1. Structure générale du langage SQL

Les instructions essentielles SQL se répartissent en trois familles fonctionnellement distinctes et trois formes d'utilisation :
Selon la norme SQL 92, le **SQL interactif** permet d'exécuter une requête et d'en obtenir immédiatement une réponse. Le **SQL intégré** (ou module SQL) permet d'utiliser SQL dans un langage de troisième génération (C, Cobol, ...), les instructions permettant essentiellement de gérer les curseurs. Le **SQL dynamique** est une extension du SQL intégré qui permet d'exécuter des requêtes SQL non connues au moment de la compilation. Hors norme, SQL est utilisable sous la forme de librairies de fonctions API (exemple : ODBC). Nous nous limitons au SQL interactif.
Dans le SQL interactif, le LDD (Langage de Définition de données) permet la description de la structure de la base (tables, vues, index, attributs, ...). Le dictionnaire contient à tout moment le descriptif complet de la structure de données. Le LMD (Langage de Manipulation de Données) permet la manipulation des tables et des vues. Le LCD (Langage de Contrôle des Données) contient les primitives de gestion des transactions et des privilèges d'accès aux données. Le tableau ci-dessous vous donne les principales primitives SQL et leur classification. Nous nous intéresserons essentiellement au LMD et à la commande SELECT.

LDD	**LMD**	**LCD**
CREATE	SELECT	GRANT
DROP	INSERT	REVOKE
ALTER	DELETE	CONNECT
	UPDATE	COMMIT
		ROLLBACK
		SET

2. Objets manipulés par SQL

2.1. Identificateurs

SQL utilise des identificateurs pour désigner les objets qu'il manipule :
utilisateurs, tables, colonnes, index, fonctions, etc.
Un identificateur est un mot formé d'au plus 30 caractères, commençant obligatoirement par une lettre de l'alphabet. Les caractères suivants peuvent être une lettre, un chiffre, ou l'un des symboles $ et _. SQL ne fait pas la différence entre les lettres minuscules et majuscules. Les voyelles accentuées ne sont pas acceptées. Un identificateur ne doit pas figurer dans la liste des mots clés réservés. Voici quelques mots clés que l'on risque d'utiliser comme identificateurs : ASSERT, ASSIGN, AUDIT, COMMENT, DATE, DECIMAL, DEFINITION, FILE, FORMAT, INDEX, LIST, MODE, OPTION, PARTITION, PRIVILEGES, PUBLIC, SELECT, SESSION, SET, TABLE.

2.2. Tables

Les relations (d'un schéma relationnel) sont stockées sous forme de tables composées de lignes et de colonnes.
(a) Selon la norme SQL-2, le nom d'une table devrait être précédé d'un nom de schéma.
(b) Il est d'usage (mais non obligatoire évidemment) de mettre les noms de table au singulier : plutôt EMPLOYE que EMPLOYES pour une table d'employés.

2.3. Les Types de données

2.3.1. Les types numériques

Ils sont divisés en deux grandes classes : les types entiers et réels.
Quelques types entiers : **TINYINT, SMALLINT, INT, BIGINT**
Quelques types réels : **REAL, FLOAT, DOUBLE, NUMERIC, DECIMAL(m,n)**
(m,n) veut dire m chiffres au total dont n dans la partie décimale et m-n dans la partie entière.
Exemple
SALAIRE DECIMAL(8,2)
définit une colonne numérique SALAIRE. Les valeurs auront au maximum 2 décimales et 8 chiffres au plus au total (donc 6 chiffres avant le point décimal)

2.3.2. Le type chaines de caractères

Les constantes chaines de caractères sont entourées par de simples côtes. Dans le cas où une chaine contient une apostrophe, l'apostrophe est doublée. Example 'aujourd''hui'.
Il existe deux types pour les colonnes qui contiennent des chaînes de caractères :

- le type CHAR pour les colonnes qui contiennent des chaînes de longueur constante.

La déclaration de type chaîne de caractères de longueur constante a le format suivant :
CHAR(longueur)
où longueur est la longueur maximale (en nombre de caractères) qu'il sera possible de stocker dans le champ ; par défaut, longueur est égale à 1. L'insertion d'une chaîne dont la longueur est supérieure à longueur sera refusée. Une chaîne plus courte que longueur sera complétée par des espaces (important pour les comparaisons de chaînes)

- le type VARCHAR pour les colonnes qui contiennent des chaînes de longueurs variables.

On déclare ces colonnes par :
VARCHAR(longueur)
longueur indique la longueur maximale des chaînes contenues dans la colonne.

2.3.3. Types temporels

Les types temporels de SQL-2 sont :
DATE réserve 2 chiffres pour le mois et le jour et 4 pour l'année ;

TIME pour les heures, minutes et secondes (les secondes peuvent comporter un certain nombre de décimales) ;
TIMESTAMP permet d'indiquer un moment précis par une date avec heures, minutes et secondes (6 chiffres après la virgule ; c'est-à-dire en microsecondes) ;

2.3.4. Enumérations

Un attribut de type ENUM peut prendre une valeur parmi celles définies lors de la création de la table plus la chaîne vide ainsi que NULL si la définition le permet. Ces valeurs sont exclusivement des chaînes de caractères. Une énumération peut contenir 65535 éléments au maximum.
Définition d'un tel attribut :
nom_attribut ENUM("valeur 1","valeur 2"…)
nom_attribut ENUM("valeur 1","valeur 2"…) NULL
A chaque valeur est associée un index allant de 0 à N si N valeurs ont été définies. L'index 0 est associé à la chaîne nulle, l'index 1 à la première valeur… L'index NULL est associé à la valeur NULL

2.3.5. Valeur NULL

Une colonne qui n'est pas renseignée, et donc vide, est dite contenir la valeur NULL. Cette valeur n'est pas zéro, c'est une absence de valeur.

