

*Cours*

# **BASE DE DONNEES LANGAGE SQL**

**Edition:** 2006/2007

**Auteur:** AZZI HAMID

**MODULE : M10 SGBDR**

### 3.2. La définition de tables

Les trois mots clés du LDD sont: **CREATE**, **ALTER**, **DROP**.

La commande **CREATE TABLE** permet de définir une table en définissant les différentes colonnes la composant, en leur associant un type de donnée et permet d'y ajouter un ensemble de contraintes à vérifier.

La syntaxe la plus simple de cette requête est la suivante :

```
CREATE TABLE NomTable (NomCol1 Type (Echelle, [précision]), ...);
```

Exemple :

```
CREATE TABLE département (numdept NUMBER (3), nomdept CHAR(10));
```

Il est possible de donner à certains champs des valeurs par défaut en utilisant la clause **DEFAULT** (qui peuvent être des constantes numériques ou alphanumériques, **CURRENT-DATE** pour la date de saisie, **CURRENT-TIME** pour l'heure de saisie). Sans cette clause la valeur par défaut est **NULL**.

```
CREATE TABLE Produit (numproduit NUMBER (3), désignation CHAR(10), stock NUMBER (5) DEFAULT 0);
```

Il est également possible de créer une table en insérant directement des lignes provenant d'une autre table. Exemple :

```
SELECT COL1 AS NEWCOL2,COL2 AS NEWCOL2
INTO NEWTABLE
FROM TABLE1,TABLE2,...
WHERE CONDITION
```

```
Select etudiants.mat as
matricule_etudiant,note,Ncours as nom_cours
into RESEXAM
from ETUDIANT,EXAMEN
where CONDITION
```

### ORACLE :

```
CREATE TABLE bonus (nom, salaire) AS
SELECT nom, salaire FROM employé
WHERE métier='chef de service';
```

Dans ce cas les attributs de la nouvelle table est l'ensemble ou un sous-ensemble des attributs de l'ancienne table. Par contre, les contraintes de l'ancienne table ne sont pas recopiées dans la nouvelle.

### 3.3. Les contraintes d'intégrité

Une contrainte d'intégrité est une condition qui doit constamment vérifier le contenu de la base de données.

Le langage SQL permet de définir des contraintes et le SGBD empêche à tout moment la violation de celles-ci.

Le SGBD identifie toute contrainte.

Le mot clé **CONSTRAINT** permet à l'utilisateur de choisir l'identifiant d'une contrainte ce qui permet, lors d'une mise à jour, de localiser rapidement la contrainte objet de la violation.

#### 3.3.1. Nullité d'une colonne

On peut imposer à une colonne d'être toujours renseignée. La syntaxe est la suivante :

```
CREATE TABLE NomTable (... , NomCol TYPE() [CONSTRAINT NomCont] NOT NULL, ... );
```

#### 3.3.2. Unicité de valeur dans une colonne

On peut imposer à une colonne d'avoir des valeurs différentes (sauf les valeurs non renseignées). La syntaxe est la suivante:

```
CREATE TABLE NomTable (... , NomCol TYPE() [CONSTRAINT NomCont] UNIQUE, ... );
```

La syntaxe est la suivante:

```
CREATE TABLE NomTable (... , [CONSTRAINT NomCont] UNIQUE (NomCol1, NomCol2, ...), ... );
```

#### 3.3.3. Clé primaire

SQL permet de définir un ensemble de colonnes d'une table comme clé primaire. Le SGBD garantit à tout moment l'**unicité** de la clé dans la table et la **non nullité** de chaque champ de la clé. S'il existe plusieurs clés candidates, SQL permet de définir une seule comme étant Primary Key. Par contre, pour plus de cohérence, on pourrait définir les autres clés comme étant Unique

Si la clé primaire n'est formée que d'une seule colonne, on peut la définir en même temps que la colonne avec la syntaxe suivante :

```
CREATE TABLE NomTable (NomCol1 TYPE(..) [CONSTRAINT NomCont] PRIMARY KEY, ... );
```

