

## Le langage SQL

Le langage SQL est une évolution commercialisée par IBM du langage SEQUEL initialement développé à IBM San-José comme un langage de recherche. Ce langage peut être perçu comme une expression agréable et très complète de séquences d'opérations relationnelles.

Nous illustrerons SQL sur la base de données relationnelles composées des relations suivantes :

**client**(N°cl, nom\_cl, adresse\_cl)

**article**(N°art, nom\_art, prix\_achat\_art, prix\_vente\_art, N°fourn, poids\_art, coul\_art)

**commande**(N°cde, N°cl, date\_cde, N°fourn)

**ligne\_commande**(N°cde, N°ligne, N°art, prix\_unit, Qte\_cde)

**fournisseur**(N°fourn, nom\_fourn, adresse\_fourn)

### 1. Structure générale du langage

La forme générale :

SELECT  $x_1, x_2, \dots, x_p$

FROM R

WHERE F

où  $x_1, x_2, \dots, x_p$  sont les noms des attributs, R est le nom d'une relation et F est une expression logique qui utilise les attributs de R.

**SELECT** permet de rechercher les attributs que l'on souhaite extraire de la base de données

**FROM** indique dans quelle table (relation) les données sont stockées

Ces deux clauses sont généralement suivies d'une clause **WHERE** pour définir la condition à remplir par les valeurs d'un tuple.

Le résultat d'une clause SELECT est une table, sous-ensemble de la table de départ. Par défaut, cette table-résultat s'affiche à l'écran, mais elle peut servir de point de départ pour une autre clause SELECT (requête imbriquée)

### 2. Projection d'une table

Rappelons qu'une projection effectue l'extraction de colonnes (attributs) spécifiées d'une relation puis élimine les tuples en double. Une projection s'exprime à l'aide du langage SQL par la clause :

SELECT liste d'attributs

FROM nom de relation

#### choisir les attributs

La clause SELECT permet de désigner les attributs que l'on veut voir affiché. L'ordre des attributs du résultat sera l'ordre d'apparition dans le SELECT ou l'ordre spécifié dans la structure de la table pour un SELECT

Exemple : pour afficher tous les noms de fournisseurs :

SELECT nom\_four

FROM fournisseur

### Empêcher les répétitions de lignes

Par défaut, SQL affiche le résultat d'une requête sans éliminer les répétitions de lignes. Pour le cas contraire, il suffit de rajouter UNIQUE ou DISTINCT dans la clause SELECT.

Alors que la même question sans double s'exprime :

```
SELECT UNIQUE nom_four
FROM fournisseur
```

L'astérisque \* désigne tous les attributs d'une relation :

```
Exemple : SELECT *           ⇔      SELECT N°cl, nom_cl, adresse_cl
           FROM client        FROM client
```

## 3. Sélection dans une table

### 3.1 les éléments de la clause WHERE

La clause WHERE permet de spécifier un critère de sélection ou prédicat. Si ce prédicat est vérifié par le tuple, celui-ci est retenu. On réalise ainsi l'opération relationnelle de sélection.

Le prédicat est une expression logique composée d'une suite de conditions combinées entre elles par les opérateurs logiques AND, OR ou NOT.

Un élément d'une expression logique peut prendre une des formes suivantes :

- Comparaison à une valeur (=, ≠, <, >, <=, >=)
- Comparaison à un intervalle de valeurs (BETWEEN)
- Comparaison à une liste de valeurs (IN, MATCH)
- Comparaison à une filtre (LIKE)
- Test sur l'indétermination d'une valeur (IS NULL)
- Test 'tous' ou 'au moins un' (ALL, ANY)
- Test existentiel (EXIST)
- Test d'unicité (UNIQUE)

### 3.2 Les comparaisons

**comparaison à une constante :** les chaînes de caractères consistent en une séquence de caractères entre apostrophes, par exemple 'Bonjour', '01234'. Pour inclure l'apostrophe dans une chaîne de caractères, il suffit de la doubler.