3. Création de tables et contraintes d'intégrité

3.1. Création de tables

L'ordre **CREATE TABLE** permet de créer une table en définissant le nom et le type de chacune des colonnes de la table

```
CREATE TABLE NomDeLaTable(
colonne1 type1,
colonne2, type2,
...
)
```

NomDeLaTable est le nom que l'on donne à la table ; colonne1, colonne2,.. sont les noms des colonnes ; type1, type2,.. sont les types des données qui seront contenues dans les colonnes.
On peut ajouter après la description d'une colonne l'option NOT NULL qui interdira que cette colonne contienne la valeur NULL. On peut aussi ajouter des contraintes d'intégrité portant sur une ou plusieurs colonnes de la table
Exemple : Soit à donner le script SQL de création de la table produit de schéma suivant :
Produit(NomProduit, Prix, DatePeremtion);
Le script est le suivant :

CREATE TABLE Produit(
NomProduit varchar(20) not null,
Prix decimal(9,2),
DatePeremtion date);

3.2. Contrainte d'intégrité

Dans la définition d'une table, on peut indiquer des contraintes d'intégrité portant sur une ou plusieurs colonnes. Les contraintes possibles sont :
PRIMARY KEY, UNIQUE, FOREIGN KEY...REFERENCES, CHECK
Toute définition de table doit comporter au moins une contrainte de type **PRIMARY KEY.**
Chaque contrainte doit être nommée (ce qui permettra de la désigner par un ordre **ALTER TABLE**, et ce qui est requis par les nouvelles normes SQL)
CONSTRAINT nom-contrainte définition-contrainte
Le nom d'une contrainte doit être unique parmi toutes les contraintes de toutes les tables de la base de données.

3.3. Types de contraintes

3.3.1. Contrainte de clé primaire

(pour une contrainte sur une table :)
PRIMARY KEY (colonne1, colonne2,...) (pour une contrainte sur une colonne :)
PRIMARY KEY définit la clé primaire de la table. Aucune des colonnes qui forment cette clé ne doit avoir une valeur NULL.

3.3.2. Contrainte d'unicité

(pour une contrainte sur une table :)
UNIQUE (colonne1, colonne2,...)
(pour une contrainte sur une colonne :)
UNIQUE interdit qu'une colonne (ou la concaténation de plusieurs colonnes) contienne deux valeurs identiques.

3.3.3. Contrainte de clé étrangère

(pour une contrainte sur une table :)
FOREIGN KEY (colonne1, colonne2,...)
REFERENCES tableref [(col1, col2,...)]
[ON DELETE CASCADE]
(pour une contrainte sur une colonne :)
REFERENCES tableref [(col1)]
[ON DELETE CASCADE]
indique que la concaténation de colonne1, colonne2,... (ou la colonne que l'on dénit

pour une contrainte sur une colonne) est une clé étrangère qui fait référence à la concaténation des colonnes col1, col2,... de la table tableref (contrainte d'intégrité référentielle). Si aucune colonne de tableref n'est indiquée, c'est la clé primaire de tableref qui est prise par défaut.
Cette contrainte ne permettra pas d'insérer une ligne de la table si la table tableref ne contient aucune ligne dont la concaténation des valeurs de col1, col2,... est égale à la concaténation des valeurs de colonne1, colonne2,...
col1, col2,... doivent avoir la contrainte PRIMARY KEY ou UNIQUE.
Ceci implique qu'une valeur de colonne1, colonne2,... va référencer une et une seule ligne de tableref.
L'option **ON DELETE CASCADE** indique que la suppression d'une ligne de tableref va entraîner automatiquement la suppression des lignes qui la référencent dans la table. Si cette option n'est pas indiquée, il est impossible de supprimer des lignes de tableref qui sont référencées par des lignes de la table.
A la place de **ON DELETE CASCADE** on peut donner l'option **ON DELETE SET NULL**. Dans ce cas, la clé étrangère sera mise à NULL si la ligne qu'elle référence dans tableref est supprimée.
La norme SQL2 ore 2 autres options qui ne sont pas implémentée dans Oracle :
ON DELETE SET DEFAULT met une valeur par défaut dans la clé étrangère quand la clé primaire référencée est supprimée.
ON UPDATE CASCADE modifie la clé étrangère si on modifie la clé primaire (ce qui est à éviter).
ON UPDATE SET NULL met NULL dans la clé étrangère quand la clé primaire référencée est modifiée.
ON UPDATE SET DEFAULT met une valeur par défaut dans la clé étrangère quand la clé primaire référencée est modifiée.

3.3.4. Contrainte CHECK

CHECK(condition)
donne une condition que les colonnes de chaque ligne devront vérifier (exemples dans la section suivante). On peut ainsi indiquer des contraintes d'intégrité de domaines. Cette contrainte peut être une contrainte de colonne ou de table. Si c'est une contrainte de colonne, elle ne doit porter que sur la colonne en question.

3.4. Ajouter, supprimer ou renommer une contrainte

Des contraintes d'intégrité peuvent être ajoutées ou supprimées par la commande ALTER TABLE. On peut aussi modifier l'état de contraintes par MODIFY CONSTRAINT. On ne peut ajouter que des contraintes de table.
Exemple

```
ALTER TABLE EMP
DROP CONSTRAINT NOM_UNIQUE
ADD CONSTRAINT SAL_MIN CHECK(SAL + NVL(COMM,0) > 1000)
```

RENAME CONSTRAINT NOM1 TO NOM2
MODIFY CONSTRAINT SAL_MIN DISABLE

4. Langage de manipulation des données

Le langage de manipulation de données (LMD) est le langage permettant de modifier les informations contenues dans la base. Il existe trois commandes SQL permettant d'effectuer les trois types de modification des données :
INSERT ajout de lignes
UPDATE mise à jour de lignes
DELETE suppression de lignes
Ces trois commandes travaillent sur la base telle qu'elle était au début de l'exécution de la commande. Les modifications effectuées par les autres utilisateurs entre le début et la fin de l'exécution ne sont pas prises en compte (même pour les transactions validées).

