BASES de DONNEES
SQL

Ce document est basé sur le livre "L'art des Bases de Données : Introduction aux bases de données" de S. Miranda et J-M Busta (Eyrolles) ainsi que sur les supports de cours de Rosine Cicchetti et Nicole Bidoit.

BASES de DONNEES
SQL

Ce document est basé sur le livre "L'art des Bases de Données : Introduction aux bases de données" de S. Miranda et J-M Busta (Eyrolles) ainsi que sur les supports de cours de Rosine Cicchetti et Nicole Bidoit.

1 Structured Query Language (SQL)

- Langage de Définition de Données (LDD)
  création et modification de la structure des Bases de Données

- Langage de Manipulation de Données (LMD)
  insertion et modification des données des Bases de Données

- Langage de Contrôle des Données (LCD)
  gestion de la sécurité, confidentialité et Contraintes d'Intégrité

Petit lexique entre le modèle relationnel et SQL :

Modèle relationnel	SQL
Relation	Table
Attribut	Colonne
Tuple	Ligne

1.1 SQL LDD

1.1.1 Types syntaxiques (presque les domaines)

La notion de domaine n'est pas très présente dans les SGBD. Il nous faut donc nous limiter à la définition des types syntaxiques suivants dans la majorité des cas :

`VARCHAR2(n)`	Chaîne de caractères de longueur variable (maximum n)
`CHAR(n)`	Chaîne de caractères de longueur fixe (n caractères)
`NUMBER`	Nombre entier (40 chiffres maximum)
`NUMBER(n, m)`	Nombre de longueur totale n avec m décimales
`DATE`	Date (DD-MON-YY est le format par défaut)
`LONG`	Flot de caractères

ATTENTION : plusieurs SQL donc plusieurs syntaxes (ici ORACLE)

1.1.2 Création de table

CREATE TABLE <nom de la table> (<nom de colonne> <type> [**NOT NULL**]
[, <nom de colonne> <type>]…, [<contrainte>]…);

où <contrainte> représente la liste des contraintes d'intégrité structurelles concernant les colonnes de la table crée. Elle s'exprime sous la forme suivante :

CONSTRAINT <nom de contrainte> <sorte de contrainte>

où <sorte de contrainte> est :

- **PRIMARY KEY** (attribut1, [attribut2…])
- **FOREIGN KEY** (attribut1, [attribut2…]) **REFERENCES** <nom de table associée>(attribut1, [attribut2…])
- **CHECK** (attribut <condition>) avec <condition> qui peut être une expression booléenne "simple" ou de la forme **IN** (liste de valeurs) ou **BETWEEN** <borne inférieure> **AND** <borne supérieure>

1.1.3 Modification de la structure d'une table

Ajout :

ALTER TABLE <nom de la table> **ADD** (<nom de colonne> <type>
[, <contrainte>]…);

ALTER TABLE <nom de la table> **ADD** (<contrainte> [, <contrainte>]…);

Modification :

ALTER TABLE <nom de la table> **MODIFY**
([<nom de colonne> <nouveau type>] [,<nom de colonne>
<nouveau type>]…);

1.1.4 Consultation de la structure d'une base

DESCRIBE <nom de la table>;

1.1.5 Destruction de table

DROP TABLE <nom de la table>;

1.2 SQL LMD

1.2.1 Interrogation

SELECT [**DISTINCT**] <nom de colonne>[, <nom de colonne>]…
FROM <nom de table>[, <nom de table>]…
[WHERE <condition>**]**
[GROUP BY <nom de colonne>[, <nom de colonne>]…
[HAVING <condition avec calculs verticaux>**]]**
[ORDER BY <nom de colonne>[, <nom de colonne>]…**]**

1.2.2 Insertion de données

INSERT INTO <nom de table> [(colonne, …)] **VALUES** (valeur, …)

INSERT INTO <nom de table> [(colonne, …)] **SELECT**…

1.2.3 Modification de données

UPDATE <nom de table> **SET** colonne = valeur, … [**WHERE** condition]

UPDATE <nom de table> **SET** colonne = **SELECT**…

1.2.4 Suppression de données

DELETE FROM <nom de table> [**WHERE** condition]

2 Les clauses de SELECT

Base utilisée :

PILOTE(NUMPIL, NOMPIL, PRENOMPIL, ADRESSE, SALAIRE, PRIME)
AVION(NUMAV, NOMAV, CAPACITE, LOCALISATION)
VOL(NUMVOL, ***NUMPIL***, ***NUMAV***, DATE_VOL, HEURE_DEP,
HEURE_ARR, VILLE_DEP, VILLE_ARR).

On suppose qu'un vol, référencé par son numéro NUMVOL, est effectué par un unique pilote, de numéro NUMPIL, sur un avion identifié par son numéro NUMAV.

Rappel de la syntaxe :

SELECT [**DISTINCT**] <nom de colonne>[, <nom de colonne>]…
FROM <nom de table>[, <nom de table>]…
[WHERE <condition>**]**
[GROUP BY <nom de colonne>[, <nom de colonne>]…
[HAVING <condition avec calculs verticaux>**]]**
[ORDER BY <nom de colonne>[, <nom de colonne>]…**]**

2.1 Expression des projections

```
SELECT    *
FROM      table
```

où *table* est le nom de la relation à consulter. Le caractère * signifie qu'aucune projection n'est réalisée, i.e. que tous les attributs de la relation font partie du résultat.

Exemple : donner toutes les informations sur les pilotes.

```
SELECT    *
FROM      PILOTE
```

La liste d'attributs sur laquelle la projection est effectuée doit être précisée après la clause **SELECT**.
Exemple : donner le nom et l'adresse des pilotes.

```
SELECT    NOMPIL, ADRESSE
FROM      PILOTE
```

Alias :

```
SELECT    colonne [alias],…
```

Si alias est composé de plusieurs mots (séparés par des blancs), il faut utiliser des guillemets.

Exemple : sélectionner l'identificateur et le nom de chaque pilote.

```
SELECT  NUMPIL "Num pilote", NOMPIL Nom_du_pilote
FROM    PILOTE
```

Suppression des duplicats : mot clef SQL **DISTINCT**

Exemple : quelles sont toutes les villes de départ des vols ?

```
SELECT  DISTINCT  VILLE_DEP
FROM    VOL
```

2.2 Expression des sélections

Clause WHERE

Exemple : donner le nom des pilotes qui habitent à Marseille.

```
SELECT    NOMPIL
FROM      PILOTE
WHERE     ADRESSE = 'MARSEILLE'
```

Exemple : donner le nom et l'adresse des pilotes qui gagnent plus de 3.000 €.

```
SELECT    NOMPIL
FROM      PILOTE
WHERE     SALAIRE > 3000
```

Autres opérateurs spécifiques :

- **IS NULL** : teste si la valeur d'une colonne est une valeur nulle (inconnue).

Exemple : rechercher le nom des pilotes dont l'adresse est inconnue.

```
SELECT   NOMPIL
FROM     PILOTE
WHERE    ADRESSE  IS  NULL
```

- **IN** (liste) : teste si la valeur d'une colonne coïncide avec l'une des valeurs de la liste.