- pour retrouver tous les clients habitant à Béchar :

```
SELECT *
FROM client
WHERE adresse_cl='Béchar'
```

- pour sélectionner tous les articles dont le poids est supérieur à 500g :

```
SELECT *
FROM article
WHERE poids_art>500
```

**Comparaison à une expression:** une expression numérique peut comporter des opérateurs arithmétiques, des noms d'attributs et des constantes.

- pour sélectionner tous les articles pour lesquels le prix de vente est supérieur ou égal au double du prix d'achat :

```
SELECT *
FROM article
WHERE prix_vente_art >= 2*prix_achat_art
```

**Comparaison avec l'opérateur AND :** le AND représente le ET logique.

- pour sélectionner tous les articles rouges de poids supérieur à 500g :

```
SELECT *  
FROM article  
WHERE coul_art='rouge'  
AND    poids_art>500
```

**Comparaison avec l'opérateur OR :** le OR représente le OU logique.

- Pour sélectionner les articles rouges ou de poids supérieur à 500g :

```
SELECT *  
FROM article  
WHERE coul_art='rouge'  
OR    poids_art>500
```

**Comparaison avec l'opérateur NOT :** l'opérateur NOT inverse la valeur d'une expression logique.

- On souhaite l'inverse de la requête précédente. Pour obtenir les articles qui ne sont pas rouges et dont le poids est inférieur ou égal à 500g :

```
SELECT *  
FROM article  
WHERE NOT(coul_art='rouge' OR poids_art>500)
```

**Comparaison entre deux valeurs :** l'opérateur BETWEEN permet de sélectionner les tuples dont l'attribut spécifié contient une valeur dans un intervalle précis.

- Pour afficher la liste des articles dont le prix d'achat est compris entre 500 et 1000 DA :

```
SELECT *  
FROM article  
WHERE prix_achat_art BETWEEN 500 AND 1000
```

- NOT BETWEEN permet de sélectionner les tuples dont un attribut contient une valeur en dehors d'un intervalle.

**Comparaison avec une liste de valeurs :** l'opérateur IN permet de sélectionner les tuples dont l'attribut spécifié contient une valeur appartenant à une liste.

- pour afficher la liste des articles de couleur soit rouge soit vert :

```
SELECT *  
FROM article  
WHERE coul_art IN ('rouge', 'vert')
```

- NOT IN permet de sélectionner les tuples dont un attribut contient une valeur autre que celle de la liste.

**Comparaison avec un filtre pour caractères :** l'opérateur LIKE permet de sélectionner les tuples dont l'attribut spécifié (de type caractère) contient une valeur correspondant au filtre donné. Avec les opérateurs '=' ou 'IN', la valeur de l'attribut doit correspondre au filtre exactement à la constante qui suit. Deux caractères spéciaux permettent de définir le filtre :

% : n'importe quelle séquence de 0 ou plusieurs caractères.  
\_ : un seul caractère quelconque.

- pour retrouver un client dont le nom commence par Bendj :

```
SELECT *  
FROM client  
WHERE nom_cl LIKE 'Bendj%'
```

- Pour chercher les clients dont le nom commence par Bendj, se termine par ma et est composé de 8 lettres :

```
SELECT *  
FROM client  
WHERE nom_cl LIKE 'Bendj_ma'
```

**Comparaison avec un attribut indéterminé :** avec SQL, on utilise la syntaxe **IS NULL** ou inversement **IS NOT NULL**.

- pour rechercher tous les articles pour lesquels la couleur est indéterminée : **SELECT \***  
FROM article  
WHERE coul\_art IS NULL

### 3.3 Trier les tuples résultats

Il est possible de trier des tuples sélectionnés d'après la valeur d'un ou de plusieurs attributs, ceux-ci doivent obligatoirement faire partie de la liste des attributs dans la clause **SELECT**. Un index n'est pas obligatoire et SQL crée éventuellement un index temporaire si cela peut optimiser sa recherche. Pour ce faire, utiliser la clause **ORDER BY** suivie du nom des attributs sur lesquels il faut trier. En ajoutant **ASC** ou **DESC**, on précise l'ordre croissant ou décroissant (*par défaut le tri se fait par ordre croissant*).