4.1. Insertion

INSERT INTO table (col1,..., coln)
VALUES (val1,...,valn)
ou
INSERT INTO table (col1,..., coln)
SELECT ...
table est le nom de la table sur laquelle porte l'insertion. col1,..., coln est la liste des noms des colonnes pour lesquelles on donne une valeur. Cette liste est optionnelle. Si elle est omise, le SGBD prendra par défaut l'ensemble des colonnes de la table dans l'ordre où elles ont été données lors de la création de la table. Si une liste de colonnes est spécifiée, les colonnes ne figurant pas dans la liste auront la valeur NULL.
Exemples
(a) **INSERT INTO dept**
VALUES (10, 'FINANCES', 'DOUALA')
(b) **INSERT INTO dept (lieu, nomd, dept)**
VALUES ('NKONGSAMBA', 'RECHERCHE', 20)
La deuxième forme avec la clause SELECT permet d'insérer dans une table des lignes provenant d'une table de la base. Le SELECT a la même syntaxe qu'un SELECT normal.
Exemple
Enregistrer la participation de MARTIN au groupe de projet numéro 10 :
INSERT INTO PARTICIPATION (MATR, CODEP)
SELECT MATR, 10 FROM EMP
WHERE NOME = 'MARTIN'

4.2. Modification

La commande **UPDATE** permet de modifier les valeurs d'un ou plusieurs champs, dans une ou plusieurs lignes existantes d'une table.

UPDATE table
SET col1 = exp1, col2 = exp2, ...
WHERE prédicat

ou

UPDATE table
SET (col1, col2,...) = (SELECT ...)
WHERE prédicat

table est le nom de la table mise à jour ; col1, col2, ... sont les noms des colonnes qui seront modifiées ; exp1, exp2,... sont des expressions. Elles peuvent aussi être un ordre SELECT renvoyant les valeurs attribuées aux colonnes (deuxième variante de la syntaxe).

Les valeurs de col1, col2... sont mises à jour dans toutes les lignes satisfaisant le prédicat. La clause WHERE est facultative. Si elle est absente, toutes les lignes sont mises à jour.

Exemples

(a) **Faire passer MARTIN dans le département 10 :**

UPDATE EMP
SET DEPT = 10
WHERE NOME = 'MARTIN'

(b) Augmenter de 10 % les commerciaux :

UPDATE EMP
SET SAL = SAL * 1.1
WHERE POSTE = 'COMMERCIAL'

(c) Donner à CLEMENT un salaire 10 % au dessus de la moyenne des salaires des secrétaires :

UPDATE EMP
SET SAL = (SELECT AVG(SAL) * 1.10
FROM EMP
WHERE POSTE = 'SECRETAIRE')
WHERE NOME = 'CLEMENT'

4.3. Suppression

L'ordre **DELETE** permet de supprimer des lignes d'une table.

DELETE FROM table
WHERE prédicat

La clause WHERE indique quelles lignes doivent être supprimées. ATTENTION : cette clause est facultative ; si elle n'est pas précisée, TOUTES LES LIGNES DE LA TABLE SONT SUPPRIMEES (heureusement qu'il existe ROLLBACK!).

Le prédicat peut contenir des sous-interrogations.
Exemple 3.4 **DELETE FROM dept**
WHERE dept = 10

5. Interrogations

5.1. Syntaxe générale

L'ordre SELECT possède six clauses différentes, dont seules les deux premières sont obligatoires. Elles sont données ci-dessous, dans l'ordre dans lequel elles doivent apparaître, quand elles sont utilisées :
SELECT ...
FROM ...
WHERE ...
GROUP BY ...
HAVING ...
ORDER BY ...

5.2. Clause SELECT

Cette clause permet d'indiquer quelles colonnes, ou quelles expressions doivent être retournées par l'interrogation.
SELECT [DISTINCT] *
ou
SELECT [DISTINCT] exp1 [[AS] nom1], exp2 [[AS] nom2], ...
exp1, exp2, ... sont des expressions, nom1, nom2, ... sont des noms facultatifs de 30 caractères maximum, donnés aux expressions. Chacun de ces noms est inséré derrière l'expression, séparé de cette dernière par un blanc ou par le mot clé AS (optionnel) ; il constituera le titre de la colonne dans l'affichage du résultat de la sélection. Ces noms ne peuvent être utilisés dans les autres clauses (where par exemple).
'*' signifie que toutes les colonnes de la table sont sélectionnées.
Le mot clé facultatif DISTINCT ajouté derrière l'ordre SELECT permet d'éliminer les duplications : si, dans le résultat, plusieurs lignes sont identiques, une seule sera conservée.
Exemples
(a) **SELECT * FROM DEPT**
(b) **SELECT DISTINCT POSTE FROM EMP**
(c) **SELECT NOME, SAL + NVL(COMM,0) AS Salaire FROM EMP**
(d) La requête suivante va provoquer une erreur car on utilise le nom Salaire dans la clause where :
SELECT NOME, SAL + NVL(COMM,0) AS Salaire FROM EMP
WHERE Salaire > 1000
Si le nom contient des séparateurs (espace, caractère spécial), ou s'il est identique à un

mot réservé SQL (exemple : DATE), il doit être mis entre guillemets.
Exemple
SELECT NOME, SAL + NVL(COMM,0) "Salaire Total" FROM EMP
Le nom complet d'une colonne d'une table est le nom de la table suivi d'un point et du nom de la colonne. Par exemple : EMP.MATR, EMP.DEPT, DEPT.DEPT
Le nom de la table peut être omis quand il n'y a pas d'ambiguïté. Il doit être précisé s'il y a une ambiguïté, ce qui peut arriver quand on fait une sélection sur plusieurs tables à la fois et que celles-ci contiennent des colonnes qui ont le même nom

5.2.1. select comme expression

La norme SQL-2, mais pas tous les SGBD, permet d'avoir des select emboîtés parmi les expressions. Il faut éviter cette facilité dans les programmes si on veut qu'ils soient portables. Rien n'empêche de l'utiliser en interactif si elle existe.
C'est possible avec Oracle :

```
select nomE, sal/(select sum(sal) from emp)*100 from emp
```