Exemple : rechercher les avions de nom A310, A320, A330 et A340.

```
SELECT   *
FROM     AVION
WHERE    NOMAV IN ('A310', 'A320', 'A330', 'A340')
```

- **BETWEEN** v1 **AND** v2 : teste si la valeur d'une colonne est comprise entre les valeurs v1 et v2 (v1 <= valeur <= v2).

Exemple : quel est le nom des pilotes qui gagnent entre 3.000 et 5.000 € ?

```
SELECT   NOMPIL
FROM     PILOTE
WHERE    SALAIRE  BETWEEN  3000  AND  5000
```

- **LIKE** 'chaîne_générique'. Le symbole **%** remplace une chaîne de caractère quelconque, y compris le vide. Le symbole **_** remplace un caractère.

Exemple : quelle est la capacité des avions de type Airbus ?

```
SELECT   CAPACITE
FROM     AVION
WHERE    NOMAV  LIKE  'A%'
```

Négation des opérateurs spécifiques : **NOT**.
Nous obtenons alors **IS NOT NULL**, **NOT IN**, **NOT LIKE** et **NOT BETWEEN**.

Exemple : quels sont les noms des avions différents de A310, A320, A330 et A340 ?

```
SELECT   NOMAV
FROM     AVION
WHERE    NOMAV  NOT IN  ('A310','A320','A330','A340')
```

Opérateurs logiques : **AND** et **OR**. Le **AND** est prioritaire et le parenthésage doit être utilisé pour modifier l'ordre d'évaluation.

Exemple : quels sont les vols au départ de Marseille desservant Paris ?

```
SELECT   *
FROM     VOL
WHERE    VILLE_DEP = 'MARSEILLE' AND VILLE_ARR =
'PARIS'
```

Requête paramétrées : les paramètres de la requêtes seront quantifiés au moment de l'exécution de la requête. Pour cela, il suffit dans le texte de la requête de substituer aux différentes constantes de sélection des symboles de variables qui doivent systématiquement commencer par le caractère &.

Remarque : si la constante de sélection est alphanumérique, on peut spécifier `'&var'` dans la requête, l'utilisateur n'aura plus qu'à saisir la valeur, sinon il devra taper 'valeur'.

Exemple : quels sont les vols au départ d'une ville et dont l'heure d'arrivée est inférieure à une certaine heure ?

```
SELECT   *
FROM     VOL
WHERE    VILLE_DEP = '&1' AND HEURE_ARR < &2
```

Lors de l'exécution de cette requête, le système demande à l'utilisateur de lui indiquer une ville de départ et une heure d'arrivée.

2.3 Calculs horizontaux

Des expressions arithmétiques peuvent être utilisées dans les clauses **WHERE** et **SELECT**.

Ces expressions de calcul horizontal sont évaluées puis affichées et/ou testées pour chaque tuple appartenant au résultat.

Exemple : donner le revenu mensuel des pilotes Bordelais.

```
SELECT   NUMPIL, NOMPIL, SALAIRE + PRIME
FROM     PILOTE
WHERE    ADRESSE = 'BORDEAUX'
```

Exemple : quels sont les pilotes qui avec une augmentation de 10% de leur prime gagnent moins de 5.000 € ? Donner leur numéro, leurs revenus actuel et simulé.

```
SELECT   NUMPIL, SALAIRE+PRIME, SALAIRE + (PRIME*1.1)
FROM     PILOTE
WHERE    SALAIRE + (PRIME*1.1) < 5000
```

Expression d'un calcul :
Les opérateurs arithmétiques +, -, *, et / sont disponibles en **SQL**. De plus, l'opérateurs || (concaténation de chaînes de caractères -ex : C1 || C2 correspond concaténation de C1 et C2). Enfin, les opérateurs + et - peuvent être utilisés pour ajouter ou soustraire un nombre de jour à une date. L'opérateur - peut être utilisé entre deux dates et rend le nombre de jours entre les deux dates arguments.

Les fonctions disponibles en SQL dépendent du SGBD. Sous ORACLE, les fonctions suivantes sont disponibles :

ABS(n)	la valeur absolue de *n* ;
FLOOR(n)	la partie entière de *n* ;
POWER(m, n)	m à la puissance *n* ;
TRUNC(n[, m])	*n* tronqué à *m* décimales après le point décimal. Si *m* est négatif, la troncature se fait avant le point décimal ;
ROUND(n [, d])	arrondit *n* à dix puissance -*d* ;
CEIL(n)	entier directement supérieur ou égal à *n* ;
MOD(n, m)	*n* modulo *m* ;
SIGN(n)	1 si $n > 0$, 0 si $n = 0$, -1 si $n < 0$;
SQRT(n)	racine carrée de *n* (NULL si $n < 0$) ;
GREATEST(n1, n2,…)	la plus grande valeur de la suite ;
LEAST(n1, n2,…)	la plus petite valeur de la liste ;
NVL(n1, n2)	permet de substituer la valeur *n2* à *n1*, au cas où cette dernière est une valeur nulle ;
LENGTH(ch)	longueur de la chaîne ;
SUBSTR(ch, pos [, long])	extraction d'une sous-chaîne de *ch* à partir de la position *pos* en donnant le nombre de caractères à extraire *long* ;
INSTR(ch, ssch [, pos [, n]])	position de la sous-chaîne dans la chaîne ;
UPPER(ch)	mise en majuscules ;
LOWER(ch)	mise en minuscules ;
INITCAP(ch)	initiale en majuscules ;
SOUNDEX(ch)	comparaison phonétique ;
LPAD(ch, long [, car])	complète *ch* à gauche à la longueur *long* par *car* ;
RPAD(ch, long [, car])	complète *ch* à droite à la longueur *long* par *car* ;
LTRIM(ch, car)	élague à gauche *ch* des caractères *car* ;
RTRIM(ch, car)	élague à droite *ch* des caractères *car* ;
TRANSLATE(ch, car_source, car_cible)	change *car_source* par *car_cible* ;
TO_CHAR(nombre)	convertit un nombre ou une date en chaîne de caractères ;
TO_NUMBER(ch)	convertit la chaîne en numérique ;
ASCII(ch)	code ASCII du premier caractère de la chaîne ;
CHR(n)	conversion en caractère d'un code ASCII ;
TO_DATE(ch[, fmt])	conversion d'une chaîne de caractères en date ;
ADD_MONTHS(date, nombre)	ajout d'un nombre de mois à une date ;
MONTHS_BETWEEN(date1, date2)	nombre de mois entre *date1* et *date2* ;
LAST_DAY(date)	date du dernier jour du mois ;

NEXT_DAY(date, nom du jour)	date du prochain jour de la semaine ;
SYSDATE	date du jour.

Exemple : donner la partie entière des salaires des pilotes.

```
SELECT NOMPIL,FLOOR(SALAIRE) "Salaire:partie entière"
FROM   PILOTE
```

ATTENTION : Tous les SGBD n'évaluent pas correctement les expressions arithmétiques que si les valeurs de ses arguments ne sont pas **NULL**. Pour éviter tout problème, il convient d'utiliser la fonction **NVL** (décrite ci avant) qui permet de substituer une valeur par défaut aux valeurs nulles éventuelles.