- pour trier les articles selon l'ordre croissant de leur poids :

```
SELECT *  
FROM article  
ORDER BY poids_art
```

- lister les numéros, les noms, les poids, et les couleurs des articles triées par ordre croissant de leur poids on aura :

```
SELECT N°art, nom_art, poids_art, coul_art  
FROM article  
ORDER BY poids_art
```

- ☛ L'expression 'ORDER BY 3' est suffisant, car '3' fait référence au troisième attribut spécifié dans la clause **SELECT**.

Pour demander un tri avec plusieurs attributs, il suffit de citer leur nom ou leur numéro d'ordre dans la clause **ORDER BY**, séparés par des virgules.

- pour trier les articles de poids inférieur ou égal à 100g selon l'ordre croissant de leur poids et à poids égal par prix d'achat décroissant :

```
SELECT *  
FROM article  
WHERE poids_art <= 100  
ORDER BY poids_art ASC, prix_achat_art DESC
```

### 3.4 Les fonctions de groupe

Les fonctions de groupe effectuent un calcul sur l'ensemble des valeurs d'un attribut pour un groupe de tuples. Un groupe est un sous-ensemble des tuples d'une table, pour lesquels la valeur d'un attribut reste constante. Les groupes sont spécifiés par la clause **GROUP BY** suivie du nom de l'attribut sur lequel s'effectue le groupement. En l'absence de clause **GROUP BY**, le groupe est constitué de l'ensemble des tuples sélectionnées.

Les fonctions de groupe classique sont :

**COUNT** : compte le nombre d'occurrences de l'attribut.

**SUM** : calcule la somme des valeurs de l'attribut

**AVG** : calcule la moyenne des valeurs de l'attribut

**MAX** : recherche la plus grande valeur de l'attribut

**MIN** : recherche la plus petite valeur de l'attribut

L'argument de la fonction contient le nom de l'attribut sur lequel elle doit être calculée. L'argument précédé de **DISTINCT** signifie qu'il faut éliminer les répétitions de valeur. Les valeurs indéterminées ne sont jamais prises en compte.

- pour calculer le poids moyen des articles on aurait :

```
SELECT AVG(poids_art)
FROM article
```

- pour calculer le prix de l'article le plus cher du stock on aurait :

```
SELECT MAX(prix_vente_art)
FROM article
```

- pour calculer le poids moyen, la marge maximum, la différence entre le prix de vente maximum et le prix d'achat maximum, pour les articles dont l'attribut `coul_art` est défini :

```
SELECT AVG(poids_art), MAX(prix_vente_art-prix_achat_art),
       MAX(prix_vente_art)-MAX(prix_achat_art)
FROM article
WHERE coul IS NOT NULL
```

- pour compter le nombre de couleurs différentes existant dans le stock :

```
SELECT COUNT(DISTINCT coul_art)
FROM article
```

### 3.5 Requete sur les groupes

#### La clause **GROUP BY**

La clause **GROUP BY** réagence la table résultant du **SELECT** en un nombre minimum de groupes, telsque, à l'intérieur de chaque groupe, l'attribut spécifié possède la même valeur pour chaque tuple. Ce n'affecte pas l'organisation physique de la table. Lorsqu'une clause **GROUP BY** est spécifiée, les fonctions de groupe sont calculées pour chaque groupe.

Cette clause permet d'afficher la valeur de l'attribut commun, suivie de celles de fonctions appliquées à chaque groupe.

## La clause HAVING


La clause HAVING est l'équivalent du WHERE appliqué aux groupes. Le critère spécifié dans la clause HAVING porte sur la valeur d'une fonction calculée sur un groupe. Les groupes ne répondent pas au critère spécifié dans la clause HAVING ne font pas partie du résultat.