Un select "expression" doit ramener une valeur au plus. S'il ne ramène aucune valeur, l'expression est égale à la valeur null. Si cette possibilité existe, le select emboîté peut même alors être synchronisé avec le select principal (le vérifier sur le SGBD que vous utilisez). La requête suivante affiche les noms des employés avec le nombre d'employés qui gagnent plus :

```
select nomE, (select count(*) from emp where sal > e1.sal) as rang
from emp e1
```

pour obtenir les 5 plus gros salaires) :

```
select nome, rang + 1
from (select nome, (select count(*) from emp where sal > e1.sal) as rang
      from emp e1)
where rang < 5
order by rang;
```

5.3. Clause FROM

La clause **FROM** donne la liste des tables participant à l'interrogation. Il est possible de lancer des interrogations utilisant plusieurs tables à la fois. **FROM table1 [synonyme1] , table2 [synonyme2] , ...**
synonyme1, synonyme2,... sont des synonymes attribués facultativement aux tables pour le temps de la sélection. On utilise cette possibilité pour lever certaines ambiguïtés, quand la même table est utilisée de plusieurs façons différentes dans une même interrogation. Quand on a donné un synonyme à une table dans une requête, elle n'est plus reconnue sous son nom d'origine dans cette requête. Le nom complet d'une table est celui de son créateur, suivi d'un point et du nom de la table. Par défaut, le nom du créateur est celui de l'utilisateur en cours. Ainsi, on peut se dispenser de préciser ce nom quand on travaille sur ses propres tables. Mais il faut le préciser dès que l'on se sert de la table d'un autre utilisateur.

Quand on précise plusieurs tables dans la clause FROM, on obtient le produit cartésien des tables. On verra plus loin (remarque 4.5 page 32) une syntaxe spéciale pour les produits cartésiens.
Exemple
Produit cartésien des noms des départements par les numéros des départements :
SQL> **SELECT B.dept, A.nomd**
2 FROM dept A,dept B;
DEPT NOMD
---- ------------------
10 FINANCES
20 FINANCES
30 FINANCES
10 RECHERCHE
20 RECHERCHE
30 RECHERCHE
10 VENTES
20 VENTES
30 VENTES
La norme SQL2 permet d'avoir un SELECT à la place d'un nom de table.
Exemples
(a) Pour obtenir la liste des employés avec le pourcentage de leur salaire par rapport au total des salaires, il fallait auparavant utiliser une vue. Il est maintenant possible d'avoir cette liste avec une seule instruction SELECT :

```
select nome, sal, sal/total*100
from emp, (select sum(sal) as total from emp)
```

(b) On peut donner un synonyme comme nom de table au select pour l'utiliser par ailleurs dans le select principal :

```
select nome, sal, sal/total*100
from emp, (select dept, sum(sal) as total from emp group by dept) TOTALDEPT
where emp.dept = TOTALDEPT.dept
```

5.4. Clause WHERE

La clause **WHERE** permet de spécifier quelles sont les lignes à sélectionner dans une table ou dans le produit cartésien de plusieurs tables. Elle est suivie d'un prédicat (expression logique ayant la valeur vrai ou faux) qui sera évalué pour chaque ligne. Les lignes pour lesquelles le prédicat est vrai seront sélectionnées. La clause where est étudiée ici pour la commande SELECT. Elle peut se rencontrer aussi dans les commandes UPDATE et DELETE avec la même syntaxe.

5.4.1. Clause WHERE simple

WHERE prédicat

Un prédicat simple est la comparaison de deux expressions ou plus au moyen d'un opérateur logique :

WHERE exp1 = exp2
WHERE exp1 != exp2
WHERE exp1 < exp2
WHERE exp1 > exp2
WHERE exp1 <= exp2
WHERE exp1 >= exp2
WHERE exp1 BETWEEN exp2 AND exp3
WHERE exp1 LIKE exp2
WHERE exp1 NOT LIKE exp2
WHERE exp1 IN (exp2, exp3,...)
WHERE exp1 NOT IN (exp2, exp3,...)
WHERE exp IS NULL
WHERE exp IS NOT NULL

Les trois types d'expressions (arithmétiques, caractères, ou dates) peuvent être comparées au moyen des opérateurs d'égalité ou d'ordre (=, !=, <, >, <=, >=) : pour les types date, la relation d'ordre est l'ordre chronologique ; pour les types caractères, la relation d'ordre est l'ordre lexicographique.

Il faut ajouter à ces opérateurs classiques les opérateurs suivants **BETWEEN, IN, LIKE, IS NULL** :

exp1 BETWEEN exp2 AND exp3

est vrai si exp1 est compris entre exp2 et exp3, bornes incluses.

exp1 IN (exp2 , exp3...)

est vrai si exp1 est égale à l'une des expressions de la liste entre parenthèses.

exp1 LIKE exp2

teste l'égalité de deux chaînes en tenant compte des caractères jokers dans
la 2ème chaîne :

- "_" remplace 1 caractère exactement
- "%" remplace une chaîne de caractères de longueur quelconque, y compris de longueur nulle

Le fonctionnement est le même que celui des caractères joker ? et * pour le shell sous Unix. Ainsi l'expression 'MARTIN' LIKE '_AR%' sera vraie.

Remarque

L'utilisation des jokers ne fonctionne qu'avec LIKE ; elle ne fonctionne pas avec "=".

L'opérateur IS NULL permet de tester la valeur NULL :

exp IS [NOT] NULL

est vrai si l'expression a la valeur NULL (ou l'inverse avec NOT).