Si la clé est formée de plusieurs champs, la définition se fait de la façon suivante:

```
CREATE TABLE NomTable (NomCol1 TYPE(..), ...,
[CONSTRAINT NomCont] PRIMARY KEY (NomCol1, NomCol2,...));
```

#### 3.3.4. Clés étrangères

On peut imposer que les valeurs d'une ou plusieurs colonnes d'une table correspondent à des valeurs existantes d'une clé primaire ou d'un champ unique d'une autre table: c'est l'**intégrité référentielle**. La syntaxe est la suivante :

- Cas d'une seule colonne

```
CREATE TABLE NomTable (... , NomCol TYPE(..) [CONSTRAINT NomCont]
REFERENCES NomTableRef (NomCol) [ON DELETE CASCADE], ... );
```

- Cas de plusieurs colonnes

```
CREATE TABLE NomTable (... , ..., [CONSTRAINT NomCont] FOREIGN KEY
(NomCol1, NomCol2, ..., NomColn) REFERENCES NomTableRef (Col1, Col2, ...,
Coln) [ON DELETE CASCADE], ...);
```

La clause **ON DELETE CASCADE** permet de supprimer les lignes en cas d'effacement d'une ligne de la table référencée.

### 3.3.5. Ensemble de valeurs admises

On peut imposer que les valeurs d'une colonne prennent des valeurs dans un ensemble défini par une condition.

La syntaxe est la suivante :

```
CREATE ... (... , ..., NomColn TYPE () CONSTRAINT NomCont CHECK Condition, ...);
```

- Ou si la condition porte sur plusieurs attributs

```
CREATE ... (... , NomColn TYPE (), [CONSTRAINT NomCont] CHECK Condition, ...);
```

```
CREATE TABLE département (numdept NUMBER(3), nomdept CHAR(10) CHECK(nomdept
in('info','electronique','langues','maths')));
```

## 3.4. Modifier la définition d'une table

### 3.4.1. Ajouter ou modifier des colonnes d'une table :

```
ALTER TABLE nomtable ADD (col1 type1, col2 type2, ...);
```

```
ALTER TABLE table MODIFY (col1 type1, col2 type2, ...);
```

Les colonnes doivent exister dans la table et la modification de type est possible si la colonne ne contient que des valeurs NULL ou si le nouveau type est compatible avec le contenu de la colonne (on ne peut pas diminuer sa taille maximale, ni ajouter NOT NULL si elle contient des lignes ou cette colonne n'est pas renseignée).

### 3.4.2. Ajouter des contraintes à une table :

Ajouter une contrainte de clé étrangère:

```
ALTER TABLE nom_table ADD CONSTRAINT nom_contrainte FOREIGN KEY
(NomCol1, ..., NomColn) REFERENCES NomTable (Col1, ..., Coln);
```

Ajouter une contrainte de clé primaire:

```
ALTER TABLE nom_table ADD CONSTRAINT nom_contrainte PRIMARY KEY
(NomCol1, NomCol2, ...);
```

### 3.4.3. Supprimer une colonne

```
ALTER TABLE NOMTABLE DROP COLUMN NOMCOLUMN
```

### 3.4.4. Supprimer des contraintes à une table :

```
ALTER TABLE NomTable DROP CONSTRAINT NomCont [CASCADE];
```

```
ALTER TABLE NomTable DROP Primary Key [CASCADE];  
ALTER TABLE NomTable DROP Unique((NomCol1, NomCol2, ...) [CASCADE];
```

On ne peut supprimer une contrainte de type primary key ou unique qui fait partie d'une intégrité référentielle. La Clause CASCADE force à supprimer les contraintes de type clé étrangère qui se réfèrent à une clé primaire ou unique.

### 3.4.5. Supprimer une table

```
DROP TABLE NomTable [CASCADE CONSTRAINTS];
```

La suppression d'une table permet de supprimer, à la fois, le schéma et l'extension de celle-ci. La clause Cascade Constraints permet de supprimer toutes les contraintes d'intégrité référentielles qui se réfèrent aux clés unique ou primaire de la table à supprimer.