- pour rechercher la couleur des articles dont le prix de vente moyen des articles de la couleur est supérieur à 100 :

```
SELECT coul_art, AVG(prix_vente_art)
FROM article
GROUP BY coul_art
HAVING AVG(prix_vente_art)>100
ORDER BY coul_art
```

- pour calculer le prix de vente moyen de chaque couleur d'articles :

```
SELECT coul_art, AVG(prix_vente_art)
FROM article
GROUP BY coul_art
ORDER BY coul_art
```

 Les groupes sont créés d'après la valeur de l'attribut coul\_art, puis pour chaque valeur de coul\_art, SQL calcule la moyenne de l'attribut prix\_vente\_art.

- pour calculer le prix de vente moyen des articles de chaque couleur en excluant les articles pour lesquels le prix d'achat est inférieur à 50 :

```
SELECT coul_art, AVG(prix_vente_art)
FROM article
WHERE prix_achat_art >= 50
GROUP BY coul_art
ORDER BY coul_art
```

## 4. Les sélections sur les tables multiples

SQL permet de réaliser la liaison entre plusieurs tables en utilisant les techniques suivantes :

- La jointure réalise la liaison entre deux tables, via l'égalité d'un des attributs, présent dans les deux tables.
- Les requêtes imbriquées permettent d'employer le résultat d'une requête comme élément d'une autre requête.
- Les opérateurs ensemblistes offrent la combinaison des résultats de requêtes

### 4.1 La jointure


Pour écrire une jointure voici quelques conseils :

- Déterminer les tables à mettre en jeu, les inclure dans la clause FROM
- Inclure les attributs à visualiser dans la clause SELECT
- Si la clause SELECT comporte des fonctions sur les groupes, il faut une clause GROUP BY reprenant tous les attributs cités dans la clause SELECT, sauf les arguments des fonctions en question.
- Déterminer les conditions limitant la recherche. Les conditions portant sur les groupes doivent figurer dans une clause HAVING, celles portant sur des valeurs individuelles dans une clause WHERE
- Pour employer une fonction sur les groupes dans une clause WHERE, on si on a besoin de la valeur d'un attribut d'une autre table, il est nécessaire d'utiliser une requête imbriquée

- Pour fusionner les résultats venant de deux clauses SELECT, il suffit de les joindre par une UNION
- Préciser l'ordre d'apparition des tuples du résultat dans une clause ORDER BY

Exemple : pour sélectionner les articles de couleur 'rouge' et afficher le numéro, le poids du article et le nom du fournisseur :

```
SELECT N°art, poids_art, nom_fourn
FROM article, fournisseur
WHERE article.N°fourn = fournisseur.N°fourn
AND coul_art = 'rouge'
```

 On pourrait penser, de façon simple, que SQL effectue le produit cartésien des tables citées dans la clause FROM. Ainsi, si la 1<sup>ère</sup> table comporte N tuples et la 2<sup>ème</sup> table comporte M tuples, le produit cartésien contient M\*N tuples : ceci correspond à une simple vue de l'esprit

En réalité, l'optimiseur va jouer son rôle en effectuant les sélections et les projections sur les tables de départ.

- un cas particulier simple de jointure sans condition est le produit cartésien. Celui-ci s'exprime très simplement en incluant plusieurs relations dans la clause FROM  
exemple : le produit cartésien des relations article et client  
SELECT \*  
FROM client, article

## 4.2 les requetes imbriquées

Donc pour optimiser la liaison entre plusieurs tables, rien n'empêche d'imbriquer plusieurs requêtes, le résultat de la requête la plus imbriquée servant de table de départ à la requête de niveau immédiatement supérieur.

Il faut s'avoir que le temps nécessaire à obtenir le résultat peut être très différent d'une écriture à l'autre, et qu'il dépend de la logique suivie par l'optimiseur pour organiser la recherche. Il n'est donc pas, à priori, possible de prévoir l'écriture la plus performante.