5.4.2. Opérateurs logiques

Les opérateurs logiques AND et OR peuvent être utilisés pour combiner plusieurs prédicats (l'opérateur AND est prioritaire par rapport à l'opérateur OR). Des

parenthèses peuvent être utilisées pour imposer une priorité dans l'évaluation du prédicat, ou simplement pour rendre plus claire l'expression logique.
L'opérateur NOT placé devant un prédicat en inverse le sens.
Exemples
(a) Sélectionner les employés du département 30 ayant un salaire supérieur à 1500 frs.

```
SELECT NOME FROM EMP
WHERE DEPT = 30 AND SAL > 1500
```

(b) Afficher une liste comprenant les employés du département 30 dont le salaire est supérieur à 11000 Frs et (attention, à la traduction par OR) les employés qui ne touchent pas de commission.

```
SELECT nome FROM emp
WHERE dept = 30 AND sal > 11000 OR comm IS NULL
```

(c)

```
SELECT * FROM EMP
WHERE (POSTE = 'DIRECTEUR' OR POSTE = 'SECRETAIRE')
AND DEPT = 10
```

La clause WHERE peut aussi être utilisée pour faire des jointures (vues dans le cours sur le modèle relationnel) et des sous-interrogations (une des valeurs utilisées dans un WHERE provient d'une requête SELECT emboîtée) comme nous allons le voir dans les sections suivantes.

5.5. Jointure

Quand on précise plusieurs tables dans la clause FROM, on obtient le produit cartésien des tables. Ce produit cartésien offre en général peu d'intérêt. Ce qui est normalement souhaité, c'est de joindre les informations de diverses tables, en "recollant" les lignes des tables suivant les valeurs qu'elles ont dans certaines colonnes. Une variante de la clause FROM permet de préciser les colonnes qui servent au recollement. On retrouve l'opération de jointure du modèle relationnel.
Exemple
Liste des employés avec le nom du département où ils travaillent :

```
SELECT NOME, NOMD
FROM EMP JOIN DEPT ON EMP.DEPT = DEPT.DEPT
```

La clause FROM indique de ne conserver dans le produit cartésien des tables EMP et DEPT que les éléments pour lesquels le numéro de département provenant de la table DEPT est le même que le numéro de département provenant de la table EMP. Ainsi, on obtiendra bien une jointure entre les tables EMP et DEPT d'après le numéro de département. Par opposition aux jointures externes que l'on va bientôt étudier, on peut ajouter le mot clé INNER :

```
SELECT NOME, NOMD
FROM EMP INNER JOIN DEPT ON EMP.DEPT = DEPT.DEPT
```

Remarque
Cette syntaxe SQL2 n'est pas supportée par tous les SGBD (par exemple, les versions d'Oracle antérieures à la version 9 ne la supportaient pas). La jointure peut aussi être

traduite par la clause WHERE :

SELECT NOME, NOMD
FROM EMP, DEPT
WHERE EMP.DEPT = DEPT.DEPT

Cette façon de faire est encore très souvent utilisée, même avec les SGBD qui supportent la syntaxe SQL2.

Remarque

La norme SQL2 a aussi une syntaxe spéciale pour le produit cartésien de deux tables :

SELECT NOME, NOMD
FROM EMP CROSS JOIN DEPT

5.5.1. Jointure naturelle

Lorsque l'on a une équi-jointure (c'est-à-dire quand la condition de jointure est une égalité de valeurs entre une colonne de la première table et une colonne de la deuxième), il est le plus souvent inutile d'avoir les deux colonnes de jointure dans le résultat, puisque les valeurs sont égales. On peut préférer la jointure naturelle qui ne garde dans la jointure qu'une colonne pour les deux colonnes qui ont servi à la jointure. Évidemment, à moins d'utiliser "*" dans la clause du select, on ne voit pas la différence à l'affichage. Voici l'exemple précédent avec une jointure naturelle :

SELECT NOME, NOMD
FROM EMP NATURAL JOIN DEPT

On remarque qu'on n'a pas besoin dans cet exemple d'indiquer les colonnes de jointure car la clause "natural join" joint les deux tables sur toutes les colonnes qui ont le même nom dans les deux tables.

Remarque

Si on utilise une jointure naturelle, il est interdit de préfixer une colonne utilisée pour la jointure par un nom de table. La requête suivante provoque une erreur :

SELECT NOME, NOMD, DEPT.DEPT
FROM EMP NATURAL JOIN DEPT

Il faut écrire :

SELECT NOME, NOMD, DEPT
FROM EMP NATURAL JOIN DEPT

Au cas où on voudrait faire une jointure naturelle sur une partie seulement des colonnes qui ont le même nom, il faut utiliser la clause "join using' (s'il y a plusieurs colonnes, le séparateur de colonnes est la virgule). La requête suivante est équivalente à la précédente :

SELECT NOME, NOMD
FROM EMP JOIN DEPT USING (DEPT)

5.5.2. Jointure d'une table avec elle-même

Il peut être utile de rassembler des informations venant d'une ligne d'une table avec des informations venant d'une autre ligne de la même table. Dans ce cas il faut

renommer au moins l'une des deux tables en lui donnant un synonyme , afin de pouvoir préfixer sans ambiguïté chaque nom de colonne.
Exemple
Lister les employés qui ont un supérieur, en indiquant pour chacun le nom de son supérieur :

```
SELECT EMP.NOME EMPLOYE, SUPE.NOME SUPERIEUR
FROM EMP join EMP SUPE on EMP.SUP = SUPE.MATR
```

ou

```
SELECT EMP.NOME EMPLOYE, SUPE.NOME SUPERIEUR
FROM EMP, EMP SUPE
WHERE EMP.SUP = SUPE.MATR
```

5.5.3. Jointure externe

Le **SELECT** suivant donnera la liste des employés et de leur département :

```
SELECT DEPT.DEPT, NOMD, NOME
FROM DEPT JOIN EMP ON DEPT.DEPT = EMP.DEPT
```

Dans cette sélection, un département qui n'a pas d'employé n'apparaîtra jamais dans la liste, puisqu'il n'y aura dans le produit cartésien des deux tables aucun élément où l'on trouve une égalité des colonnes DEPT.
On pourrait pourtant désirer une liste des divers départements, avec leurs employés s'ils en ont, sans omettre les départements sans employés. On écrira alors :

```
SELECT DEPT.DEPT, NOMD, NOME
FROM emp RIGHT OUTER JOIN dept ON emp.dept = dept.dept
```