### 3.4.6. Renommer une table

```
RENAME NomOldTable to NomNewTable;
```

## 4. Langage de manipulation de données LMD

On peut ajouter, modifier ou supprimer des lignes d'une table.

### 4.1. Insertion de lignes dans une table

Quand les valeurs de toutes les colonnes sont connues, la syntaxe est :

```
INSERT INTO NomTable VALUES( Val [, ...]);
```

Quand certaine(s) valeur(s) ne sont pas connues, la syntaxe est :

```
INSERT INTO NomTable(NomCol [, ...]) VALUES( Val [, ...]);
```

Il doit y avoir parfaite correspondance entre la liste des colonnes et la liste de valeurs. Les colonnes non précisées sont automatiquement remplies avec la valeur NULL ou la valeur définie dans le DEFAULT.

Insertion de lignes résultant d'un select : Clause SELECT

```
INSERT INTO NomTable[(NomCol , ...)] SELECT ...
```

CREATE TABLE CLIENT

```
insert into CLIENT Select Num, Nom, Prenom, Adresse from
CLIENT_REC Where condition_recherche;
```

### 4.2. Mise à jour de lignes d'une table

La syntaxe est :

```
Update NomTable SET NomCol = Expression [, ...] [WHERE Condition];
```

```
Update NomTable SET (NomCol1,...) = Select ...[WHERE Condition];
```

La condition WHERE permet de sélectionner les lignes mises à jour. Si la clause WHERE n'existe pas, toutes les lignes sont mises à jour.

Expression peut être une constante, une expression arithmétique. Exemple :

```
Update Client Set Ville='New York' Where NumClient='C1' ;
```

```
Update Employe Set salaire=salaire*1.1 ;
```

### 4.3. Suppression de lignes d'une table

La syntaxe est :

```
DELETE FROM NomTable [WHERE Condition]
```

## 5. Privilèges d'accès à la base

Les ordres `GRANT` et `REVOKE` permettent de définir les droits de chaque utilisateur sur les objets de la base.

L'utilisateur qui crée une table est considéré comme le propriétaire de cette table. Il a tous les droits sur cette table et son contenu. Les autres utilisateurs n'ont aucun droit sur cette table (ni lecture, ni modification) à moins que le propriétaire ne leur donne explicitement ces droits avec un ordre `GRANT`.

```
GRANT privilège ON table TO utilisateur [with grant option]
```

Les privilèges peuvent être :

- Select (droit de lecture)
- Insert (droit d'insertion de lignes)
- Update (droit de modification de lignes)
- Update (col1, col2, ...) (limité à certaines colonnes)
- Delete (droit de suppression de lignes)
- Alter (droit de modification de la déf. d'une table)
- ALL (tous les droits ci-dessus)

On peut associer plusieurs privilèges à plusieurs utilisateurs dans un même ordre `Grant`. L'option `[with grant option]` permet aux utilisateurs à qui l'on a accordé des privilèges de les accorder eux-mêmes à d'autres utilisateurs.

Un utilisateur1 ayant accordé un privilège à l'utilisateur2 peut le reprendre à l'aide de `REVOKE` :

```
REVOKE privilège ON table FROM utilisateur2;
```

*il reprend en même temps tous les droits que cet utilisateur2 aurait pu donner.*

## 6. Langage d'interrogation des données LID

La commande SELECT permet de réaliser l'opération la plus courante dans une base de données, à savoir rechercher des informations selon différents critères. La structure d'une telle requête comporte 3 clauses :

- clause FROM : dans quelle(s) table(s) les données sont stockées.
- clause SELECT : quels sont les attributs que l'on souhaite projeter.
- clause WHERE : quels sont les n-uplets que l'on souhaite sélectionner.