- pour rechercher tous les articles dont le poids est inférieur au poids de l'article numéro 'A002' :

```
SELECT N°art, poids_art
FROM article
WHERE poids_art < (SELECT poids_art
                   FROM article
                   WHERE N°art='A002')
ORDER BY 2
```

- pour rechercher dans la table article, les articles de même couleur que l'article 'A020' et dont le poids est supérieur au poids moyen de tous les articles :

```
SELECT N°art, nom_art
FROM article
WHERE coul_art = (SELECT coul_art
                  FROM article
                  WHERE N°art='A020')
AND poids_art > (SELECT AVG(poids_art)
                 FROM article)
```

- donner la liste des fournisseurs qui vendent au moins un article de couleur noir

```
SELECT nom_four
FROM fournisseur
WHERE N°_four IN (SELECT N°_four
                  FROM article
                  WHERE coul_art = 'noir')
```

- Si la liste obtenue est traitée par un des opérateurs **ALL** ou **ANY**, la requête est à placer dans une clause **WHERE** composée d'un opérateur de comparaison (=, <>, <, >, ≤, ≥) suivi d'un des opérateurs **ALL** ou **ANY**, et suivi de la requête imbriquée.

- Avec l'opérateur **ALL**, la condition est vérifiée si la comparaison se vérifie pour toutes les valeurs ramenées par la requête imbriquée.
- Avec l'opérateur **ANY** ou **SOME**, la condition est vérifiée si la comparaison se vérifie pour au moins une des valeurs ramenées par la requête imbriquée.
- ☛ L'emploi de **ANY** ou **ALL** est assez subtil et peut facilement conduire à des erreurs. Il est préférable d'employer à leur place des requêtes employant des fonctions sur les groupes ou les requêtes existentielles.

- on veut rechercher la liste des articles dont le prix de vente est supérieur au prix de vente de l'article de couleur blanche le moins cher.

```
SELECT nom_art
FROM article
WHERE prix_vente_art > ANY (SELECT prix_vente_art
                           FROM article
                           WHERE coul_art = 'blanc')
```

Autre méthode :

```
SELECT nom_art
FROM article
WHERE prix_vente_art > (SELECT MIN(prix_vente_art)
                       FROM article
                       WHERE coul_art = 'blanc')
```

- On peut utiliser la clause **EXISTS** après la clause **WHERE**. La condition est vérifiée si la requête imbriquée donne un résultat composé d'au moins un tuple.

Exemple : pour résoudre l'exemple précédent par une requête existentielle :

```
SELECT nom_art
FROM article a
WHERE EXISTS (SELECT *
              FROM article b
              WHERE b.coul_art = 'blanc'
              AND a.prix_vente_art > b.prix_vente_art)
```

- 📖 Pour chaque tuple de la table **article**, on réalise la requête existentielle : Si on trouve un article de couleur 'blanc' dont le prix de vente est inférieur, le tuple sera sélectionné.



- La requête est à placer dans une clause **WHERE** après le mot clé **UNIQUE**. La condition est vérifiée si la requête imbriquée donne un résultat contenant un seul tuple.  
Exemple : pour rechercher la liste des clients qui ont acheté plus d'une fois sur la semaine du 9 au 18 septembre :

```
SELECT N°cl, nom_cl
FROM client
WHERE NOT UNIQUE (SELECT *
                  FROM commande
                  WHERE date_cde BETWEEN '09/09/01' AND '18/09/01')
```

- Une clause **HAVING** permet de spécifier un critère valable pour les groupes. Ce critère peut lui-même dépendre du résultat d'une requête imbriquée.

Exemple : pour rechercher la liste des articles dont la somme des ventes est supérieure à la moyenne de la somme des ventes de tous les articles :

```
SELECT N°art, SUM(prix_u_art*qte_cde)
FROM ligne_commande
GROUP BY N°art
HAVING SUM(prix_u_art*qte_cde) > (SELECT AVG(prix_u_art*qte_cde)
                                FROM ligne_commande)
```

### 4.3 Les opérateurs ensemblistes

les opérations ensemblistes sont :

