- **SELECT voiture.Marque, voiture.Couleur, vente.Prix**
  FROM voiture, vente
  WHERE voiture.NumVoit=vente.NumVoit ;

Marque	Couleur	Prix
Peugeot	Rouge	10 000
Citroen	Noire	70 000
Peugeot	Blanche	30 000
Renault	Rose	45 000

- Une autre manière d'exprimer la jointure interne passe par un opérateur de jointure spécifique **JOIN**. Il faut bien sûr **spécifier la colonne sur laquelle s'effectue la jointure**

SELECT voiture.Marque, voiture.Couleur, vente.Prix
FROM vente JOIN voiture ON
voiture.NumVoit=vente.NumVoit ;

- Le traitement de la requête est dans ce cas optimisé par le SGBD. **C'est important, car l'opération de jointure est complexe à réaliser pour un SGBD et est coûteuse en temps et en ressources**

- Il est bien sûr possible d'effectuer la jointure sur plus de deux tables : on indique alors les différents critères de jointure entre les tables.

- **SELECT vo.Marque, vo.Couleur, ve.Prix, pe.Nom, pe.Age**
  FROM voiture AS vo, vente AS ve, personne AS pe
  WHERE (vo.NumVoit=ve.NumVoit) AND (pe.NumAch=ve. NumAch);

Marque	Couleur	Prix	Nom	Age
Peugeot	Rouge	10 000	Nestor	96
Citroen	Noire	70 000	Josette	34
Peugeot	Blanche	30 000	Nestor	96
Renault	Rose	45 000	Irma	20

- **Ou SELECT vo.Marque, vo.Couleur, ve.Prix, pe.Nom, pe.Age FROM voiture AS vo JOIN vente AS ve JOIN personne AS pe ON (vo.NumVoit=ve.NumVoit) AND (pe.NumAch=ve. NumAch);kjnhb**

Jointure externe (OUTER JOIN)

- L'opération de jointure interne ne permet pas de répondre à des questions du type : « Quelles sont les voitures qui n'ont pas été vendues ? » .
- À cette fin, il nous faut utiliser un opérateur capable d'inclure dans le résultat les lignes de la table 'voiture' qui n'ont pas de correspondance dans la table « vente » (par rapport aux valeurs de la colonne 'NumVoit') sans qu'il s'agisse d'un produit cartésien : cette opération spécifique se nomme la **jointure externe.**
- L'opérateur SQL de jointure externe s'exprime par le mot clé **OUTER JOIN**. Cette opération n'est pas symétrique : soit on inclut toutes les lignes d'une table, soit toutes celles de l'autre. On précise cela à l'aide des mots clés **LEFT** et **RIGHT** ou en inversant simplement l'ordre des tables dans l'expression de l'instruction de jointure.
- **SELECT voiture.NumVoit, vente.NumVoit, voiture.Marque, voiture.Couleur, vente.Prix**

  FROM voiture LEFT OUTER JOIN vente ON voiture.NumVoit=vente.NumVoit ;

NumVoit	NumVoit	Marque	Couleur	Prix
1	1	Peugeot	Rouge	10 000
2	2	Citroen	Noire	70 000
3	NULL	Opel	Blanche	NULL
4	4	Peugeot	Blanche	30 000
5	5	Renault	Rose	45 000
6	NULL	Renault	Bleue	NULL

Jointure externe (OUTER JOIN)

- On dispose alors d'un moyen de contrôler la cohérence des données entre les tables

- Dans notre cas, on pourra ainsi vérifier qu'il n'y a pas de valeur d'identifiant de voiture dans la table 'vente' (contenu de la colonne 'NumVoit' de la table 'vente') qui ne se trouve pas dans la table 'voiture' (contenu de la colonne 'NumVoit' de la table 'voiture').

- **SELECT voiture.NumVoit, vente.NumVoit, voiture.Marque, voiture.Couleur, vente.Prix**

  FROM vente LEFT OUTER JOIN voiture

  ON voiture.NumVoit=vente.NumVoit ;

NumVoit	NumVoit	Marque	Couleur	Prix
1	1	Peugeot	Rouge	10 000
2	2	Citroen	Noire	70 000
4	4	Peugeot	Blanche	30 000
5	5	Renault	Rose	45 000

- **Ou SELECT voiture.NumVoit, vente.NumVoit, voiture.Marque, voiture.Couleur, vente.Prix**

  FROM voiture RIGTH OUTER JOIN vente

  ON voiture.NumVoit=vente.NumVoit ;

Jointure externe (OUTER JOIN)

- Revenons à la question de départ : « **Quelles sont les voitures qui n'ont pas été vendues ?** »

- Pour ce faire, **il suffit de sélectionner les lignes dont l'une des colonnes issues de la table 'vente' n'a pas pu être mise en correspondance avec une ligne de la table 'voiture'** : le contenu de cette colonne sera vide, ce qui signifie que l'on peut le tester avec le mot clé NULL.

- Par exemple, on teste le contenu de la colonne 'Prix' issue de la table 'vente'.

- **SELECT voiture.NumVoit, voiture.Marque, voiture.Couleur, vente.Prix**

  FROM voiture LEFT OUTER JOIN vente ON voiture.NumVoit=vente.NumVoit

  WHERE vente.Prix IS NULL;

NumVoit	Marque	Couleur	Prix
3	Opel	Blanche	NULL
6	Renault	Bleue	NULL

Tri des résultats des requêtes

- On utilise le mot clé **ORDER BY** pour spécifier la (les) colonne(s) sur laquelle (lesquelles) on souhaite trier le résultat.

- **SELECT Marque, Type**

  FROM voiture

  ORDER BY Marque ;

Marque	Type
Citroen	SM
Opel	GT
Peugeot	404
Peugeot	403
Renault	Alpine A310
Renault	Floride

- Il est possible de préciser l'ordre de tri par les mots clés **ASC** (croissant par défaut) ou **DESC** (décroissant).