La jointure externe ajoute des lignes fictives dans une des tables pour faire la correspondance avec les lignes de l'autre table. Dans l'exemple précédent, une ligne fictive (un employé fictif) est ajoutée dans la table des employés si un département n'a pas d'employé. Cette ligne aura tous ses attributs null, sauf celui des colonnes de jointure.
RIGHT indique que la table dans laquelle on veut afficher toutes les lignes (la table dept) est à droite de **RIGHT OUTER JOIN**. C'est dans l'autre table (celle de gauche) dans laquelle on ajoute des lignes fictives. De même, il existe **LEFT OUTER JOIN** qui est utilisé si on veut afficher toutes les lignes de la table de gauche (avant le "**LEFT OUTER JOIN**") et **FULL OUTER JOIN** si on veut afficher toutes les lignes des deux tables.

5.5.4. Jointure "non équi"

Les jointures autres que les équi-jointures peuvent être représentées en remplaçant dans la clause ON ou la clause WHERE le signe "=" par un des Exemples
(a) Liste des employés, avec tous les employés qui gagnent plus :

```
select emp.nome, emp.sal, empplus.nome, empplus.sal
from emp join emp empplus on emp.sal < empplus.sal
order by emp.sal
```

(b) Si la table tranche contient les informations sur les tranches d'impôts, on peut obtenir le taux de la tranche maximum liée à un salaire par la requête suivante :

```
select nomE, sal, pourcentage
from emp join tranche on sal between min and max
```

(c) Dans les anciennes versions d'Oracle, on écrit :

```
select nomE, sal, pourcentage
from emp, tranche
where sal between min and max
```

5.6. Sous-interrogation

Une caractéristique puissante de SQL est la possibilité qu'un prédicat employé dans une clause WHERE (expression à droite d'un opérateur de comparaison) comporte un SELECT emboîté.

Par exemple, la sélection des employés ayant même poste que MARTIN peut s'écrire en joignant la table EMP avec elle-même :

```
SELECT EMP.NOME
FROM EMP JOIN EMP MARTIN ON EMP.POSTE = MARTIN.POSTE
WHERE MARTIN.NOME = 'MARTIN'
```

mais on peut aussi la formuler au moyen d'une sous-interrogation :

```
SELECT NOME FROM EMP
WHERE POSTE = (SELECT POSTE
FROM EMP
WHERE NOME = 'MARTIN')
```

Les sections suivantes exposent les divers aspects de ces sous-interrogations.

5.6.1. Sous-interrogation à une ligne et une colonne

Dans ce cas, le **SELECT** imbriqué équivaut à une valeur.

```
WHERE exp op (SELECT ...)
```

où op est un des opérateurs = != < > <= >= exp est toute expression légale.

Exemple 4.9

Liste des employés travaillant dans le même département que MERCIER :

```
SELECT NOME FROM EMP
WHERE DEPT = (SELECT DEPT FROM EMP
WHERE NOME = 'MERCIER')
```

Un SELECT peut comporter plusieurs sous-interrogations, soit imbriquées, soit au même niveau dans différents prédicats combinés par des AND ou des OR.

Exemples

(a) Liste des employés du département 10 ayant même poste que quelqu'un du département VENTES :

```
SELECT NOME, POSTE FROM EMP
WHERE DEPT = 10
```

AND POSTE IN
(SELECT POSTE
FROM EMP
WHERE DEPT = (SELECT DEPT
FROM DEPT
WHERE NOMD = 'VENTES'))

(b) Liste des employés ayant même poste que MERCIER ou un salaire supérieur à CHATEL :

SELECT NOME, POSTE, SAL FROM EMP
WHERE POSTE = (SELECT POSTE FROM EMP
WHERE NOME = 'MERCIER')
OR SAL > (SELECT SAL FROM EMP WHERE NOME = 'CHATEL')

Jointures et sous-interrogations peuvent se combiner.

Exemple

Liste des employés travaillant à LYON et ayant même poste que FREMONT.

SELECT NOME, POSTE
FROM EMP JOIN DEPT ON EMP.DEPT = DEPT.DEPT
WHERE LIEU = 'LYON'
AND POSTE = (SELECT POSTE FROM EMP
WHERE NOME = 'FREMONT')

On peut aussi plus simplement utiliser la jointure naturelle puisque les noms des colonnes de jointures sont les mêmes :

SELECT NOME, POSTE
FROM EMP NATURAL JOIN DEPT
WHERE LIEU = 'LYON'
AND POSTE = (SELECT POSTE FROM EMP
WHERE NOME = 'FREMONT')

Attention : une sous-interrogation à une seule ligne doit ramener une seule ligne ; dans le cas où plusieurs lignes, ou pas de ligne du tout seraient ramenées, un message d'erreur sera affiché et l'interrogation sera abandonnée.

5.6.2. Sous-interrogation ramenant plusieurs lignes

Une sous-interrogation peut ramener plusieurs lignes à condition que l'opérateur de comparaison admette à sa droite un ensemble de valeurs. Les opérateurs permettant de comparer une valeur à un ensemble de valeurs sont :

- l'opérateur **IN**
- les opérateurs obtenus en ajoutant **ANY** ou **ALL** à la suite des opérateurs de comparaison classique =, !=, <, >, <=, >=.

ANY : la comparaison sera vraie si elle est vraie pour au moins un élément de l'ensemble (elle est donc fausse si l'ensemble est vide).