- ◆ l'union
  - ◆ l'intersection
  - ◆ la différence
  - ◆ le produit cartésien
- L'opérateur **UNION** effectue l'union des résultats de deux requêtes **SELECT**, c'est à dire qu'à partir des deux tables-résultats, il en crée une troisième comportant l'ensemble des tuples des deux tables de départ, en éliminant les répétitions de tuples.  
En faisant suivre l'opérateur **UNION** du mot clé **ALL**, les tuples égaux sont conservés.
  - ☛ Si les tables unies sont des sous-ensembles de la même table, l'emploi de cet opérateur est tout à fait équivalent à celui du **OR** placé entre deux conditions.
  - L'opérateur **INTERSECT** effectue l'intersection des résultats de deux requêtes **SELECT**, c'est à dire qu'à partir des deux tables résultats, il en crée une troisième comportant l'ensemble des tuples communs dans les deux tables de départ.
  - L'opérateur **EXCEPT** effectue la différence des résultats de deux requêtes **SELECT**, c'est à dire qu'à partir des deux tables résultats, il en crée une troisième comportant l'ensemble des tuples de la première requête, qui n'apparaissent pas dans la deuxième.

## 5. L'intégrité des données

Il est important de spécifier toutes les contraintes lors de l'établissement du schéma relationnel, et notamment les contraintes d'intégrité référentielle qui assurent la cohérence entre les clés primaires et les clés étrangères.

## 5.1 Création de domaine

```
CREATE DOMAIN nom_domaine IS type_de_la_donnée  
(EDIT STRING IS format_d'affichage)  
(QUERY HEADER IS nom_de_colonne_à_afficher)
```

Exemple :

```
CREATE DOMAIN date_dom IS date  
EDIT STRING IS 'JJ-MM-AAAA'  
QUERY HEADER IS 'en date de'
```

On a :

Le nom du domaine : date\_dom

Le type de domaine : date

Les attributs qui seront de type date\_dom seront affichés sous la forme :

2 caractères pour le jour

2 caractères pour le mois

4 caractères pour l'année

A chaque requête SELECT les attributs de type date\_dom qui seront affichés auront pour en tête de colonne, le libellé : 'en date de'

## 5.2 Création de table

```
CREATE TABLE nom_de_table  
(définition_des_colonnes  
(, contraintes_de_table))
```

Créer une table consiste à :

Définir les colonnes qui la constituent, donc les attributs de la relation

Eventuellement définir les contraintes portant sur la table ou plus exactement sur certains attributs de cette table.

Définir les colonnes consiste à créer une liste de colonnes.

Pour définir une colonne, on trouve :

Le nom de la colonne.

Le type de la colonne ou le nom d'un domaine.

Les mêmes options que pour un domaine, à savoir : DEFAULT, EDIT STRING, QUERY HEADER.

Les options permettant de définir si l'attribut est le résultat d'une fonction mathématique ou d'une fonction de chaînes de caractères : COMPUTED BY suivi de l'expression à évaluer.

Exemples :

```
1- CREATE DOMAINE domcli IS char(30) ;  
   CREATE TABLE client  
   (cli_num Integer, cli_nom domcli,...) ;
```

on a :

le nom du domaine : domcli

la création de la table client avec entre autres attributs :

◆ Le numéro du client, cli\_num de type entier

◆ Le nom du client de type domcli

```
2- CREATE TABLE statut  
(statut_code char(3) PRIMARY KEY,...) ;
```

on a la création de la table statut avec entre autres l'attribut statut\_code sur trois caractères, défini comme clé primaire de la relation.

## 3- CREATE TABLE candidates

```
(laste_name char(40) NOT NULL,...) ;
```

On a la création de la table candidates avec entre autres l'attribut last\_name codé sur 40 caractères qui est nécessairement informé par l'utilisateur à l'insertion de chaque tuple (NOT NULL).

## 4- CREATE TABLE résumés

```
(employee_id char(5) UNIQUE,...) ;
```

on a la création de la table résumés avec entre autres l'attribut employee\_id qui est codé sur 5 caractères avec contrôle d'unicité de la valeur. On ne peut rencontrer, dans cette table deux tuples portant la même valeur pour cet attribut.