- **SELECT Prix, DateVente**

  FROM vente

  ORDER BY Prix DESC ;

Prix	DateVente
70 000	1996-03-30
45 000	2000-04-02
30 000	1998-06-14
10 000	1985-12-03

Les opérations ensemblistes

- **UNION**

SELECT ------- FROM table1 WHERE------

UNION

SELECT ------- FROM table2 WHERE --------;

Par défaut les doublons sont automatiquement éliminés. Pour conserver les doublons, il est possible d'utiliser une clause *UNION ALL*

- **INTERSECT**

SELECT ------- FROM table1 WHERE --------

INTERSECT

SELECT ------- FROM table2 WHERE --------;

- **EXCEPT ou MINUS**

SELECT ------- FROM table1 WHERE --------

EXCEPT

SELECT ------- FROM table2 WHERE --------;

L'opérateur *EXCEPT* n'étant pas implémenté dans tous les SGBD, il est possible de le remplacer par des commandes usuelles :

SELECT a, b FROM table1

WHERE NOT EXISTS (SELECT c, d FROM table2

WHERE a=c AND b=d)

UNION: Exemples

nom	*prénom*
Chose	Jules
Machin	Pierre
Truc	Patrick

Table1

union

nom	*prénom*
Pouf	Jean
Chose	Jules

Table2

=

nom	*prénom*
Chose	Jules
Machin	Pierre
Pouf	Jean
Truc	Patrick

Résultat

```
SELECT nom, prénom
FROM Table1
UNION
SELECT nom, prénom
FROM Table2;
```

nom	*prénom*
Chose	Jules
Machin	Pierre
Truc	Patrick

Table1

union

last-name	*first-name*
Pouf	Jean
Chose	Jules

Table2

=

nom	*prénom*
Chose	Jules
Machin	Pierre
Pouf	Jean
Truc	Patrick

Résultat

```
SELECT nom, prénom
FROM Table1
UNION
SELECT [last-name], [first-name]
FROM Table2;
```

nom	*prénom*
Chose	Jules
Machin	Pierre
Truc	Patrick

Table1

union

nom	*prénom*
Pouf	Jean
Chose	Jules

Table2

=

Col1	*Col2*
Chose	Jules
Machin	Pierre
Pouf	Jean
Truc	Patrick

Résultat

```
SELECT nom AS Col1, prénom AS Col2
FROM Table1
UNION
SELECT nom AS Col1, prénom AS Col2
```

nom	*prénom*
Chose	Jules
Machin	Pierre
Truc	Patrick

Table1

union

nom	*prénom*
Pouf	Jean
Chose	Jules

Table2

=

nom	*prénom*
Machin	Pierre
Pouf	Jean
Truc	Patrick

Résultat (avec critères)

```
SELECT nom, prénom
FROM Table1
WHERE nom>"D"
UNION
SELECT nom, prénom
FROM Table2
WHERE nom>"D";
```

nom	*prénom*
Chose	Jules
Machin	Pierre
Truc	Patrick

Table1

union all

nom	*prénom*
Pouf	Jean
Chose	Jules

Table2

=

Col1	*Col2*
Chose	Jules
Machin	Pierre
Truc	Patrick
Pouf	Jean
Chose	Jules

Résultat

```
SELECT nom, prénom
FROM Table1
UNION ALL
SELECT nom, prénom
FROM Table2;
```

Comment faire en sorte que la requête Union crée une table? Une des techniques possibles consiste à emboîter la requête union dans une commande d'insertion

```
INSERT INTO Table3
SELECT *
FROM (SELECT nom, prénom
FROM Table1
UNION SELECT [last-name], [first-name]
FROM Table2);
```

INTERSECT: Exemples

nom	*prénom*
Chose	Jules
Machin	Pierre
Truc	Patrick

inter

nom	*prénom*
Pouf	Jean
Chose	Jules

=

nom	*prénom*
Chose	Jules

- code SQL sans INTERSECT

```
SELECT nom, prénom
FROM Table1
WHERE Table1.nom IN (SELECT nom FROM Table2) AND Table1.prénom IN (SELECT prénom
FROM Table2);
```

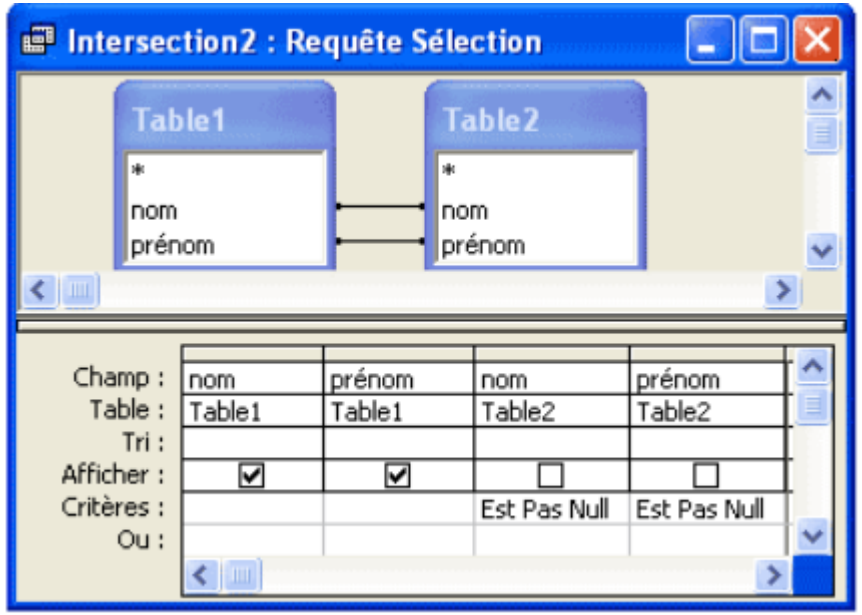
- code SQL avec INTERSECT

```
SELECT nom, prénom
FROM Table1
INTERSECT
SELECT nom, prénom
FROM Table2;
```

- Imaginer une troisième solution qui résulte directement de la définition de l 'intersection

```
SELECT DISTINCT Table1.nom, Table1.prénom
FROM Table1, Table2
WHERE Table1.nom=Table2.nom AND Table1.prénom=Table2.prénom;
```

Nous pouvons encore traduire l'intersection en utilisant des relations (ou jointures) entre les champs des deux tables.