Exemple : l'attribut Prime pouvant avoir des valeurs nulles, la requête donnant le revenu mensuel des pilotes toulousains doit se formuler de la façon suivante.

```
SELECT   NUMPIL, NOMPIL, SALAIRE + NVL(PRIME,0)
FROM     PILOTE
WHERE    ADRESSE = 'BORDEAUX'
```

Pour plus d'information, vous pouvez consulter cette Illustration de la gestion des valeurs nulles par différents SGBD (http://dbforums.com/showthread.php?threadid=421723).

2.4 Calculs verticaux (fonctions agrégatives)

Les fonctions agrégatives s'appliquent à un ensemble de valeurs d'un attribut de type numérique sauf pour la fonction de comptage **COUNT** pour laquelle le type de l'attribut argument est indifférent.

Contrairement aux calculs horizontaux, le résultat d'une fonction agrégative est évalué une seule fois pour tous les tuples du résultat[1].

Les fonctions agrégatives disponibles sont généralement les suivantes :

SUM	somme,
AVG	moyenne arithmétique,
COUNT	nombre ou cardinalité,
MAX	valeur maximale,
MIN	valeur minimale,
STDDEV	écart type,
VARIANCE	variance.

Chacune de ces fonctions a comme argument un nom d'attribut ou une expression arithmétique. Les valeurs nulles ne posent pas de problème d'évaluation.
La fonction **COUNT** peut prendre comme argument le caractère `*`, dans ce cas, elle rend comme résultat le nombre de lignes sélectionnées par le bloc.

Exemple : quel est le salaire moyen des pilotes Marseillais.

```
SELECT   AVG(SALAIRE)
FROM     PILOTE
WHERE    ADRESSE = 'MARSEILLE'
```

Exemple : trouver le nombre de vols au départ de Marseille.

```
SELECT   COUNT(NUMVOL)
FROM     VOL
WHERE    VILLE_DEP = 'MARSEILLE'
```

Dans cette requête, **COUNT**`(*)` peut être utiliser à la place de **COUNT**`(NUMVOL)`

La colonne ou l'expression à laquelle est appliquée une fonction agrégative peut avoir des valeurs redondantes. Pour indiquer qu'il faut considérer seulement les valeurs distinctes, il faut faire précéder la colonne ou l'expression de **DISTINCT**.

Exemple : combien de destinations sont desservies au départ de Bordeaux ?

```
SELECT   COUNT(DISTINCT VILLE_ARR)
FROM     VOL
WHERE    VILLE_DEP = 'BORDEAUX'
```

[1] En cas de partitionnement, cette évaluation est faite pour chaque classe d'équivalence de tuples.

2.5 Expression des jointures sous forme prédicative

La clause **FROM** contient tous les noms de tables à fusionner. La clause **WHERE** exprime le critère de jointure sous forme de condition.

ATTENTION : si deux tables sont mentionnées dans la clause **FROM** et qu'aucune jointure n'est spécifiée, le système effectue le produit cartésien des deux relations (chaque tuple de la première est mis en correspondance avec tous les tuples de la seconde), le résultat est donc faux car les liens sémantiques entre relations ne sont pas utilisés. Donc respectez et vérifiez la règle suivante.

Si n tables apparaissent dans la clause **FROM**, il faut au moins (n – 1) opérations de jointure.

La forme générale d'une requête de jointure est :

```
SELECT   colonne,…
FROM     table1, table2…
WHERE    <condition>
```

où <*condition*> est de la forme `attribut1 θ attribut2` et θ est un opérateur de comparaison.

Exemple : quel est le numéro et le nom des pilotes résidant dans la ville de localisation de l'avion n° 33 ?

```
SELECT   NUMPIL, NOMPIL
FROM     PILOTE, AVION
WHERE    ADRESSE = LOCALISATION AND NUMAV = 33
```

Les noms des colonnes dans la clause **SELECT** et **WHERE** doivent être uniques. Ne pas oublier de lever l'ambiguïté à l'aide d'alias (cf. auto-jointure) ou en préfixant les noms de colonnes.

Exemple : donner le nom des pilotes faisant des vols au départ de Marseille sur des Airbus ?

```
SELECT   DISTINCT NOMPIL
FROM     PILOTE, VOL, AVION
WHERE    VILLE_DEP = 'MARSEILLE'
     AND NOMAV LIKE 'A%'
     AND PILOTE.NUMPIL = VOL.NUMPIL
     AND VOL.NUMAV = AVION. NUMAV
```

Alias de nom de table : se place dans la clause **FROM** après le nom de la table.

Exemple : quels sont les avions localisés dans la même ville que l'avion numéro 103 ?

```
SELECT   AUTRES.NUMAV, AUTRES.NOMAV
FROM     AVION AUTRES, AVION AV103
WHERE    AV103.NUMAV = 103
     AND AUTRES.NUMAV <> 103
     AND AV103.LOCALISATION = AUTRES.LOCALISATION
```

Dans cette requête l'alias AV103 est utilisée pour retrouver l'avion de numéro 103 et l'alias AUTRES permet de balayer tous les tuples de AVION pour faire la comparaison des localisations.

Exemple : quelles sont les correspondances (villes d'arrivée) accessibles à partir de la ville d'arrivée du vol IT100 ?

```
SELECT   DISTINCT AUTRES.VILLE_ARR
FROM     VOL AUTRES, VOL VOLIT100
WHERE    VOLIT100.NUMVOL = 'IT100'  AND
         VOLIT100.VILLE_ARR = AUTRES.VILLE_DEP
```

La requête recherche les vols dont la ville de départ correspond à la ville d'arrivée du vol IT100 puis ne conserve que les villes d'arrivée de ces vols.

2.6 Autre expression de jointures : forme imbriquée

1. **Premier cas :**

Le résultat de la sous-requête est formé *d'une seule valeur*. C'est surtout le cas de sous-requêtes utilisant une fonction agrégat dans la clause **SELECT**. L'attribut spécifié dans le **WHERE** est simplement comparé au résultat du **SELECT** imbriqué à l'aide de l'opérateur de comparaison voulu.

Exemple : quel est le nom des pilotes gagnant plus que le salaire moyen des pilotes ?

```
SELECT   NOMPIL
FROM     PILOTE
WHERE    SALAIRE > (SELECT AVG(SALAIRE)  FROM PILOTE)
```

2. Deuxième cas :

Le résultat de la sous-requête est formé *d'une liste de valeurs*. Dans ce cas, le résultat de la comparaison, dans la clause **WHERE**, peut être considéré comme vrai :

- soit si la condition doit être vérifiée avec *au moins une des valeurs résultats* de la sous-requête. Dans ce cas, la sous-requête doit être précédée du comparateur suivi de **ANY** (remarque : = **ANY** est équivalent à **IN**) ;

- soit si la condition doit être vérifiée pour *toutes les valeurs résultats* de la sous-requête, alors la sous-requête doit être précédée du comparateur suivi de **ALL**.