- Si l'on ne précise rien pour un attribut, il peut être non informé (NULL). Si l'on veut contraindre l'utilisateur à donner une valeur à un attribut, il suffit de définir la contrainte NOT NULL pour ce dernier.
- La contrainte DEFAULT vous permet de donner une valeur par défaut à un attribut.
- La contrainte UNIQUE oblige le SGBD à contrôler l'unicité des valeurs pour cet attribut, au niveau d'une table. Un attribut défini comme clé primaire (PRIMARY KEY) est d'office UNIQUE et NOT NULL.
- La contrainte CHECK permet de définir une valeur citée dans une liste suivant l'opérateur IN.

Exemples :

## 1- CREATE TABLE.....

```
(.....,  
clt_type char(16) DEFAULT 'particulier'  
CHECK (clt_type IN ('particulier', 'administration', 'grand_compte'))  
CONSTRAINT type_clients,  
.....) ;
```

on a la création d'une table dans laquelle on définit un attribut clt\_type avec :

- ◆ Par défaut la valeur 'particulier'
- ◆ S'il est informé, cet attribut ne peut avoir que l'une des valeurs citées dans la liste suivant l'opérateur IN.
- ◆ Cette contrainte porte le nom type\_clients.
- ◆ En cas d'erreur le SGBD donnera pour message erreur :violation de la contrainte 'type\_clients'

- Lorsque la clé primaire est constituée d'un seul attribut, il suffit de spécifier la clause PRIMARY KEY dans la définition de l'attribut comme dans le code qui suit :

```
CREATE TABLE.....
```

```
(N°clt INTEGER PRIMARY KEY,...) ;
```

Si la clé primaire est constituée de plusieurs attributs, elle ne pourra être définie qu'au niveau des contraintes de table, donc après la définition des diverses colonnes comme ci dessous :

```
CREATE TABLE assemblage  
(artfini char(8) NOT NULL,  
artprim char(8) NOT NULL,  
assqte Integer NOT NULL,  
PRIMARY KEY(artfini, artprim)CONSTRAINT cleass  
.....) ;
```

### 5.3 La sécurité

Toute opération ne peut être effectuée que si l'utilisateur en a l'autorisation ou le privilège.

Tous les objets manipulables, au niveau d'un BDD, sont définis dans le schéma par le propriétaire de ce dernier. Celui-ci a tous les droits et octroie aux autres utilisateurs diverses autorisations suivant les manipulations qu'ils sont susceptibles de pouvoir faire.

Les clauses GRANT et REVOKE permettent l'octroi ou la suppression de privilèges.

Dès qu'un utilisateur se connecte à une BDD, le SGBD l'identifie en tant que 'USER' possédant un certain profil de privilèges.

 Les privilèges sont SELECT, INSERT, UPDATE, DELETE, CONNECT.....

#### La clause GRANT

GRANT liste\_des\_privilèges ON objet TO liste\_des\_utilisateurs

Exemples :

- GRANT UPDATE, SELECT ON personnel TO Mostefa  
On a Mostefa qui est autorisé à lire et mettre à jour les données de la table personnel
- GRANT UPDATE(N°cli, nom\_cli, ville\_cli) ON client TO Omar  
On a Omar qui est autorisé à ne mettre à jour que les colonnes N°cli, nom\_cli, ville\_cli de la table client.
- GRANT SELECT ON departement TO PUBLIC  
On a tous les utilisateurs qui sont autorisés à lire les données de la table departement

#### La clause REVOKE

REVOKE liste\_des\_privilèges ON objet FROM liste\_des\_utilisateurs

Exemples :

- REVOKE UPDATE ON personnel FROM Mostefa  
On a Mostefa qui n'est plus autorisé à modifier la table personnel
- REVOKE UPDATE (prix\_u\_article) ON article FROM Rafik  
On a Rafik qui n'est plus autorisé à modifier le prix unitaire dans la table article
- GRANT ALL ON client TO Mostefa  
REVOKE DELETE ON client FROM Mostefa  
On a Mostefa qui a tous les droits pour la table client, sauf celui de supprimer des tuples.
- GRANT SELECT ON client TO PUBLIC  
REVOKE SELECT ON client FROM Omar, Rafik  
On a tous les utilisateurs, sauf Omar et Rafik qui peuvent lire les données de la table client.