Le résultat est une table :

- Par défaut, s'affiche à l'écran,
- Peut servir pour une autre clause FROM,
- Peut servir pour une autre SELECT (sous-requête)
- Peut être combinée avec d'autres tables ou d'autres Select au moyen d'opérateurs ensembliste

### 6.1. Projection d'une table

```
SELECT [DISTINCT] NomCol [, ...] FROM NomTable [, ...] ;
```

Le caractère \* peut être utilisé si l'on souhaite obtenir toutes les colonnes d'une table. Le clause DISTINCT élimine les lignes résultats identiques.

Si le nom de la colonne n'est pas assez significatif, il est possible de définir un alias qui se déclare immédiatement après l'attribut sous forme d'une chaîne de caractères.

```
SELECT NumClient Numéro, Nom "Nom du Client" FROM CLIENT;
```

### 6.2. Tri du résultat d'une sélection

Les lignes résultant d'un select sont affichées dans un ordre qui dépend des algorithmes internes du SGBD. On peut, dans le select demander que le résultat soit trié (ascendant ASC par défaut ou descendant DESC) en fonction d'un ou plusieurs critères.

```
SELECT design, couleur FROM produit ORDER BY design , couleur DESC ;
```

Les critères de la clause ORDER BY peuvent faire référence à un attribut de la liste des attributs projetés soit par son nom soit par sa position dans la liste (dans une même clause ORDER BY).

La clause ORDER BY peut aussi contenir comme critère de tri une expression construite à partir d'attributs.

### 6.3. Prédicats de sélection

L'utilisation de la clause **WHERE** permet de ne sélectionner qu'une partie des lignes :

```
SELECT NomCol[, ...] FROM NomTable [, ...][WHERE Condition];
```

Un prédicat (condition) simple :

- comparaison à une valeur : exp opérateur\_comparaison exp (>, >=, <, <=, =, <> (ou !=))
- comparaison à une fourchette de valeurs : exp [NOT] BETWEEN exp and exp
- comparaison à une liste de valeurs : exp [NOT] IN liste\_valeurs
- comparaison à un filtre exp de type chaîne [NOT] LIKE chaîne (deux méta-caractères \_ caractère quelconque et % suite de caractères quelconque)
- attribut IS [NOT] NULL (= ou != NULL n'est pas autorisé)

Une expression exp peut être un nom d'attribut, une constante numérique ou alphanumérique ou date, une fonction prédéfinie (USER, SYSDATE, ...) ou une combinaison de ces éléments par des opérateurs arithmétiques (+, \*, -, /)

Un prédicat composé est constitué de plusieurs prédicats simples ou composés; reliés par des opérateurs logiques NOT, OR, AND. Des parenthèses peuvent être utilisées pour imposer une priorité dans l'évaluation du prédicat (NOT est prioritaire, AND et OR ont la même priorité).

Exemples :

```
SELECT design FROM produits WHERE prix <300;
SELECT * FROM fournisseur WHERE ville NOT IN ('Paris', 'Lyon');
SELECT design FROM produit WHERE poids NOT BETWEEN 15 AND 35;
SELECT * FROM fournisseur WHERE nom NOT LIKE '___D%';
SELECT design FROM produit WHERE prix <300 AND poids<100;
SELECT * FROM fournisseur WHERE adresse IS NOT NULL ;
SELECT NomProduit FROM produit WHERE PrixVente >= 2* PrixAchat;
```

#### 6.4. Produit cartésien

Le produit cartésien de deux relations, en SQL, est exprimé dans la clause FROM comportant au moins deux relations.

```
SELECT [DISTINCT] NomCol [, ...] FROM NomTable1,NomTable2 [, ...] ;
```

Exemple :

Client : ('C1','Clinton','Bill','New York'), ('C2','Chirac','Jacques','Paris')

Commande : ('Com1','02-JAN-02','C1'), ('Com2','02-MAR-02','C1')

```
SELECT * FROM client, commande;
```

On obtient :

```
('C1','Clinton','Bill','New York','Com1','02-JAN-02','C1')
('C1','Clinton','Bill','New York','Com2','02-MAR-02','C1')
('C2','Chirac','Jacques','Paris','Com1','02-JAN-02','C1')
('C2','Chirac','Jacques','Paris','Com2','02-MAR-02','C1')
```