Exemple : quels sont les noms des pilotes en service au départ de Marseille ?

```
SELECT   NOMPIL
FROM     PILOTE
WHERE    NUMPIL IN
                  (SELECT DISTINCT NUMPIL
                  FROM     VOL
                  WHERE    VILLE_DEP = 'MARSEILLE')
```

Exemple : quels sont les numéros des avions localisés à Marseille dont la capacité est supérieure à celle de l'un des appareils effectuant un Paris-Marseille ?

```
SELECT   NUMAV
FROM     AVION
WHERE    LOCALISATION = 'MARSEILLE' AND CAP > ANY
                  (SELECT DISTINCT CAP FROM AVION
                  WHERE    NUMAV = ANY
                       (SELECT     DISTINCT NUMAV
                       FROM        VOL
                       WHERE       VILLE-DEP = 'PARIS'
                           AND VILLE_ARR = 'MARSEILLE'
          ) )
```

Exemple : quels sont les noms des pilotes Marseillais qui gagnent plus que tous les pilotes parisiens ?

```
SELECT   NOMPIL
FROM     PILOTE
WHERE    ADRESSE = 'MARSEILLE'  AND SALAIRE >  ALL
         (SELECT DISTINCT SALAIRE
         FROM     PILOTE
         WHERE    ADRESSE = 'PARIS')
```

Exemple : donner le nom des pilotes Marseillais qui gagnent plus qu'un pilote parisien.

```
SELECT   NOMPIL
FROM     PILOTE
WHERE    ADRESSE = 'MARSEILLE'  AND SALAIRE >  ANY
         (SELECT  SALAIRE
         FROM     PILOTE
         WHERE    ADRESSE = 'PARIS')
```

3. Troisième cas :

- Le résultat de la sous-requête est un *ensemble de colonnes*.
- le nombre de colonnes de la clause **WHERE** doit être identique à celui de la clause **SELECT** de la sous-requête. Ces colonnes doivent être mises entre parenthèses,
- la comparaison se fait entre les valeurs des colonnes de la requête et celle de la sous-requête deux à deux,
- il faut que l'opérateur de comparaison soit le même pour tous les attributs concernés.

Exemple : rechercher le nom des pilotes ayant même adresse et même salaire que Dupont.

```
SELECT   NOMPIL
FROM     PILOTE
WHERE    NOMPIL  <> 'DUPONT'
     AND (ADRESSE, SALAIRE)  IN
               (SELECT     ADRESSE, SALAIRE
               FROM        PILOTE
               WHERE       NOMPIL = 'DUPONT')
```

2.7 Tri des résultats

Il est possible avec SQL d'ordonner les résultats. Cet ordre peut être croissant (**ASC**) ou décroissant (**DESC**) sur une ou plusieurs colonnes ou expressions.

```
ORDER BY expression [ASC | DESC],…
```

L'argument de **ORDER BY** peut être un nom de colonne ou une expression basée sur une ou plusieurs colonnes mentionnées dans la clause **SELECT**.

Exemple : En une seule requête, donner la liste des pilotes Marseillais par ordre de salaire décroissant et par ordre alphabétique des noms.

```
SELECT   NOMPIL, SALAIRE
FROM     PILOTE
WHERE    ADRESSE = 'MARSEILLE'
ORDER BY SALAIRE  DESC, NOMPIL
```

2.8 Test d'absence de données

Pour vérifier qu'une donnée est absente dans la base, le cas le plus simple est celui où l'existence de tuples est connue et on cherche si un attribut a des valeurs manquantes. Il suffit alors d'utiliser le prédicat **IS NULL**. Mais cette solution n'est correcte que si les tuples existent. Elle ne peut en aucun cas s'appliquer si on cherche à vérifier l'absence de tuples ou l'absence de valeur quand on ne sait pas si des tuples existent.

Exemple : Les requêtes suivantes ne peuvent pas être formulées avec **IS NULL**.
Quels sont les pilotes n'effectuant aucun vol ? Quelles sont les villes de départ dans lesquelles aucun avion n'est localisé ?
Pour formuler des requêtes de type " un élément n'appartient pas à un ensemble donné ", trois techniques peuvent être utilisées. La première consiste à utiliser une jointure imbriquée avec **NOT IN**. La sous-requête est utilisée pour calculer l'ensemble de recherche et le bloc de niveau supérieur extrait les éléments n'appartenant pas à cet ensemble.
Exemple : quels sont les pilotes n'effectuant aucun vol ?

```
SELECT   NUMPIL, NOMPIL
FROM     PILOTE
WHERE    NUMPIL   NOT IN
              (SELECT NUMPIL   FROM     VOL)
```

Une deuxième approche fait appel au prédicat **NOT EXISTS** qui s'applique à un bloc imbriqué et rend la valeur vrai si le résultat de la sous-requête est vide (et faux sinon). Il faut faire très attention à ce type de requête car **NOT EXISTS** ne dispense pas d'exprimer des jointures. Ce danger est détaillé à travers l'exemple suivant.

Exemple : reprenons la requête précédente. La formulation suivante est fausse.

```
SELECT   NUMPIL, NOMPIL
FROM     PILOTE
WHERE    NOT EXISTS
              (SELECT NUMPIL   FROM     VOL)
```

Il suffit qu'un vol soit créé dans la relation VOL pour que la requête précédente retourne systématiquement un résultat vide. En effet le bloc imbriqué rendra une valeur pour NUMPIL et le **NOT EXISTS** sera toujours évalué à faux, donc aucun pilote ne sera retourné par le premier bloc.
Le problème de cette requête est que le lien entre les éléments cherchés dans les deux blocs n'est pas spécifié. Or, il faut indiquer au système que le 2ème bloc doit être évalué pour chacun des pilotes examinés par le 1er bloc. Pour cela, on introduit un alias pour la relation PILOTE et une jointure est exprimée dans le 2ème bloc en utilisant cet alias. La formulation correcte est la suivante :

```
SELECT   NUMPIL, NOMPIL
FROM     PILOTE  PIL
WHERE    NOT EXISTS
              (SELECT     NUMPIL
              FROM        VOL
              WHERE       VOL.NUMPIL = PIL.NUMPIL)
```

Enfin, il est possible d'avoir recours à l'opérateur de différence (**MINUS**).

2.9 Classification ou partitionnement

La classification permet de regrouper les lignes d'une table dans des classes d'équivalence ou sous-tables ayant chacune la même valeur pour la colonne de la classification. Ces classes forment une partition de l'extension de la relation considérée (i.e. l'intersection des classes est vide et leur union est égale à la relation initiale).

Exemple : considérons la relation VOL illustrée par la figure 1. Partitionner cette relation sur l'attribut NUMPIL consiste à regrouper au sein d'une même classe tous les vols assurés par le même pilote. NUMPIL prenant 4 valeurs distinctes, 4 classes sont introduites. Elles sont mises en évidence dans la figure 2 et regroupent respectivement les vols des pilotes n° 100, 102, 105 et 124.

NUMVOL	NUMPIL	…
IT100	100	
AF101	100	
IT101	102	
BA003	105	
BA045	105	
IT305	102	
AF421	124	
BA047	105	
BA087	105	

Figure 1 : relation exemple VOL

NUMVOL	NUMPIL	…
IT100	100	
AF101	100	
IT101	102	
IT305	102	
BA003	105	
BA045	105	
BA047	105	
BA087	105	
AF421	124	

Figure 2 : regroupement selon NUMPIL

En SQL, l'opérateur de partitionnement s'exprime par la clause **`GROUP BY`** qui doit suivre la clause **`WHERE`** (ou **`FROM`** si **`WHERE`** est absente). Sa syntaxe est :

`GROUP BY` *colonne1*, *[colonne2,…]*

Les colonnes indiquées dans **`SELECT`**, sauf les attributs arguments des fonctions agrégatives, doivent être mentionnées dans **`GROUP BY`**.