La version SQL de cette requête s'écrit:

```
SELECT Table1.nom, Table1.prénom
FROM Table1 INNER JOIN Table2 ON Table1.nom = Table2.nom AND Table1.prénom =
Table2.prénom
WHERE Table2.nom Is Not Null AND Table2.prénom Is Not Null;
```

EXCEPT ou MINUS: Exemples

nom	*prénom*
Chose	Jules
Machin	Pierre
Truc	Patrick

diff

nom	*prénom*
Pouf	Jean
Chose	Jules

=

nom	*prénom*
Machin	Pierre
Truc	Patrick

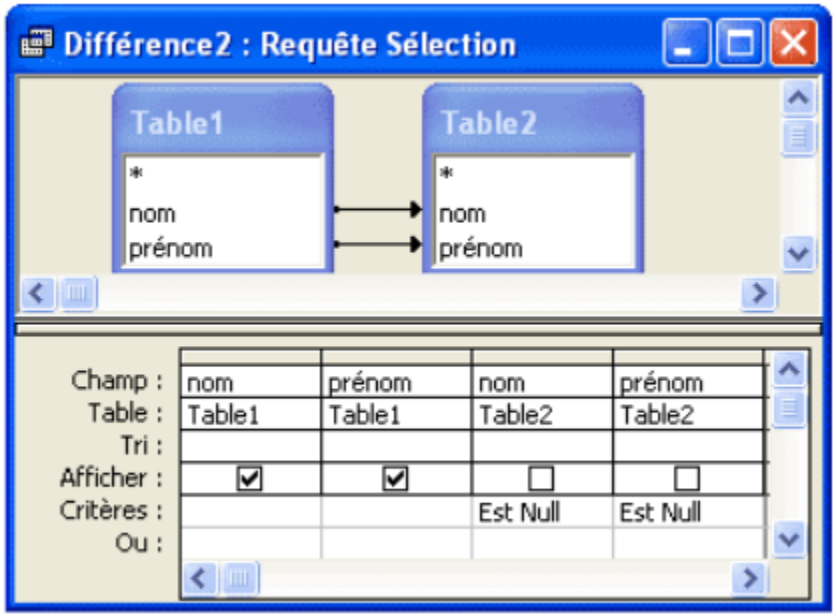
- En SQL 1, cela s ’écrit:

```
SELECT nom, prénom
FROM Table1
WHERE Table1.nom NOT IN (SELECT nom FROM Table2) AND Table1.prénom NOT IN (SELECT
prénom FROM Table2);
```

- La syntaxe précédente, qui fait appel à l'emboîtement autant de fois qu'il y a de colonnes, a été simplifiée par l'introduction de l'opérateur EXCEPT dans SQL2 (MINUS dans le SGBD Oracle).

```
SELECT nom, prénom
FROM Table1
EXCEPT
SELECT nom, prénom
FROM Table2;
```

ACCESS: la jointure gauche est représentée par une flèche allant de la première vers la seconde table




```
SELECT Table1.nom, Table1.prénom
FROM Table1 LEFT JOIN Table2 ON Table1.prénom = Table2.prénom AND Table1.nom =
Table2.nom
WHERE Table2.nom Is Null AND Table2.prénom Is Null;
```

Les Sous requêtes

- Une sous-requête est une requête **à l'intérieur** d'une autre requête

- Une sous requête peut être faite dans une requête de type **SELECT**, **INSERT**, **UPDATE** ou **DELETE**

- Il faut cependant savoir qu'une jointure sera en général **au moins aussi rapide** que la même requête faite avec une sous-requête.

- Par conséquent, s'il est important pour vous d'optimiser les performances de votre application, utilisez plutôt des jointures lorsque c'est possible.

Sous requête dans le FROM

- **Exemple :** on sélectionne toutes les femelles parmi les perroquets et les tortues

```
SELECT Animal.id, Animal.sexe, Animal.date_naissance, Animal.nom, Animal.espece_id
        FROM Animal
                INNER JOIN Espece  ON Espece.id = Animal.espece_id
                        WHERE sexe = 'F' AND Espece.nom_courant IN ('Tortue ', 'Perroquet');
```

- Parmi ces femelles perroquets et tortues, on veut connaître la date de naissance de la plus âgée.

```
SELECT MAX(date_naissance) FROM (
        SELECT Animal.id, Animal.sexe, Animal.date_naissance, Animal.nom, Animal.espece_id
                FROM Animal INNER JOIN Espece ON Espece.id = Animal.espece_id
                        WHERE sexe = 'F' AND Espece.nom_courant IN ('Tortue ', 'Perroquet ') )
                                AS tortues_perroquets_F;
```

Règles à respecter

- Une sous-requête doit toujours se trouver dans des **parenthèses**, afin de définir clairement ses limites.

- Dans le cas des sous-requêtes dans le FROM, il est également obligatoire de **préciser un alias pour la table intermédiaire** (le résultat de notre sous-requête)

- Nommer votre table intermédiaire permet de plus de vous y référer si vous faites une jointure dessus, ou si certains noms de colonnes sont ambigus et que le nom de la table doit être précisé

- Si vous voulez préciser le nom de la table dans le SELECT principal, vous devez écrire SELECT ***MIN(tortues_perroquets_F.date_naissance),*** et non pas ***SELECT MIN(Animal.date_naissance).***

Règles à respecter

- Attention aux noms de colonnes ambigus. Une table, même intermédiaire, ne peut pas avoir deux colonnes ayant le même nom.