#### 6.5. Jointure

#### 6.6. Equi-jointure ou jointure naturelle

La jointure naturelle deux relations où l'ensemble est une jointure où les attributs de la jointure représente l'ensemble des attributs de la clé (primaire ou unique) dans la première relation et l'ensemble des attributs de la clé étrangère qui référence cette clé dans la deuxième relation.

```
SELECT * FROM client,commande where client.NumClient=commande.NumClient;
```

On obtient :

```
('C1','Clinton','Bill','New York','Com1','02-JAN-02','C1')
('C1','Clinton','Bill','New York','Com2','02-MAR-02','C1')
```

**Théta-jointure** : est une jointure dont l'expression du pivot utilise des opérateurs autre que l'égalité, tel que: <, <=, >, >=, != ou <>

## 6.7. Opérateurs ensemblistes

**Union:** L'opérateur d'union entre deux tables permet de créer une table résultat contenant l'ensemble des lignes des deux tables de départ. Les attributs de même rang des tables de départ doivent être de types compatibles.

```
SELECT NomCol [, ...] FROM NomTable [, ...][WHERE Condition]UNION
SELECT NomCol [, ...] FROM NomTable [, ...][WHERE Condition];
```

**Intersection:** L'opérateur d'intersection entre deux tables permet de créer une table résultat contenant l'ensemble des lignes appartenant simultanément aux deux tables. Les attributs de même rang des tables de départ doivent être de types compatibles.

ORACLE :

```
SELECT NomCol [, ...] FROM NomTable [, ...][WHERE Condition] INTERSECT
SELECT NomCol [, ...] FROM NomTable [, ...][WHERE Condition];
```

### SQL SERVER :

Utilisation de EXISTS et NOT EXISTS pour la recherche des intersections et des différences

Les sous-requêtes introduites par EXISTS et NOT EXISTS peuvent s'employer comme deux opérateurs de la théorie des ensembles : Intersection et différence. L'intersection de deux ensembles contient tous les éléments qui appartiennent aux deux ensembles de départ. La différence contient les éléments qui appartiennent seulement au premier des deux ensembles.

L'intersection entre les colonnes **city** des tables **authors** et **publishers** est l'ensemble des villes où se trouvent à la fois un auteur et un éditeur.

```
USE pubs
SELECT DISTINCT city
FROM authors
WHERE EXISTS
  (SELECT *
   FROM publishers
   WHERE authors.city = publishers.city)
```

**Différence:** L'opérateur de différence entre deux tables permet de créer une table résultat contenant l'ensemble des lignes appartenant à la première table et pas à la seconde. Les attributs de même rang des tables de départ doivent être de types compatibles.

ORACLE :

```
SELECT NomCol [, ...] FROM NomTable [, ...][WHERE Condition] MINUS
SELECT NomCol [, ...] FROM NomTable [, ...][WHERE Condition];
```

SQL SERVER :

La différence entre les colonnes **city** des tables **authentés** et **publishers** est l'ensemble des villes où vit un auteur, mais où aucun éditeur ne se trouve, c'est-à-dire toutes les villes sauf Berkeley.

```
USE pubs
SELECT DISTINCT city
FROM authors
WHERE NOT EXISTS
  (SELECT *
   FROM publishers
   WHERE authors.city = publishers.city)
```

Cette requête peut aussi être formulée comme suit :

```
USE pubs
SELECT DISTINCT city
FROM authors
WHERE city NOT IN
  (SELECT city
   FROM publishers)
```

Remarques: Dans une requête utilisant des opérateurs ensemblistes:

- Les lignes identiques sont éliminées (DISTINCT implicite).
- Les noms des colonnes sont ceux du premier ordre SELECT.
- Si une clause ORDER BY est utilisée, elle doit faire référence au numéro de la colonne et non à son nom, car le nom peut être différent dans chacun des ordres SELECT.
- Dans une même requête, plusieurs opérateurs peuvent être combinés, la requête est évaluée de gauche à droite. L'ordre peut être modifié en utilisant des parenthèses.