Il est possible de faire un regroupement multi-colonnes en mentionnant plusieurs colonnes dans la clause **`GROUP BY`**. Une classe est alors créée pour chaque combinaison distincte de valeurs de la combinaison d'attributs.

En présence de la clause **`GROUP BY`**, les fonctions agrégatives s'appliquent à l'ensemble des valeurs de chaque classe d'équivalence.

Exemple : quel est le nombre de vols effectués par chaque pilote ?

```
SELECT   NUMPIL, COUNT(NUMVOL)
FROM     VOL
GROUP BY NUMPIL
```

Dans ce cas, le résultat de la requête comporte une ligne par numéro de pilote présent dans la relation VOL.

Exemple : combien de fois chaque pilote conduit-il chaque avion ?

```
SELECT   NUMPIL, NUMAV, COUNT(NUMVOL)
FROM     VOL
GROUP BY NUMPIL, NUMAV
```

Nous obtenons ici autant de lignes par pilote qu'il y a d'avions distincts conduits le pilote considéré. Chaque classe d'équivalence créée par le ***GROUP BY*** *regroupe tous les vols ayant même pilote et même avion.*

Partitionner une relation sur un attribut clef primaire ou clef candidate est évidemment complètement inutile : chaque classe d'équivalence est réduite à un seul tuple !
De même, mentionner un **DISTINCT** devant les attributs de la clause **SELECT** ne sert à rien si le bloc comporte un **GROUP BY** : ces attributs étant aussi les attributs de partitionnement, il n'y aura pas de duplicats parmi les valeurs ou combinaisons de valeurs retournées. Il faut par contre faire très attention à l'argument des fonctions agrégatives. L'oubli d'un **DISTINCT** peut rendre la requête fausse.

Exemple : donner le nombre de destinations desservies par chaque avion.

```
SELECT   NUMAV, COUNT(DISTINCT VILLE_ARR)
FROM     VOL
GROUP BY NUMAV
```

Si le **DISTINCT** est oublié, c'est le nombre de vols qui sera compté et la requête sera fausse.

2.10 Recherche dans les sous-tables

Des conditions de sélection peuvent être appliquées aux sous-tables engendrées par la clause **GROUP BY**, comme c'est le cas avec la clause **WHERE** pour les tables. Cette sélection s'effectue avec la clause **HAVING** qui doit suivre la clause **GROUP BY**. Sa syntaxe est :

```
HAVING condition
```

La condition permet de comparer une valeur obtenue à partir de la sous-table à une constante ou à une autre valeur résultant d'une sous-requête.

Exemple : donner le nombre de vols, s'il est supérieur à 5, par pilote.

```
SELECT   NUMPIL, NOMPIL, COUNT(NUMVOL)
FROM     PILOTE, VOL
WHERE    PILOTE.NUMPIL = VOL.NUMPIL
GROUP BY NUMPIL, NOMPIL
HAVING   COUNT(NUMVOL) > 5
```

Exemple : quelles sont les villes à partir desquelles le nombre de villes desservies est le plus grand ?

```
SELECT        VILLE_DEP
FROM          VOL
GROUP BY      VILLE_DEP
HAVING        COUNT(DISTINCT VILLE_ARR) >= ALL
                   (SELECT  COUNT(DISTINCT VILLE_ARR)
                   FROM     VOL
                   GROUP BY     VILLE_DEP)
```

Même si les clauses **WHERE** et **HAVING** introduisent des conditions de sélection mais elles sont différentes. Si la condition doit être vérifiée par chaque tuple d'une classe d'équivalence, il faut la spécifier dans le **WHERE**. Si la condition doit être vérifiée globalement pour la classe, elle doit être exprimée dans la clause **HAVING**.

2.11 Recherche dans une arborescence

La consultation des données de structure arborescente est une spécificité du SGBD ORACLE. Une telle structure de données est très fréquente dans le monde réel. Elle correspond à des compositions ou des liens hiérarchiques.

Exemple : supposons que l'on souhaite intégrer dans la base des informations sur la répartition géographique des villes desservies. La figure 3 illustre l'organisation hiérarchique des valeurs des différentes localisations à répertorier.

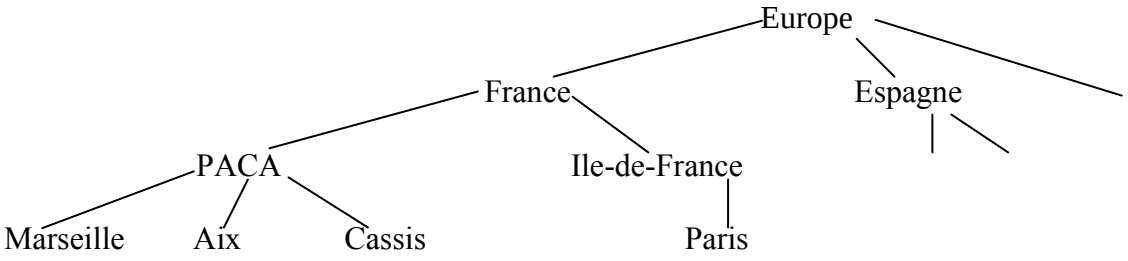

Figure 3 : hiérarchie des localisations

La relation LIEU est ajoutée au schéma initial de la base pour représenter de manière " plate " la structure arborescente donnée ci-dessus. Cette relation a le schéma suivant : LIEU (LIEUSPEC, ***LIEUGEN***).

L'attribut LIEUGEN est défini sur le même domaine que LIEUSPEC. C'est donc une clef étrangère référençant la clef primaire de sa propre relation.

L'extension de cette relation, correspondant à la hiérarchie précédente, est la suivante :

LIEUSPEC	LIEUGEN
EUROPE	NULL
FRANCE	EUROPE
ESPAGNE	EUROPE
PACA	FRANCE
ILE-DE-FRANCE	FRANCE
MARSEILLE	PACA
AIX	PACA
CASSIS	PACA
PARIS	ILE-DE-FRANCE

Figure 4 : représentation relationnelle de la hiérarchie des localisations

La syntaxe générale d'une requête de recherche dans une arborescence est :

```
SELECT                  colonne,…
FROM                    table [alias],…
[WHERE                  condition]
CONNECT BY [PRIOR]      colonne1 = [PRIOR] colonne 2
[AND                    condition]
[START WITH             condition]
[ORDER  BY  LEVEL]
```

La clause **START WITH** fixe le point de départ de la recherche dans l'arborescence.

La clause **CONNECT BY PRIOR** fixe le sens de parcours de l'arborescence. Ce parcours peut se faire :

- de haut en bas, i.e. qu'il correspond à la recherche de sous-arbres ;
- de bas en haut, i.e. qu'il correspond à la recherche du ou des " ancêtres ".