- Si deux colonnes ont le même nom, il est nécessaire de renommer explicitement au moins l'une des deux

```
SELECT MIN(date_naissance)
        FROM ( SELECT Animal.id, Animal.sexe, Animal.date_naissance, Animal.nom, Animal.espece_id,
        Espece.id AS espece_espece_id FROM Animal INNER JOIN Espece
                ON Especе.id = Animal.espece_id WHERE sexe = 'F'  AND Espece.nom_courant IN
                        ('Tortue ', 'Perroquet ') ) AS tortues_perroquets_F;
```

Sous requêtes dans les conditions

- Le cas le plus simple est évidemment d'utiliser une sous-requête qui renvoie une valeur.

```
SELECT id, sexe, nom, commentaires, espece_id, race_id
        FROM Animal
                WHERE race_id = (SELECT id
                                        FROM Race
                                                WHERE nom = 'Berger Allemand');
```

```
SELECT COUNT(*) FROM PRODUIT
```

- Cette requête peut également s'écrire avec une jointure plutôt qu'une sous-requête :

```
SELECT Animal.id, sexe, Animal.nom, commentaires, Animal.espece_id, race_id
        FROM Animal
                INNER JOIN Race ON Race.id = Animal.race_id
                        WHERE Race.nom = 'Berger Allemand';
```

Sous requêtes dans les conditions

- Un exemple de requête avec sous-requête qu'il est impossible de faire avec une simple jointure

```
SELECT id, nom, espece_id
        FROM Race
                WHERE espece_id = ( SELECT MIN(id)
                                    FROM Espece );
```

MIN() permet de récupérer la plus petite valeur de la colonne parmi les lignes sélectionnées

Colonne fictive et conditions complexes

- Par exemple, la requête suivante nous renvoie, pour tout produit, le nombre de fournisseurs proposant ce produit :

```
SELECT nomprod, (SELECT COUNT(*)
        FROM PROPOSER PR
                WHERE PR.numprod = P.numprod) AS NB_FOURNISSEURS
                        FROM PRODUIT P
```

- **Conditions complexes:** déclarons d'abord une vue contenant le nombre d'articles proposés par chaque fournisseur

```
CREATE VIEW NB_PROD_PAR_FOU AS     SELECT numfou, (SELECT COUNT(*)
    FROM PROPOSER P
            WHERE P.numfou = F.numfou)  AS NB_PROD FROM FOURNISSEUR F
```

Colonne fictive et conditions complexes

Recherchons les noms des fournisseurs proposant le plus de produits :

```
SELECT nomfou
        FROM FOURNISSEUR F, NB_PROD_PAR_FOU N
                WHERE F.numfou = N.numfou  AND NB_PROD = (SELECT MAX(NB_PROD)
                                                FROM NB_PROD_PAR_FOU)
```

Si le résultat de la sous requête select max() donne 3, nous avons:

```
SELECT nomfou
        FROM FOURNISSEUR F, NB_PROD_PAR_FOU N
                WHERE F.numfou = N.numfou AND NB_PROD = 3
```

- **INSERT ET UPDATE**

```
INSERT INTO PERSONNE (numpers, nom, prenom)
        VALUES ((SELECT MAX(numpers) + 1
                                FROM PERSONNE), 'Abakar', 'Ali');
```

Sous requêtes dans les conditions

- En ce qui concerne les autres opérateurs de comparaison, le principe est exactement le même :

```
SELECT id, nom, espece_id
        FROM Race
                WHERE espece_id < ( SELECT id
                                    FROM Espece
                                            WHERE nom_courant = 'Tortue ');
```

- Dans le cas d'une sous-requête dont le résultat est une ligne, la syntaxe est la suivante :

```
SELECT * FROM nom_table1
        WHERE [ROW](colonne1, colonne2) =SELECT colonneX, colonneY
                FROM nom_table2
                        WHERE...); -- Condition qui ne retourne qu'UNE SEULE LIGNE
```

- Exemple:

```
SELECT id, sexe, nom, espece_id, race_id
        FROM Animal
                WHERE (id, race_id) = ( SELECT id, espece_id
                                        FROM Race
                                                WHERE id = 7);
```

Conditions avec IN et NOT IN

- Avec **IN** qui compare une colonne avec une liste de valeurs.

```
SELECT Animal.id, Animal.nom, Animal.espece_id
        FROM Animal  INNER JOIN Espece  ON Espece.id = Animal.espece_id
                WHERE Espece.nom_courant IN ('Tortue ', 'Perroquet ');
```

- Cet opérateur peut également s'utiliser avec une sous-requête dont le résultat est une **colonne** ou une **valeur**. On peut donc réécrire la requête ci-dessus en utilisant une sous-requête plutôt qu'une jointure

```
SELECT id, nom, espece_id
        FROM Animal
                WHERE espece_id IN ( SELECT id
                                        FROM Espece
                                                WHERE nom_courant IN ('Tortue ',    'Perroquet ') );
```

Conditions avec IN et NOT IN

- La requête suivante nous renvoie le nombre de produits proposés par les fournisseurs proposant le plus de produits :

```
SELECT MAX(NB_PROD) FROM NB_PROD_PAR_FO
```

- Recherchons les numéros des fournisseurs proposant un tel nombre de produits :

```
SELECT N.numfou
        FROM NB_PROD_PAR_FOU N
                WHERE NB_PROD = (SELECT MAX(NB_PROD)
                        FROM NB_PROD_PAR_FOU)
```

- L'écriture avec IN simplifierait les choses si la réponse est supérieur à 1

```
SELECT nomfou
        FROM FOURNISSEUR F
                WHERE F.numfou IN (SELECT N.numfou
                        FROM NB_PROD_PAR_FOU N
                                WHERE NB_PROD = (SELECT MAX(NB_PROD)
                                        FROM NB_PROD_PAR_FOU))
```

Conditions avec IN et NOT IN

- Si l'on utilise **NOT IN**, c'est bien sûr le contraire. on exclut les lignes qui correspondent au résultat de la sous-requête

```
SELECT id, nom, espece_id
        FROM Animal
                WHERE espece_id NOT IN ( SELECT id
                                                FROM Espece
                                                        WHERE nom_courant IN ('Tortue ',      'Perroquet')
);
```

- Les conditions avec **IN** et **NOT IN** sont un peu limitées, puisqu'elles ne permettent que des comparaisons de type "est égal" ou "est différent".