## 6.8. Les sous-requêtes

La requête dont le résultat sert de valeur de référence dans le prédicat est une requête imbriquée ou sous-requête. On peut imbriquer plusieurs requêtes, le résultat de chaque requête imbriquée servant de valeur de référence dans la condition de sélection de la requête de niveau supérieure: la requête principale. Le prédicat :

```
WHERE liste_exps operateur_sous_requete (SELECT ...)
```

Le nombre d'expressions dans la liste doit être égal au nombre d'expressions projetées dans le SELECT. D'autres opérateurs peuvent être utilisés dans le cas d'une sous-requête, par exemple:

- Les opérateurs comparaison =, !=, <>, <, >, <=, >= suivi de ALL ou ANY (=ANY est équivalent à IN et !=ALL est équivalent à NOT IN).
- L'opérateur de test d'existence EXISTS.

### 6.8.1. Sous-requête indépendante

Une sous requête est dite indépendante si elle est entièrement évaluée avant la requête principale. L'évaluation peut renvoyer une ou plusieurs lignes.

Par exemple, la sélection des employés ayant le même supérieur que l'employé Martin dont le matricule est 'Martin01' peut s'écrire en joignant la table avec elle même.

```
SELECT Autres.nom FROM employe A, employe B WHERE A.sup=B.sup AND
B.Matricule = 'Martin01' ;
```

Mais on peut également formuler cette requête au moyen d'une sous-requête :

```
SELECT nom FROM employe WHERE sup=(SELECT sup from employe WHERE
Matricule='Martin01');
```

Lister les avions du même type que l'avion numéro '8832' et mis en service la même année.

```
SELECT NumAvion FROM AVION WHERE (AnnServ, CodeType)=(SELECT AnnServ,
CodeType from AVION NumAvion='8832');
```

Lister les clients qui ont passé des commandes le 5/6/98.

```
SELECT NumClient, Nom, Adresse FROM CLIENT WHERE NumClient IN(SELECT
NumClient FROM COMMANDE WHERE DateCommande='05-JUN-98');
```

### 6.8.2. Sous-requête dépendante

Une sous-requête est dépendante quand elle fait référence à une table de la requête principale. Par conséquent celle-ci est évaluée pour chaque ligne de la requête principale.

Exemple: Lister les vols et les numéros de pilotes qui habitent la même ville d'un départ d'un vol.

```
SELECT p.NumPilote, v.NumVol, v.VilDep from PILOTE p, VOL v where
v.VilDep=(SELECT PILOTE.Adresse from PILOTE where
p.NumPilote=PILOTE.NumPilote );
```

**Cas particulier de l'opérateur EXISTS:** L'opérateur EXISTS permet de construire un prédicat évalué à VRAI si la sous-requête renvoie au moins une ligne. L'utilisation de l'opérateur EXISTS n'a de sens que si la sous-requête est dépendante.

```
SELECT NomCol [, ...] FROM NomTable [, ...][WHERE EXISTS (SELECT ...)
```

Lister les vols ayant utilisé au moins une fois un avion de code type '741'.

```
SELECT A.NumVol FROM AFFECTATION A WHERE EXISTS(SELECT * from AVION
WHERE A.NumAvion=AVION.NumAvion AND CodeType='741');
```

## 6.9. Expressions et fonctions

La présence d'expressions dans le langage SQL enrichit la projection et les conditions de sélection d'une clause WHERE. Il existe plusieurs types d'expressions:

### 6.10. Fonctions conversion :

Il est possible de restructurer la forme d'une colonne à des fonctions telles que CAST et CONVERT

Il est possible d'effectuer des calculs ce qui permet d'obtenir une colonne "résultat" en sortie.