Le parcours de haut en bas est spécifié en faisant précéder la colonne représentant un " fils " dans l'arborescence par le mot-clef **PRIOR** comme suit :

```
     CONNECT  BY  PRIOR  <fils> = <père>
ou   CONNECT  BY  <père> = PRIOR  <fils>
```

Le parcours de bas en haut est spécifié en faisant précéder la colonne représentant un " père " par le mot-clef **PRIOR**. Il peut s'exprimer aussi de deux façons :

```
     CONNECT  BY  <fils> = PRIOR  <père>
ou   CONNECT  BY  PRIOR <père> = <fils>
```

Remarque : avec la représentation relationnelle plate, le <fils> dans une structure hiérarchique correspond à la clef primaire de la relation, le <père> à la clef étrangère. Lorsque la valeur de l'attribut <fils> est la racine de la hiérarchie, la valeur de l'attribut <père> est une valeur nulle : il s'agit d'un cas exceptionnel où l'on admettra une valeur nulle pour une clef étrangère.

Exemple : donner la liste des villes desservies en France.

```
SELECT                  LIEUSPEC
FROM                    LIEU
CONNECT BY  PRIOR       LIEUSPEC = LIEUGEN
START WITH              LIEUGEN = 'FRANCE'
```

Exemple : où se situe (continent, pays, région) la ville de Valladolid ?

```
SELECT                  LIEUSPEC
FROM                    LIEU
CONNECT BY              LIEUSPEC = PRIOR  LIEUGEN
START WITH              LIEUSPEC = 'VALLADOLID'
```

La clause facultative [**AND** condition] permet de spécifier une condition évaluée lors de la recherche dans un sous-arbre. Plus précisément, si la condition n'est pas satisfaite pour un nœud de la hiérarchie alors qu'aucun de ses descendants ne sera plus examiné.

Exemple : donner la liste des villes desservies en France en dehors de la région AQUITAINE.

```
SELECT                  LIEUSPEC
FROM                    LIEU
CONNECT BY  PRIOR       LIEUSPEC = LIEUGEN
AND                     LIEUGEN <> 'AQUITAINE'
START WITH              LIEUGEN = 'FRANCE'
ORDER  BY  LEVEL
```

Dès qu'un nœud ne vérifiant pas la condition donnée est trouvé, le sous-arbre dont ce nœud est racine est éliminé de la recherche, donc la région AQUITAINE et toutes les villes de cette région n'appartiendront pas au résultat.

ORACLE associe une numérotation aux nœuds d'une arborescence : c'est la notion de niveau (**LEVEL**). En fait le premier nœud répondant à la requête est considéré de niveau 1, ses fils (dans le cas d'une recherche de haut en bas) ou son père (dans le cas d'une recherche de bas en haut) sont considérés de niveau 2, et ainsi de suite.
Cette numérotation peut être utilisée pour classer les résultats (clause **ORDER BY LEVEL**) et surtout pour en donner une meilleure présentation. Celle-ci s'obtient en utilisant la fonction **LPAD** qui permet une indentation.

Exemple : pour obtenir la liste des villes desservies en France mais avec présentation plus claire, nous pouvons formuler la requête comme suit :

```
SELECT            LPAD('-', 2*LEVEL,' ')|| LIEUSPEC
FROM              LIEU
CONNECT BY PRIOR LIEUSPEC = LIEUGEN
START WITH        LIEUGEN = 'FRANCE'
ORDER BY LEVEL
```

|| est le symbole de concaténation de deux chaînes de caractères.
La fonction **LPAD** permet d'ajouter à la chaîne '-', un nombre d'espaces égal au double du niveau, devant chaque valeur de LIEUSPEC pour mieux visualiser la division géographique.

2.12 Expression des divisions

La division consiste à rechercher les tuples qui se trouvent associés à tous les tuples d'un ensemble particulier (le diviseur). SQL ne propose pas d'opérateur de division (quantificateur universel "tous les"). Il est néanmoins possible d'exprimer cette opération mais de manière peu naturelle. Les deux techniques utilisées consistent à reformuler la requête différemment, on parle de paraphrasage. Elles ne sont pas systématiquement applicables (suivant l'interrogation posée).

La première technique a recours aux partitionnements et fonctions agrégatives. Pour savoir si un tuple t est associé à tous les tuples du diviseur, l'idée est de compter d'une part le nombre de tuples associés à t et d'autre part le nombre de tuples du diviseur puis de comparer les valeurs obtenues.

Exemple : quels sont les numéros des pilotes qui conduisent tous les avions de la compagnie ?
Pour l'exprimer en SQL avec la première technique exposée, cette requête est traduite de la manière suivante : " Quels sont les pilotes qui conduisent autant d'avions que la compagnie en possède ? ".

```
SELECT        NUMPIL
FROM          VOL
GROUP  BY     NUMPIL
HAVING        COUNT(DISTINCT  NUMAV) =
                        (SELECT  COUNT(NUMAV)
                        FROM     AVION)
```

Le comptage dans la clause **HAVING** permet pour chaque pilote de dénombrer les appareils conduits. L'oubli du **DISTINCT** rend la requête fausse (on compterait alors le nombre de vols assurés).

Cette technique de paraphrasage ne peut être utilisée que si les deux ensembles dénombrés sont parfaitement comparables.

La deuxième forme d'expression des divisions est plus délicate à formuler. Elle se base sur une double négation de la requête et utilise la clause **`NOT EXISTS`**. Au lieu d'exprimer qu'un tuple t doit être associé à tous les tuples du diviseur pour appartenir au résultat, on recherche t tel qu'il n'existe aucun tuple du diviseur qui ne lui soit pas associé. Rechercher des tuples associés ou non, signifie concrètement effectuer des opérations de jointure.

Exemple : quels sont les numéros des pilotes qui conduisent tous les avions de la compagnie ?
Pour l'exprimer en SQL, cette requête est traduite par : "Existe-t-il un pilote tel qu'il n'existe aucun avion de la compagnie qui ne soit pas conduit par ce pilote ?".

```
SELECT   NUMPIL
FROM     PILOTE  P
WHERE    NOT EXISTS   (SELECT  *
         FROM     AVION A
         WHERE    NOT EXISTS
              (SELECT  *
              FROM     VOL
              WHERE    VOL.NUMAV = A.NUMAV  AND
                       VOL.NUMPIL = P.NUMPIL ))
```

Détaillons l'exécution de cette requête. Pour chaque pilote examiné par le 1er bloc, les différents tuples de AVION sont balayés au niveau du 2ème bloc et pour chaque avion, les conditions de jointure du 3ème bloc sont évaluées. Supposons que les trois relations de la base sont réduites aux tuples présentés dans la figure suivante.