- Avec **ANY** et **ALL**, on va pouvoir utiliser les autres comparateurs (plus grand, plus petit, etc.).

- Bien entendu, comme pour **IN**, il faut des sous-requêtes dont le résultat est soit une **valeur**, soit une **colonne**.

Conditions avec ANY, SOME, ALL

- **ANY** : veut dire "au moins une des valeurs".
- **SOME** : est un synonyme de ANY.
- **ALL** : signifie "toutes les valeurs".

- **ANY:** "Sélectionne les lignes de la table *Animal*, dont l'*espece_id* est inférieur à **au moins une** des valeurs sélectionnées dans la sous-requête".

```
SELECT * FROM Animal
        WHERE espece_id < ANY ( SELECT id
                                FROM Espece
                                        WHERE nom_courant IN ('Tortue ', 'Perroquet ') );
```

- **ALL**: "Sélectionne les lignes de la table *Animal*, dont l'*espece_id* est inférieur à **toutes** les valeurs sélectionnées dans la sous-requête"

```
SELECT * FROM Animal
        WHERE espece_id < ALL ( SELECT id
                FROM Espece
                        WHERE nom_courant IN ('Tortue ', 'Perroquet ') );
```

EXISTS et NO EXISTS

- Les conditions **EXISTS** et **NOT EXISTS** s'utilisent de la manière suivante

```
SELECT * FROM nom_table
        WHERE [NOT] EXISTS (sous-requête)
```

- Une condition avec **EXISTS** sera vraie (et donc la requête renverra quelque chose) si la sous-requête correspondante renvoie au moins une ligne.
- Une condition avec **NOT EXISTS** sera vraie si la sous-requête correspondante ne renvoie aucune ligne
- **Exemple :** on sélectionne les races s'il existe un animal qui s'appelle Balou.

```
SELECT id, nom, espece_id FROM Race
        WHERE EXISTS (SELECT * FROM Animal WHERE nom = 'Balou');
```

- **Exemple :** je veux sélectionner toutes les races dont on ne possède aucun animal.

```
SELECT * FROM Race
        WHERE NOT EXISTS (SELECT * FROM Animal WHERE Animal.race_id = Race.id);
```

Les Opérateurs ensemblistes et les sous requêtes

Syntaxe

(<sous-requête1>) **union** (<sous-requête2>)
(<sous-requête1>) **intersect** (<sous-requête2>)
(<sous-requête1>) **minus** (<sous-requête2>)

Exemples.

1. Restituer les matricules des enseignants qui ne dispensent aucun enseignement :

```
(select Mat-P from PROFESSEURS)
minus
(select distinct Mat from ENSEIGNE);
```

2. Restituer les codes et noms des enseignants de Faya ou Biltine

```
(select Mat, Nom from ENSEIGNANTS where Adresse = 'Faya')
union
(select Mat, Nom from ENSEIGNANTS where Adresse = 'Biltine');
```

3. Restituer les Numéros des étudiants inscrits au module 6 et au module 13 :

```
(select Numéro from INSCRITS where Sigle = 'M6')
intersect
(select Numéro from INSCRITS where Sigle = 'M13');
```

Exemples

- Soit le schema suivant:

  MODULES (Sigle, Libellé, Durée, Mat-P)

  ENSEIGNANTS (Mat, Nom, Prénom 1, Prénom 2, Adresse)

  ENSEIGNE (Sigle, Mat)

1. Restituer le nom des enseignants qui enseignent au moins les mêmes modules que 'ABAKAR'?

Pour avoir $A \subset B$, il faut que A – B = $\varnothing$, c'est-à-dire que la différence entre A et B ne contienne rien.

2. Restituer le nom des enseignants qui enseignent exactement les mêmes modules que ABAKAR?

Dans ce type de requête, on utilise l'égalité ensembliste :

$A=B$ si et seulement si $(A - B) \cup (B - A) = \varnothing$

Sous requêtes non corrélées renvoyant une table

- Soit la requête suivante:

```
SQL> SELECT
  2    (SELECT COUNT(*)
  3        FROM PROPOSER PR
  4       WHERE PR.numfou = F.numfou
  5    ) AS NB_PROD
  6  FROM FOURNISSEUR F;

   NB_PROD
----------
         2
         1
         1
         0
```

```
SQL>  SELECT MAX(NB_PROD) AS MAX_NB_PROD
  2   FROM
  3    (SELECT
  4         (SELECT COUNT(*)
  5          FROM PROPOSER PR
  6          WHERE PR.numfou = F.numfou
  7         ) AS NB_PROD
  8     FROM FOURNISSEUR F
  9      );

MAX_NB_PROD
-----------
          2
```

```
SQL> SELECT nomfou
  2  FROM FOURNISSEUR
  3  WHERE numfou IN
  4    (SELECT numfou
  5     FROM
  6      (SELECT numfou,
  7           (SELECT COUNT(*)
  8            FROM PROPOSER PR
  9            WHERE PR.numfou = F.numfou
 10           ) AS NB_PROD
 11       FROM FOURNISSEUR F
 12       ) N
 13     WHERE NB_PROD =
 14       (SELECT MAX(NB_PROD)
 15        FROM
 16           (SELECT numfou,
 17                 (SELECT COUNT(*)
 18                  FROM PROPOSER PR
 19                 WHERE PR.numfou = F.numfou
 20                ) AS NB_PROD
 21        FROM FOURNISSEUR F
 22           ) N
 23       )
 24     );

NOMFOU
------------------------------
f1
```