ALL : la comparaison sera vraie si elle est vraie pour tous les éléments de l'ensemble

(elle est vraie si l'ensemble est vide).
WHERE exp op ANY (SELECT ...)
WHERE exp op ALL (SELECT ...)
WHERE exp IN (SELECT ...)
WHERE exp NOT IN (SELECT ...)
où op est un des opérateurs =, !=, <, >, <=, >=.
Exemple
Liste des employés gagnant plus que tous les employés du département 30 :
SELECT NOME, SAL FROM EMP
WHERE SAL > ALL (SELECT SAL FROM EMP
WHERE DEPT=30)
Remarque
L'opérateur IN est équivalent à = ANY, et l'opérateur NOT IN est équivalent à != ALL.

5.6.3. Sous-interrogation synchronisée

Il est possible de synchroniser une sous-interrogation avec l'interrogation principale. Dans les exemples précédents, la sous-interrogation pouvait être évaluée d'abord, puis le résultat utilisé pour exécuter l'interrogation principale. SQL sait également traiter une sous-interrogation faisant référence à une colonne de la table de l'interrogation principale. Le traitement dans ce cas est plus complexe car il faut évaluer la sous-interrogation pour chaque ligne de l'interrogation principale.
Exemple
Liste des employés ne travaillant pas dans le même département que leur supérieur.
SELECT NOME FROM EMP E
WHERE DEPT != (SELECT DEPT FROM EMP
WHERE MATR = E.SUP)
Il a fallu renommer la table EMP de l'interrogation principale pour pouvoir la référencer dans la sous-interrogation.

5.6.4. Sous-interrogation ramenant plusieurs colonnes

Il est possible de comparer le résultat d'un SELECT ramenant plusieurs colonnes à une liste de colonnes. La liste de colonnes figurera entre parenthèses à gauche de l'opérateur de comparaison.
Avec une seule ligne sélectionnée :
WHERE (exp, exp,...) op (SELECT ...)
Avec plusieurs lignes sélectionnées :
WHERE (exp, exp,...) op ANY (SELECT ...)
WHERE (exp, exp,...) op ALL (SELECT ...)
WHERE (exp, exp,...) IN (SELECT ...)
WHERE (exp, exp,...) NOT IN (SELECT ...)
WHERE (exp, exp,...)
où op est un des opérateurs '=' ou '!='

Les expressions figurant dans la liste entre parenthèses seront comparées à celles qui sont ramenées par le SELECT.
Exemple
Employés ayant même poste et même salaire que MERCIER :

```
SELECT NOME, POSTE, SAL FROM EMP
WHERE (POSTE, SAL) =
(SELECT POSTE, SAL FROM EMP
WHERE NOME = 'MERCIER')
```

On peut utiliser ce type de sous-interrogation pour retrouver les lignes qui correspondent à des optima sur certains critères pour des regroupements de lignes.

5.6.5. Clause EXISTS

La clause EXISTS est suivie d'une sous-interrogation entre parenthèses, et prend la valeur vrai s'il existe au moins une ligne satisfaisant les conditions de la sous-interrogation.
Exemple

```
SELECT NOMD FROM DEPT
WHERE EXISTS (SELECT NULL FROM EMP
WHERE DEPT = DEPT.DEPT AND SAL > 10000);
```

Cette interrogation liste le nom des départements qui ont au moins un employé ayant plus de 10.000 comme salaire ; pour chaque ligne de DEPT la sous-interrogation synchronisée est exécutée et si au moins une ligne est trouvée dans la table EMP, EXISTS prend la valeur vrai et la ligne de DEPT satisfait les critères de l'interrogation. Souvent on peut utiliser IN à la place de la clause EXISTS. Essayez sur l'exemple précédent.
Remarque
Il faut se méfier lorsque l'on utilise EXISTS en présence de valeurs NULL. Si on veut par exemple les employés qui ont la plus grande commission par la requête suivante,

```
select nome from emp e1
where not exists
(select matr from emp
where comm > e1.comm)
```

on aura en plus dans la liste tous les employés qui ont une commission NULL.

5.6.6. Division avec la clause EXISTS

NOT EXISTS permet de spécifier des prédicats où le mot ’tous’ intervient dans un sens comparable à celui de l'exemple 4.16. Elle permet d'obtenir la division de deux relations. On rappelle que la division de R par S sur l'attribut B (notée R ÷B S, ou R ÷ S s'il n'y a pas d'ambiguïté sur l'attribut B) est la relation D définie par:D = {a 2 R[A] / 8 b 2 S, (a,b) 2 R} = {a 2 R[A] / 6 9 b 2 S, (a,b) 62 R}
Faisons une traduction "mot à mot" de cette dernière définition en langage SQL :

select A from R R1
where not exists
(select C from S
where not exists
(select A, B from R
where A = R1.A and B = S.C))

En fait, on peut remplacer les colonnes des select placés derrière des 'not exists' par ce que l'on veut, puisque seule l'existence ou non d'une ligne compte. On peut écrire par exemple :

select A from R R1
where not exists
(select null from S
where not exists
(select null from R
where A = R1.A and B = S.C))

On arrive souvent à optimiser ce type de select en utilisant les spécificités du cas, le plus souvent en simplifiant le select externe en remplaçant une jointure de tables par une seule table

Exemple

La réponse à la question #Quels sont les départements qui participent à tous les projets ?

- est fourni par R ÷Dept S où R = (PARTICIPATION JN{Matr} EMP) [Dept, CodeP] ("JN{Matr}" indique une jointure naturelle sur l'attribut Matr) et S = PROJET [CodeP]

Il reste à faire la traduction 'mot à mot' en SQL :

SELECT DEPT
FROM PARTICIPATION NATURAL JOIN EMP E1
WHERE NOT EXISTS
(SELECT CODEP FROM PROJET
WHERE NOT EXISTS
(SELECT DEPT, CODEP
FROM PARTICIPATION NATURAL JOIN EMP
WHERE DEPT = E1.DEPT
AND CODEP = PROJET.CODEP))

Remarque

Il faudrait ajouter DISTINCT dans le premier select pour éviter les doublons. Sur ce cas particulier on voit qu'il est inutile de travailler sur la jointure de PARTICIPATION et de EMP pour le SELECT externe. On peut travailler sur la table DEPT. Il en est de même sur tous les cas où la table "R" est une jointure. D'après cette remarque, le SELECT précédent devient :