PILOTE

NUMPIL	…
100	
101	
102	

AVION

NUMAV	…
10	
21	

VOL

NUMVOL	NUMPIL	NUMAV
IT100	100	10
IT200	100	10
AF214	101	21
AF321	102	10
AF036	101	10

Figure 5 : une instance de la base

Considérons le pilote n° 100. Parcourons les tuples de la relation AVION. Pour l'avion n° 10, le 3ème bloc retourne un résultat (le vol IT100), **NOT EXISTS** est donc faux et l'avion n° 10 n'appartient pas au résultat du 2ème bloc. L'avion n° 21 n'étant jamais piloté par le pilote 100, le 3ème bloc ne rend aucun tuple, le **NOT EXISTS** associé est donc évalué à vrai. Le 2ème bloc rend donc un résultat non vide (l'avion n° 21) et donc le **NOT EXISTS** du 1er bloc est faux. Le pilote n° 100 n'est donc pas retenu dans le résultat de la requête.
Pour le pilote 101 et l'avion 10, il existe un vol (AF214). Le 3ème bloc retourne un résultat, **NOT EXISTS** est donc faux. Pour le même pilote et l'avion n° 21, le 3ème bloc restitue un tuple et à nouveau **NOT EXISTS** est faux. Le 2ème bloc rend donc un résultat vide ce qui fait que le **NOT EXISTS** du 1er bloc est évalué à vrai. Le pilote 101 fait donc partie du résultat de la requête. Le processus décrit est à nouveau exécuté pour le pilote 102 et comme il ne pilote pas l'avion 21, le 2ème bloc retourne une valeur et la condition du 1er bloc élimine ce pilote du résultat.

2.13 Gestion des transactions

Comme tout SGBD, ORACLE est un système transactionnel, i.e. il gère les accès concurrents de multiples utilisateurs à des données fréquemment mises à jour et soumises à diverses contraintes. Une transaction est une séquence d'opérations manipulant les données. Exécutée sur une base cohérente, elle laisse la base dans un état cohérent : toutes les opérations sont réalisées ou aucune.
Les commandes **COMMIT** et **ROLLBACK** de SQL permettent de contrôler la validation et l'annulation des transactions :

- **COMMIT** : valide la transaction en cours. Les modifications de la base deviennent définitives ;
- **ROLLBACK** : annule la transaction en cours. La base est restaurée dans son état au début de la transaction.

Une modification directe de la base (**INSERT**, **UPDATE** ou **DELETE**) ne sera validée que par la commande **COMMIT**, ou lors de la sortie normale de sqlplus par la commande **EXIT**.

Il est possible de se mettre en mode "validation automatique", dans ce cas, la validation est effectuée automatiquement après chaque commande SQL de mise à jour de données. Pour se mettre dans ce mode, il faut utiliser l'une des commandes suivantes :
SET AUTOCOMMIT IMMEDIATE ou **SET AUTOCOMMIT ON**
L'annulation de ce mode se fait avec la commande : **SET AUTOCOMMIT OFF**

3 SQL comme langage de contrôle des données

3.1 Création d'index

Pour accélérer les accès aux tuples, la création d'index peut être réalisée pour un ou plusieurs attributs fréquemment consultés. La commande à utiliser est la suivante :

```
CREATE [UNIQUE] [NOCOMPRESS] INDEX  <nom_index>
ON <nom_table> (<nom_colonne>, [nom_colonne]…)
```

Lorsque le mot-clef **UNIQUE** est précisé, l'attribut (ou la combinaison) indexé doit avoir des valeurs uniques.

Une fois qu'ils ont été créés, les index sont automatiquement utilisés par le système et de manière transparente pour l'utilisateur.

Remarque : lors de la spécification de la contrainte **PRIMARY KEY** pour un ou plusieurs attributs ORACLE génère automatiquement un index primaire (**UNIQUE**) pour le(s) attribut(s) concerné(s).

Exemple :

```
CREATE INDEX  ACCES_PILOTE  ON PILOTE (NOMPIL)
```

3.2 Création et utilisation des vues

Une vue permet de consulter et manipuler des données d'une ou de plusieurs tables. On considère qu'une vue est une table virtuelle car elle peut être utilisée de la même façon qu'une relation mais ses données (redondantes par rapport à la base originale) ne sont pas physiquement stockées. Plus précisément, une vue est définie sous forme d'une requête d'interrogation et c'est cette requête qui est conservée dans le dictionnaire de la base.
L'utilisation des vues permet de :

- restreindre l'accès à certaines colonnes ou certaines lignes d'une table pour certains utilisateurs (confidentialité) ;
- simplifier la tâche de l'utilisateur en le déchargeant de la formulation de requêtes complexes ;
- contrôler la mise à jour des données.

La commande de création d'une vue est la suivante :

```
CREATE VIEW nom_vue [(colonne,…)]
AS
<requête>
[WITH CHECK OPTION]
```

Si une liste de noms d'attributs est précisée après le nom de la vue, ces noms seront ceux des attributs de la vue. Ils correspondent, deux à deux, avec les attributs indiqués dans le ***SELECT*** *de la requête définissant la vue. Si la liste de noms n'apparaît pas, les attributs de la vue ont le même nom que ceux attributs indiqués dans le* ***SELECT*** *de la requête.*

Il est très important de savoir comment la vue à créer sera utilisée : consultation simplement et/ou mises à jour.

Exemple : pour éviter que certains utilisateurs aient accès aux salaire et prime des pilotes, la vue suivante est définie à leur intention et ils n'ont pas de droits sur la relation PILOTE.

```
CREATE VIEW  RESTRICT_PIL
AS
SELECT       NUMPIL, NOMPIL, PRENOM_PIL, ADRESSE
FROM         PILOTE
```

Exemple : pour épargner aux utilisateurs la formulation d'une requête complexe, une vue est définie par les développeurs pour consulter la charge horaire des pilotes. Sa définition est la suivante :

```
CREATE VIEW  CHARGE_HOR (NUMPIL, NOM, CHARGE)
AS
SELECT   P.NUMPIL, NOMPIL, SUM(HEURE_ARR - HEURE_DEP)
FROM     PILOTE  P, VOL
WHERE    P.NUMPIL = VOL.NUMPIL
GROUP BY P.NUMPIL, NOMPIL
```

Lorsque cette vue est créée, les utilisateurs peuvent la consulter simplement par :

```
SELECT   *
FROM     CHARGE_HOR
```

Un utilisateur ne s'intéressant qu'aux pilotes parisiens dont la charge excède un seuil de 40 heures formulera la requête suivante.

```
SELECT   *
FROM     CHARGE_HOR C, PILOTE P
WHERE    C.NUMPIL = P.NUMPIL AND CHARGE > 40
     AND ADRESSE = 'PARIS'
```

Lorsque le système évalue une requête formulée sur une vue, il combine la requête de l'utilisateur et la requête de définition de la vue pour obtenir le résultat.

Lorsqu'une vue est utilisée pour effectuer des opérations de mise à jour, elle est soumise à des contraintes fortes. En effet pour que les mises à jour, à travers une vue, soient automatiquement répercutées sur la relation de base associée, il faut impérativement que :

- la vue ne comprenne pas de clause **GROUP BY**.
- la vue n'ait qu'une seule relation dans la clause **FROM**. Ceci implique que dans une vue multi-relation, les jointures soient exprimées de manière imbriquée.

Lorsque la requête de définition d'une vue comporte une projection sur un sous-ensemble d'attributs d'une relation, les attributs non mentionnés prendront des valeurs nulles en cas d'insertion à travers la vue.