Sous requêtes corrélées

- **Une sous-requête corrélée est une sous-requête qui fait référence à une colonne (ou une table) qui n'est pas définie dans sa clause FROM, mais bien ailleurs dans la requête dont elle fait partie**.
- Exemple:

```
SELECT colonne1
        FROM tableA
                WHERE colonne2 IN ( SELECT colonne3
                        FROM tableB
                                WHERE tableB.colonne4 = tableA.colonne5 );
```

- Si l'on prend la sous-requête toute seule, on ne pourra pas l'exécuter :

```
SELECT colonne3
        FROM tableB
                WHERE tableB.colonne4 = tableA.colonne5
```

- Par contre, aucun problème pour l'utiliser comme sous-requête, puisque la clause FROM de la requête principale sélectionne la *tableA*. La sous-requête est donc **corrélée** à la requête principale

Sous requêtes corrélées

```
SQL> SELECT numfou,
  2    (SELECT SUM(qte)
  3     FROM DETAILLIVRAISON D
  4     WHERE D.numfou = F.numfou
  5      ) NB_PROD_L
  6  FROM FOURNISSEUR F;

    NUMFOU  NB_PROD_L
---------- ----------
         1         45
         2
         3         10
         4
```

```
SQL> SELECT nomfou, NB_PROD_L
  2  FROM FOURNISSEUR F,
  3       (SELECT numfou,
  4         (SELECT SUM(qte)
  5          FROM DETAILLIVRAISON D
  6          WHERE D.numfou = F.numfou
  7         ) NB_PROD_L
  8       FROM FOURNISSEUR F
  9        ) L
 10  WHERE F.numfou = L.numfou;

NOMFOU                         NB_PROD_L
------------------------------ ----------
f1                                     45
f2
f3                                     10
f4
```

```
SQL> SELECT nomfou, nomprod
  2  FROM FOURNISSEUR F, PRODUIT P,
  3     (SELECT FF.numfou, PP.numprod
  4     FROM FOURNISSEUR FF, PRODUIT PP
  5        WHERE
  6         (SELECT SUM(qte)
  7          FROM DETAILLIVRAISON L
  8          WHERE L.numfou = FF.numfou
  9          AND L.numprod = PP.numprod
 10         )
 11         =
 12         (SELECT MAX(NB_PROD_L)
 13          FROM
 14           (SELECT numfou, SUM(qte) AS NB_PROD_L
 15            FROM DETAILLIVRAISON L
 16            GROUP BY numprod, numfou
 17           ) Q
 18          WHERE Q.numfou = FF.numfou
 19         )
 20     GROUP BY numfou, numprod
 21     ) M
 22  WHERE M.numprod = P.numprod
 23  AND M.numfou = F.numfou;

NOMFOU                         NOMPROD
------------------------------ ------------------------------
f1                             Roue de secours
f3                             Cotons tiges
```

Références multiples à une même table

- *Quels sont les clients qui habitent dans la même localité* 17 *que le client n° B512?*

```
select *
from   CLIENT
where  LOCALITE in ( select LOCALITE
                     from   CLIENT
                     where  NCLI = 'B512')
```

- Rechercher les commandes qui spécifient une quantité du produit PA60 inférieure à celle que spécifie la commande 30182 pour ce même produit.

```
select *
from   COMMANDE
where  NCOM in (select NCOM
                from   DETAIL
                where  NPRO = 'PA60'
                and    QCOM < (select QCOM
                               from   DETAIL
                               where  NPRO = 'PA60'
                               and    NCOM = '30182'))
```

Condition de totalité

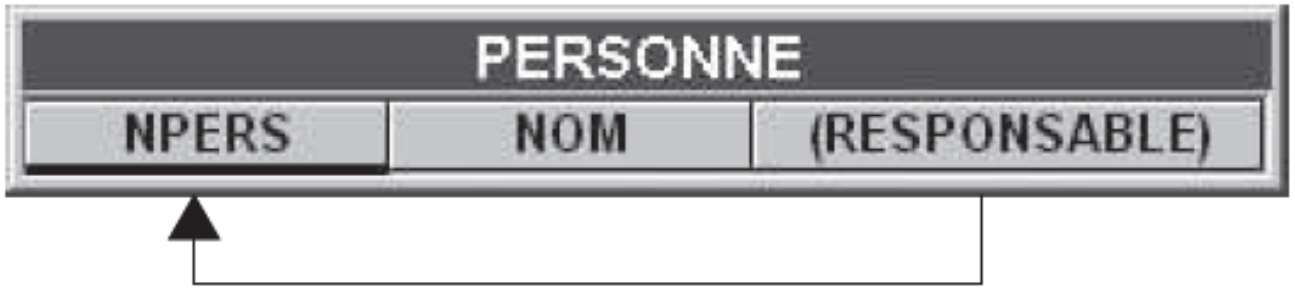
- Exemple: **Recherchons *les commandes qui spécifient tous les produits***.