#### La clause WITH GRANT OPTION

Ceux qui reçoivent un privilège ne peuvent le donner à d'autres utilisateurs sauf si ce privilège leur a été transmis avec l'option 'WITH GRANT OPTION'

Exemple :

- GRANT SELECT ON diplome TO Rafik WITH GRANT OPTION  
On a Rafik qui est autorisée à lire les données de la table diplome et pourra redistribuer ce privilège à d'autres utilisateurs.

### La clause GRANT OPTION FOR

La clause GRANT OPTION FOR permet de retirer à un utilisateur la possibilité de redistribuer un privilège reçu avec l'option 'WITH GRANT OPTION'

Exemple :

- REVOKE GRANT OPTION FOR SELECT ON departement FROM Omar  
On a Omar qui garde la possibilité de lire les données de la table departement mais elle ne pourra plus redistribuer ce privilège.

### La clause CASCADE/RESTRICT

Soit les définitions suivantes :

Mostefa permet à Omar de lire la table client :

```
GRANT SELECT ON client TO Omar WITH GRANT OPTION
```

Omar à son tour autorise rafik à lire cette table :

```
GRANT SELECT ON client TO Rafik WITH GRANT OPTION
```

Supposons que Mostefa retire le privilège à Omar, sans autre option rafik garde son privilège.

Avec l'option CASCADE, les deux privilèges disparaissent ainsi que tous les privilèges dérivés :

```
REVOKE SELECT ON client FROM Omar CASCADE
```

Avec l'option RESTRICT l'instruction est refusée à cause de la présence d'un privilège dérivé :

```
REVOKE SELECT ON client FROM Omar RESTRICT
```

## 5.4 Création une vue

Le concept de vue permet de donner à l'utilisateur une perception simplifiée de la BDD, en ne lui offrant que ce dont il a besoin. Elle apporte une réponse au souci de confidentialité de l'information. La création d'une vue est associée à un SELECT et peut faire l'objet de modifications, sous certaines conditions. On peut imaginer une vue comme une sorte de fenêtre par laquelle il est possible de visualiser ou éventuellement, de modifier l'information contenue dans la table.

La syntaxe est la suivante :

```
CREATE VIEW nom_de_la_vue(liste_des_colonnes)  
AS expression_select
```

Exemples :

```
CREATE VIEW client_5000  
AS SELECT nom, adresse, chiffre FROM client WHERE chiffre>5000
```

On a la vue client\_5000 qui aura pour colonnes nom, adresse et chiffre, c'est à dire par défaut les colonnes citées au niveau du SELECT.

### Vue et mises à jour

Soit la vue suivante :

```
CREATE VIEW clients_Béchar(ref_cli, nom, loc, nb_cde)  
AS SELECT N°cl, nom_cl, adresse_cl, cde_cl FROM client  
WHERE adresse_cl = 'Béchar'
```

```
UPDATE clients_Béchar  
SET nb_cde=0  
WHERE ref_cli > 600
```

L'opération de mise à jour se transforme en fait comme ceci :

```
UPDATE client  
SET cde_cl=0  
WHERE adresse_cl='Béchar'  
AND N°cl > 600  
INSERT INTO clients_Béchar VALUE(200, 'Mostefa', 'Béchar', 09)
```

**Privilèges et vues**

En combinant privilèges et vues, on atteint une grande souplesse de sécurité des données

```
CREATE VIEW article_revient  
AS SELECT N°art, nom_art, prix_vente_art, coul_art  
FROM article
```

- GRANT SELECT, UPDATE ON article\_revient TO Mostefa, Omar
- REVOKE SELECT UPDATE, DELETE, INSERT ON article FROM Rafik

- <sup>\*</sup> La vue créée est parfaitement modifiable et on autorise les utilisateurs à y accéder en lecture et en mise à jour. Ceci fait, on leur interdit alors l'accès à la table permanente pour limiter tout risque de perte d'information.