Dans cet exemple, conversion du prix de money en décimal puis calcul d'un montant "mont.

```
select NumArticle,
       CatArticle,
       QuantiteStock * cast(PrixUnitaire as decimal (5,2)) as mont,
from Produits
```

### 6.10.2. Expressions arithmétiques

<b>SELECT POWER(n,m)</b>	$n^m$ POWER(2,3)=8 POWER(2,-3)=0.125
<b>SELECT ROUND(n[,d])</b>	Arrondit n à $10^d$ (par défaut d= 0) ROUND(5.23, 1)=5.2 ROUND(5.25,1)=5.3
<b>SELECT CEILING(n)</b>	Renvoie le nombre entier le plus petit, supérieur ou égal à l'expression numérique donnée.CEILING(12.45)=13
<b>SELECT FLOOR(n)</b>	Prend la valeur de la partie entière de n FLOOR(12.60)=12
<b>SELECT ABS(n)</b>	Valeur absolue de n
<b>SELECT (n % m)</b>	Donne le reste de la division entière de n par m
<b>SELECT SIGN(n)</b>	Vaut 1 si n >0, -1 si n <0, 0 sinon
<b>SELECT SQRT(n)</b>	Racine carrée de n (valeur NULL si n <0)

### 6.10.3. Expressions chaîne de caractères:

L'opérateur de concaténation || est le seul opérateur sur chaîne de caractères.

```
SELECT NumVol, VilDep||' '||VilArr FROM VOL;
```

<b>LEN(ch)</b>	Renvoie la longueur de la chaîne ch
<b>SUBSTRING(ch,pos [,n])</b>	Extrait de la chaîne ch une sous-chaîne de longueur n à partir de la position pos dans ch (le premier caractère est à la position 0). n est facultatif, par défaut la sous-chaîne va jusqu'à l'extrémité.
<b>RTRIM(chaîne)</b>	permet de récupérer une chaîne de caractère sans espaces à la fin
<b>LTRIM (chaîne)</b>	permet de récupérer une chaîne de caractère sans espaces au début
<b>RIGHT(chaîne, début)</b>	permet de récupérer la fin de la chaîne à partir de la position début.
<b>LEFT(chaîne, long)</b>	permet de récupérer le début de la chaîne sur une longueur de long caractères.
<b>UPPER(ch)</b>	Convertit les minuscules en majuscules
<b>LOWER(ch)</b>	Convertit les majuscules en minuscules
<b>REPLACE(ch1,ch2,ch3)</b>	Remplace dans ch1 toutes les ch2 par ch3

### 6.10.4. Fonctions d'agrégat

<b>AVG</b>	Moyenne	<b>SUM</b>	Somme
<b>COUNT</b>	Nombre d'éléments	<b>VARIANCE</b>	Variance
<b>MAX</b>	Maximum	<b>STDDEV</b>	Ecart type
<b>MIN</b>	Minimum		

A	Numéro	Nom	Classif.	Sexe	Salaire brut
	5	Dominique	3	F	10000,00
	10	Rémi	2	F	15000,00
	35	Sidonie	5	M	14000,00
	40	Dorian	5	M	15500,00
	30	Fadette	2	M	13000,00
	25	Michelle	2	F	14500,00

```
SELECT SUM(salaire), MIN(salaire), MAX(salaire),
       AVG(salaire), COUNT(*) FROM A
```

Résultat



SUM(salaire)	MIN(salaire)	MAX(salaire)	AVG(salaire)	COUNT(*)
<b>82000,00</b>	<b>10000,00</b>	<b>15500,00</b>	<b>13666,66</b>	<b>6</b>

COUNT(\*) compte le nombre de lignes

COUNT(noncol) compte le nombre de valeurs non nulles sur cette colonne

COUNT(DISTINCT nomcol) compte le nombre de valeurs distinctes sur cette colonne

Remarque: Ces fonctions ne peuvent être utilisées dans la clause where car le where ne traite qu'une seule ligne à la fois.

Exemples :

```
SELECT COUNT(*) FROM commandes WHERE NumProduit=102 ;
```

```
SELECT COUNT(*) FROM commandes WHERE NumProduit=102 ;
```

```
SELECT SUM(qute) FROM commandes ;
```

```
SELECT COUNT(DISTINCT CodeType) FROM AVION;
```

### 6.11. Les requêtes de groupage

Il est souvent intéressant de regrouper les données d'une table en sous-tables pour y faire des opérations par groupe. On appelle groupe un ensemble de lignes, résultat d'une requête, qui ont une valeur commune sur un ensemble de colonnes. Cet ensemble de colonnes est appelé le facteur de groupage. Pour regrouper des données, il faut alors utiliser la clause **GROUP BY** suivi du facteur de groupage.