- Utilisation de l' équivalence de quelque soit: $(\forall x, P(x)) \equiv \neg(\exists x, \neg P(x))$

la COMMANDE M est retenue	→	`select NCOM from COMMANDE M`
si,	→	`where`
il n'existe pas	→	`not exists`
de PRODUIT P,	→	`(select * from PRODUIT P`
tel que	→	`where`
P n'est pas dans	→	`P.NPRO not in`
l'ensemble des PRODUITs	→	`(select NPRO from DETAIL`
commandés par M.	→	`where NCOM = M.NCOM))`

Requêtes sur des structures de données cycliques

- Un schéma cyclique : la table PERSONNE se référence elle-même.




```
create table PERSONNE ( NPERS       char (4) not null,
                        NOM         char(25) not null,
                        RESPONSABLE char (4),
                      primary key (NPERS),
                      foreign key (RESPONSABLE)
                              references PERSONNE)
```

Requêtes sur des structures de données cycliques

- Exemple:

PERSONNE		
NPERS	NOM	(RESPONSABLE)
p1	Mercier	--
p2	Durant	--
p3	Noirons	p1
p4	Dupont	p1
p5	Verger	p4
p6	Dupont	p4
p7	Dermiez	p6
p8	Anciers	p2

- **Question:** donner, pour chaque personne (S, pour *subordonné*) ayant un responsable (R), le numéro et le nom de celui-ci.

```
select S.NPERS, R.NPERS, R.NOM
from   PERSONNE S, PERSONNE R
where  S.RESPONSABLE = R.NPERS
```

Requêtes sur des structures de données cycliques

- **Exemple:** donnez pour chaque personne de nom **Dupont**, son **numéro**, ainsi que le **numéro et le nom de son responsable** s'il existe.

```
select S.NPERS, R.NPERS, R.NOM
from   PERSONNE S, PERSONNE R
where  S.RESPONSABLE = R.NPERS
and    S.NOM = 'Dupont'
union
select NPERS, '--', '--'
from   PERSONNE
where  RESPONSABLE is null
and    NOM = 'Dupont'
```

PERSONNE		
NPERS	NOM	(RESPONSABLE)
p1	Mercier	--
p2	Durant	--
p3	Noirons	p1
p4	Dupont	p1
p5	Verger	p4
p6	Dupont	p4
p7	Dermiez	p6
p8	Anciers	p2

Sécurité des BD et Contrôle d'accès

Introduction

- **Assurer la sécurité d'une base de données** consiste à:
  - **Empêcher l'accès, la modification et la destruction des données** par des accès non autorisés tel que la malveillance ou les inconsistances accidentelles.
  - **Spécifier les autorisations** ,c'est à dire les règles qui permettent de définir qui a le droit d'effectuer un tel type d'opération sur telles données. Elle sont généralement réservées à l'administrateur de le BD.

Sécurité et Intégrité

- **Sécuriser une BD Pourquoi sécurise-t-on la base ?**

(Connaître les causes de violation de la sécurité)

- **1-Causes de la violation de la sécurité:**

  - - Crashes pendant le traitement des transactions (les interruptions système ou bien matériels).

  - - Anomalies dues à la répartition des données sur plusieurs sites (ce problème apparaît dans l'approche repartie).

Sécurité et Intégrité

1-Causes de la violation de la sécurité (suite):

- - Erreurs logiques contradictoires avec l ’hypothèse de conservation de la consistance de la base par des transactions qui s’y déroulent lors de l’échange des données entre plusieurs utilisateurs.

  C’est à dire la perte de consistance lors de l’échange des données entre différents utilisateurs ou transactions

- - La malveillance des utilisateurs c’est à dire la mauvaise utilisation des données d’une façon intentionnelle (modification, destruction,…).

Sécurité et Intégrité

Pour protéger une BD, des mesures de sécurité doivent être prises sur plusieurs niveaux:

- **2- Niveaux de sécurité:**

  - ***Physique:** Les sites qui hébergent les données et les SGBDs doivent être physiquement armés contre les intrusions en force.

  - ***Humain:** Les autorisations doivent être accordées d'une façon sélective afin q'un utilisateur ne cède ses autorisations à une personne malveillante.

Sécurité et Intégrité

- ***Système:** Quelle que soit la sûreté de la BD des faiblesses éventuelles dans la sécurité de systèmes de gestion de base de données peuvent être mises à profit pour pénétrer la base.

  Donc, la sécurité du logiciel système est aussi importante que la sécurité physique de la base.

- ***Base de données:** Le système doit s'assurer que les restrictions des accès ne sont pas violées.

Sécurité et Intégrité

- **3- Contraintes d'intégrité:**

  - Les contraintes d'intégrité sont des règles par rapport à elles se fait le contrôle de validité sur les opérations effectuées.

  - Ils existent plusieurs types de contraintes d'intégrité (temporelles, référentielles ,etc.)

  - Les contraintes d'intégrité protègent la base contre une perte de consistance lors des modifications effectuées par les utilisateurs autorisés.

Contrôle d’accès

Principe:

- Un privilège est l'autorisation qui est accordée à un utilisateur d'effectuer une opération sur un objet.

- Un privilège concerne donc, et doit spécifier, une opération, un objet (ou ressource) de la base de données ou de son environnement, l’utilisateur qui accorde le privilège (c’est celui qui exécute la requête de création ou retrait du privilège) et celui qui le reçoit.

Contrôle d'accès

- La norme SQL considère que c'est l'objet à protéger qui est au centre du processus : les droits d'utilisation de cet objet sont ensuite affectés à des couples de types (nom, identifiant).
-
- Le SGBD stocke les informations suivantes pour chaque objet qu'il contient :
  - **Le type de droit.** Sélection, création, destruction…
  - **L'objet sur lequel s'appliquent les droits.** Une base de données, une table, une vue…
  - **Le nom de l'utilisateur.**
  - **L'identifiant, au sens du mot de passe.**
  - **Le donneur des droits.**

Contrôle d'accès

- Les éditeurs intègrent souvent la notion plus classique d'utilisateur dans un SGBD.
- Ce dernier permet en général de distribuer des autorisations à un ensemble d'utilisateurs qui constituent ainsi un **groupe**.
- Les tâches de gestion en sont facilitées, mais les groupes, qui représentent l'organisation de l'entreprise, sont parfois complexes à gérer.
- L'affectation de droits à une hiérarchie de groupes et sous-groupes peut relever du casse-tête.
- Afin de résoudre ce problème, on dispose d'un autre modèle de distribution des droits : le **rôle**.
- L'instruction pour créer un rôle est la suivante :

CREATE ROLE consultation_seulement ;

Contrôle d'accès

- L'affectation des droits à un rôle et la distribution de rôles à des utilisateurs seront présentées;

- La gestion des rôles, même s'ils font partie de la norme SQL, n'est pas proposée par tous les SGBD. Il en est de même pour les groupes d'utilisateurs ou tout simplement de la notion d'utilisateur qui n'existe pas toujours dans le SGBD.