Exemple : définir une vue permettant de consulter les vols des pilotes habitant Bayonne et de les mettre à jour.

```
CREATE VIEW  VOLPIL_BAYONNE
AS
SELECT       *
FROM         VOL
WHERE        NUMPIL  IN
              (SELECT  NUMPIL
              FROM PILOTE
              WHERE  ADRESSE = 'BAYONNE')
```

La vue précédente permet la consultation uniquement des vols vérifiant la condition donnée sur le pilote associé. Il est également possible de mettre à jour la relation VOL à travers cette vue mais l'opération de mise à jour peut concerner n'importe quel vol (sans aucune condition).
Par exemple supposons que le pilote n° 100 habite Paris, l'insertion suivante sera réalisée dans la relation VOL à travers la vue, mais le tuple ne pourra pas être visible en consultant la vue.

```
INSERT INTO VOLPIL_BAYONNE
     (NUMVOL,NUMPIL,NUMAV,VILLE_DEP)
     VALUES ('IT256', 100, 14, 'PARIS')
```

Si la clause **WITH CHECK OPTION** est présente dans l'ordre de création d'une vue, la table associée peut être mise à jour, avec vérification des conditions présentes dans la requête définissant la vue. La vue joue alors le rôle d'un filtre entre l'utilisateur et la table de base, ce qui permet la vérification de toute condition et notamment des contraintes d'intégrité.

Exemple : définir une vue permettant de consulter et de les mettre à jour uniquement les vols des pilotes habitant Bayonne.

```
CREATE VIEW  VOLPIL_BAYONNE
AS
SELECT       *
FROM         VOL
WHERE        NUMPIL  IN
             (SELECT  NUMPIL
             FROM PILOTE
             WHERE  ADRESSE = 'BAYONNE')
WITH CHECK OPTION
```

L'ajout de la clause **WITH CHECK OPTION** à la requête précédente interdira toute opération de mise à jour sur les vols qui ne sont pas assurés par des pilotes habitant Bayonne et l'insertion d'un vol assuré par le pilote n° 100 échouera.

Exemple : définir une vue sur PILOTE, permettant la vérification de la contrainte de domaine suivante : le salaire d'un pilote est compris entre 3.000 et 5.000.

```
CREATE VIEW  DPILOTE
AS
SELECT       *
FROM         PILOTE
WHERE        SALAIRE  BETWEEN  3000  AND  5000
WITH CHECK OPTION
```

L'insertion suivante va alors échouer.

```
INSERT INTO DPILOTE (NUMPIL,SALAIRE) VALUES(175,7000)
```

Exemple : définir une vue sur vol permettant de vérifier les contraintes d'intégrité référentielle en insertion et en modification.

```
CREATE VIEW  IMVOL
AS
SELECT       *
FROM         VOL
WHERE        NUMPIL IN  (SELECT NUMPIL FROM PILOTE)
AND          NUMAV  IN  (SELECT NUMAV  FROM AVION)
WITH CHECK OPTION
```

De manière similaire à une relation de la base, une vue peut être

- consultée via une requête d'interrogation ;
- décrite structurellement grâce à la commande **DESCRIBE** ou en interrogeant la table système ALL_VIEWS ;
- détruite par l'ordre **DROP VIEW** <nom_vue>.

3.3 Gestion des utilisateurs et de leurs privilèges

3.3.1 Création et suppression d'utilisateurs

Pour pouvoir accéder à ORACLE, un utilisateur doit être identifié en tant que tel par le système. Pour ce faire, il doit avoir un nom, un mot de passe et un ensemble de privilèges (ou droits).
Seul un administrateur (DBA - Data Base Administrator) peut créer des utilisateurs. La commande de création est la suivante :

```
GRANT [CONNECT, ][RESOURCE,] [DBA]
TO  utilisateur, …
[IDENTIFIED BY mot_de_passe,…]
```

Les options **CONNECT**, **RESOURCE** et **DBA** déterminent la classe de l'utilisateur à créer, i.e. les droits qui lui sont attribués. Les droits associés à ces options sont :

- La connexion (**CONNECT**) :
  - Connexion à tous les outils ORACLE ;
  - Manipulation des données sur lesquelles une autorisation a été attribuée au préalable ;
  - Création des vues et synonymes.

- La création de ressources (**RESOURCE**)
  Cette option n'est applicable que pour les utilisateurs ayant le privilège **CONNECT**. Elle offre en plus les droits suivants :
  - Création des tables et index ;
  - Attribution et résiliation de privilèges sur des tables et index à d'autres utilisateurs ;

- L'administration (**DBA**)
  Cette option englobe les droits des deux options précédentes. Elle offre en plus les droits suivants :
  - Accès à toutes les données de tous les utilisateurs ;
  - Création et suppression d'utilisateurs et de droits.

La clause **IDENTIFIED BY** n'est obligatoire que lors de la création d'un nouvel utilisateur ou pour modifier le mot de passe d'un utilisateur existant. Il est aussi possible de créer ou d'attribuer les mêmes privilèges à un ensemble d'utilisateurs dans une même commande.

Exemple : créer un nouvel utilisateur ut1 avec le mot de passe ut1p avec juste le droit de connexion.

```
GRANT CONNECT  TO ut1   IDENTIFIED BY ut1p
```

Cette commande ne peut être exécutée que par les utilisateurs ayant le privilège **DBA**

Exemple : changer le mot de passe de ut1. Le nouveau mot de passe est utp.

```
GRANT CONNECT  TO ut1  IDENTIFIED BY utp
```

Un utilisateur d'ORACLE peut être supprimé à tout moment ou se voir démuni de certains privilèges. La commande correspondante est :

```
REVOKE [CONNECT,] [RESOURCE,] [DBA]
FROM     utilisateur,…
```

L'option **CONNECT** interdit donc à l'utilisateur de se connecter de nouveau à ORACLE.

3.3.2 Création et suppression de droits

Toute table, vue ou synonyme n'est initialement accessible que par l'utilisateur qui l'a créé. Le partage de ces objets est souvent nécessaire. Pour rendre ce partage possible, le SGBD permet au propriétaire d'un objet d'accorder des droits sur cet objet à d'autres utilisateurs. La commande d'attribution des droits a la syntaxe suivante :

```
GRANT    droit,…
ON       objet
TO       utilisateur,…
[WITH GRANT OPTION]
```

Les droits possibles sont :

SELECT	interrogation des données ;
INSERT	ajout des données ;
UPDATE	modification des données ;
DELETE	suppression des données ;
ALTER	modification de la structure d'une relation ;
INDEX	création d'index ;
ALL	toutes les opérations.

Le privilège sur l'opération **UPDATE** peut concerner seulement quelques colonnes d'une table.

Tout utilisateur qui reçoit un privilège sur une table avec l'option **WITH GRANT OPTION** a en plus le droit de l'accorder à un autre utilisateur.
ALL peut être utiliser pour désigner tous les droits et **PUBLIC** pour désigner tous les utilisateurs.
Un droit ne peut être retiré que par l'utilisateur qui l'a accordé ou bien qui a le privilège d'administration. La syntaxe de cette commande est la suivante :

```
REVOKE   droit,…
```

```
ON      objet
FROM    utilisateur,…
```

Exemple : attribuer puis retirer à l'utilisateur ut1 le droit d'interrogation et modification de la relation VOL.

```
GRANT   SELECT, UPDATE
ON      VOL
TO      ut1

REVOKE  SELECT, UPDATE
ON      VOL
FROM    ut1
```

Exemple : interdire toute opération à tous les utilisateurs sur la table avion.

```
REVOKE  ALL
ON      AVION
FROM    PUBLIC
```