### 6.12. La clause GROUP BY

Cette clause permet de totaliser par groupe les lignes de la table sur le critère indiqué derrière GROUP BY: Elle produit une table (fichier) résultant intermédiaire en groupant les lignes (enregistrements) par colonne. Les colonnes (zones) doivent être séparées par des virgules.

ORDER BY est inutile si vous souhaitez voir la table résultante triée, classée sur les mêmes critères que ceux du GROUP BY.

A	Numéro	Nom	Classif.	Sexe	Salaire brut
	5	Dominique	3	F	10000,00
	10	Rémi	2	F	15000,00
	35	Sidonie	5	M	14000,00
	40	Dorian	5	M	15500,00
	30	Fadette	2	M	13000,00
	25	Michelle	2	F	14600,00

```
SELECT  classific, AVG(salaire)
FROM A
GROUP BY classific
```

Résultat



Classif	AVG(salaire)
2	14200,00
3	10000,00
5	14750,00

**Structure de la commande SELECT :**

```
SELECT <nom colonne 1>, ..., <nom colonne n>  
      FROM <nom de table>  
      GROUP BY <nom colonne x>
```

permet de sélectionner dans la table "nom de table" les colonnes dont le nom est cité et de les regrouper sur la valeur de <colonne x>.

**Remarque :**

Sauf pour la colonne de groupage, **toutes** les autres colonnes **doivent** être spécifiées par **fonction**.

**Exemple :**

Pour mieux comprendre, nous allons passer quelques commandes :

```
SELECT dynastie, SUM(trésor) FROM monarque  
      ORDER BY sum (trésor)  
      GROUP BY dynastie
```

Vous avez une erreur de syntaxe.

```
SELECT dynastie, SUM(trésor) FROM monarque  
      GROUP BY dynastie  
      ORDER BY 2
```

### 6.13.1 La clause HAVING ;

Cette clause permet de totaliser sous condition par groupe les lignes de la table sur le critère indiqué derrière GROUP BY: elle produit une table (fichier) résultante intermédiaire en appliquant une condition de recherche à chaque groupe de la clause GROUP BY

A	Numéro	Nom	Classif.	Sexe	Salaire brut
	5	Dominique	3	F	10000,00
	10	Rémi	2	F	15000,00
	35	Sidonie	5	M	14000,00
	40	Dorian	5	M	15500,00
	30	Fadette	2	M	13000,00
	25	Michelle	2	F	14600,00

```
SELECT classific, AVG(salaire)
FROM A
GROUP BY classific
HAVING COUNT (*) > 1
ORDER BY classific
```

Résultat



Classif	AVG(salaire)
2	14200,00
5	14750,00

Cette clause est toujours utilisée avec GROUP BY. C'est l'équivalent de la clause WHERE mais pour un ensemble de lignes regroupées par GROUP BY.

Il est possible d'utiliser des fonctions avec cette clause.

Exemple :

```
SELECT dynastie, SUM(trésor) FROM monarque
GROUP BY dynastie
HAVING SUM(trésor) > 150
```

Nous affichons deux colonnes (dynastie et trésor) en regroupant par dynastie avec le total des trésors par dynastie. Mais l'affichage n'est fait que si le total des trésors est supérieur à 150.

Autres Exemples : Afficher le nombre total et moyenne d'heures de vol par type d'avion et par année de mise en service, tri par type et année.

```
SELECT CodeType, AnnServ, SUM(NbHVol), AVG(NbHVol) from AVION GROUP BY CodeType, AnnServ;
```

Exemple : lister les clients ayant lancé au moins une commande.

```
SELECT NumClient FROM commandes GROUP BY NumClient HAVING COUNT(*)>1;
```