- En résumé, il n'y a pas de règle générale de gestion des autorisations de connexion au SGBD.

- L'initiative en est laissée à l'éditeur du SGBD. Cet aspect peut provoquer des soucis lors de la migration vers un autre SGBD.

Privilège sur les objets du SGBD

- il est possible de spécifier des droits essentiellement sur les opérations de manipulation de tables (ou de vues considérées comme des tables) suivantes:
  - **select** : extraction de données,
  - **insert** : ajout de lignes,
  - **delete** : suppression de lignes,
  - **update** : modification des valeurs de certaines colonnes,
- La commande suivante autorise les utilisateurs de nom P_MERCIER et S_FINANCIERS à consulter le contenu de la table PRODUIT et à modifier les valeurs de QSTOCK et PRIX

```
grant select, update(QSTOCK,PRIX)
on    PRODUIT
to    P_MERCIER, S_FINANCIERS
```

forme générale est la suivante :

```
GRANT <type de droit>
ON <objet>
TO <nom utilisateur>
```

Exemple d'utilisation de GRANT

- L'administrateur crée un utilisateur « Abakar » avec l'identifiant « fender ».

  CREATE USER Abakar IDENTIFIED BY fender ;

- Il lui donne tous les droits sur la base de données 'Abakar_base' qu'il vient de créer.

  CREATE DATABASE Abakar_base;

  GRANT ALL PRIVILEGES ON Abakar_base.* TO 'Abakar' ;

- L'utilisateur 'Abakar' crée la table '**Test**' dans la base de données 'Abakar_base' et donne l'accès de type « SELECT » à l'utilisateur 'Hassan' préalablement créé

  USE Abakar_base;

  CREATE TABLE Test (NumMorceau INT PRIMARY KEY, Titre CHAR(50)) ;

  GRANT SELECT ON Test TO 'Hassan' ;

- On obtient un message d'erreur : l'utilisateur 'Abakar' n'a pas le droit de retransmettre ses droits à un autre utilisateur. L'administrateur aurait dû entrer la commande suivante :

  GRANT ALL PRIVILEGES ON Abakar_base.* TO Abakar WITH GRANT OPTION ;

Exemple d'utilisation de GRANT

- Lorsque l'on veut donner un droit à tous les utilisateurs du SGBD, on utilise le mot clé **PUBLIC** comme nom d'utilisateur.

GRANT SELECT ON Test TO PUBLIC;

- On peut affiner ces permissions en ne les accordant que pour certains champs de la table. On spécifie alors pour le type de droit le ou les champs sur lesquels ils s'appliquent.

GRANT UPDATE Adresse ON Test TO 'hassan' ;

- L'utilisateur '**Hassan**' a le droit de mettre à jour le champ 'Adresse' de la table 'Test'.

Droits associés aux rôles

- **GRANT** <rôle> **TO** <liste des noms d'utilisateur séparés par des virgules>
- On considère les rôles suivants sur la table '**Test**' :
  - **consultation** : interrogation ;
  - **utilisation** : mise à jour et insertion ;
  - **gestion**: destruction.
- On crée les rôles et on met à jour

les droits nécessaires.

```
CREATE ROLE consultation;
CREATE ROLE utilisation;
CREATE ROLE gestion;
GRANT SELECT ON Test TO consultation;
GRANT UPDATE, INSERT ON Test TO utilisation;
GRANT DELETE ON Test TO gestion;
```

On affecte les rôles aux utilisateurs.

```
GRANT consultation TO 'Abakar', 'Hassan', 'Djimadoum' ;
GRANT utilisation TO 'Abakar', 'Djimadoum' ;
GRANT gestion TO 'Abakar';
```

- Pour retirer les droits accordés par l'instruction GRANT, on utilise l'instruction **REVOKE.**

```
REVOKE <type de droit>
ON <objet>
FROM <nom utilisateur>
```

```
REVOKE SELECT
ON Test
FROM 'Hassan' ;
```

Privilège sur une table ou vue

- Transmission de privilège à d'autres utilisateurs:

```
grant all privileges
on    CLIENT
to    P_MERCIER, S_FINANCIERS
with  grant option
```

- Un privilège peut être accordé à tous les utilisateurs :

```
grant select
on    COM_COMPLETE
to    public
```

Privilège sur une table ou vue

- Ce privilège permet à un utilisateur d'exécuter un programme d'application ou une procédure qui manipule le contenu d’une base de données.

```
grant run
on    SUP_DETAIL
to    public
```

```
grant run
on    COMPTA01
to    S_FINANCIERS
with  grant option
```

- Il existe d'autres privilèges liés notamment à l'administration de la base de données, en particulier à la définition et à la modification de structures de données : create table, alter table, drop index.

Suppression des privilèges

- revoke permet de retirer un privilège préalablement accordé

```
revoke update(PRIX)
on     PRODUIT
to     P_MERCIER

revoke run
on     COMPTA01
from   P_MERCIER
```

- Il est possible de retirer la faculté de transmettre un privilège par une commande telle que :

```
revoke grant option for update (COMPTE)
on CLIENT
from P_MERCIER
```