SELECT DEPT FROM DEPT

WHERE NOT EXISTS
(SELECT CODEP FROM PROJET
WHERE NOT EXISTS
(SELECT DEPT, CODEP
FROM PARTICIPATION NATURAL JOIN EMP
WHERE DEPT = DEPT.DEPT
AND CODEP = PROJET.CODEP))

5.7. Fonctions de groupes

Les fonctions de groupes peuvent apparaître dans le Select ou le Having ; ce sont les fonctions suivantes :
AVG moyenne
SUM somme
MIN plus petite des valeurs
MAX plus grande des valeurs
VARIANCE variance
STDDEV écart type (déviation standard)
COUNT(*) nombre de lignes
COUNT(col) nombre de valeurs non nulles de la colonne
COUNT(DISTINCT col) nombre de valeurs non nulles différentes
Exemples 4.17
(a) **SELECT COUNT(*) FROM EMP**
(b) **SELECT SUM(COMM) FROM EMP WHERE DEPT = 10**
Les valeurs NULL sont ignorées par les fonctions de groupe. Ainsi, SUM(col) est la somme des valeurs qui ne sont pas égales à NULL de la colonne 'col'. De même, AVG est la somme des valeurs non "NULL" divisée par le nombre de valeurs non "NULL".
Il faut remarquer qu'à un niveau de profondeur (relativement aux sous-interrogations), d'un SELECT, les fonctions de groupe et les colonnes doivent être toutes du même niveau de regroupement. Par exemple, si on veut le nom et le salaire des employés qui gagnent le plus dans l'entreprise, la requête suivante provoquera une erreur :
SELECT NOME, SAL FROM EMP
WHERE SAL = MAX(SAL)
Il faut une sous-interrogation car MAX(SAL) n'est pas au même niveau de regroupement que le simple SAL :
SELECT NOME, SAL FROM EMP
WHERE SAL = (SELECT MAX(SAL) FROM EMP)

5.8. Clause GROUP BY

Il est possible de subdiviser la table en groupes, chaque groupe étant l'ensemble des lignes ayant une valeur commune.

GROUP BY exp1, exp2,...
groupe en une seule ligne toutes les lignes pour lesquelles exp1, exp2,... ont la même valeur. Cette clause se place juste après la clause WHERE, ou après la clause FROM si la clause WHERE n'existe pas. Des lignes peuvent être éliminées avant que le groupe ne soit formé grâce à la clause WHERE.
Exemples 4.18
(a) **SELECT DEPT, COUNT(*) FROM EMP**
GROUP BY DEPT
(b) **SELECT DEPT, COUNT(*) FROM EMP**
WHERE POSTE = 'SECRETAIRE'
GROUP BY DEPT
(c) **SELECT DEPT, POSTE, COUNT(*) FROM EMP**
GROUP BY DEPT, POSTE
(d) **SELECT NOME, DEPT FROM EMP**
WHERE (DEPT, SAL) IN
(SELECT DEPT, MAX(SAL) FROM EMP
GROUP BY DEPT)
RESTRICTION :
Une expression d'un SELECT avec clause GROUP BY ne peut évidemment que correspondre à une caractéristique de groupe. SQL n'est pas très 'intelligent' pour comprendre ce qu'est une caractéristique de groupe ; une expression du SELECT ne peut être que :
- soit une fonction de groupe,
- soit une expression figurant dans le GROUP BY.
L'ordre suivant est invalide car NOMD n'est pas une expression du GROUP BY :
SELECT NOMD, SUM(SAL)
FROM EMP NATURAL JOIN DEPT
GROUP BY DEPT
Il faut, soit se contenter du numéro de département au lieu du nom :
SELECT DEPT, SUM(SAL)
FROM EMP NATURAL JOIN DEPT
GROUP BY DEPT
Soit modifier le GROUP BY pour avoir le nom du département :
SELECT NOMD, SUM(SAL)
FROM EMP NATURAL JOIN DEPT
GROUP BY NOMD

5.9. Clause HAVING

HAVING prédicat sert à préciser quels groupes doivent être sélectionnés. Elle se place après la clause GROUP BY. Le prédicat suit la même syntaxe que celui de la clause WHERE. Cependant,

il ne peut porter que sur des caractéristiques de groupe : fonction de groupe ou expression figurant dans la clause GROUP BY.
Exemple

```
SELECT DEPT, COUNT(*)
FROM EMP
WHERE POSTE = 'SECRETAIRE'
GROUP BY DEPT HAVING COUNT(*) > 1
```

On peut évidemment combiner toutes les clauses, des jointures et des sous-interrogations. La requête suivante donne le nom du département (et son nombre de secrétaires) qui a le plus de secrétaires :

```
SELECT NOMD Departement, COUNT(*) "Nombre de secretaires"
FROM EMP NATURAL JOIN DEPT
WHERE POSTE = 'SECRETAIRE'
GROUP BY NOMD HAVING COUNT(*) =
(SELECT MAX(COUNT(*)) FROM EMP
WHERE POSTE = 'SECRETAIRE'
GROUP BY DEPT)
```

On remarquera que la dernière sous-interrogation est indispensable car MAX(COUNT(*)) n'est pas au même niveau de regroupement que les autres expressions du premier SELECT.

5.10. Fonctions

Nous allons décrire ci-dessous les principales fonctions disponibles dans Oracle. Il faut remarquer que ces fonctions ne sont pas standardisées et ne sont pas toutes disponibles dans les autres SGBD; elles peuvent aussi avoir une syntaxe différente,ou même un autre nom.

5.10.1. Fonctions arithmétiques

ABS(n) valeur absolue de n
MOD(n1, n2) n1 modulo n2
POWER(n, e) n à la puissance e
ROUND(n[, p]) arrondit n à la précision p (0 par défaut)
SIGN(n) -1 si n<0, 0 si n=0, 1 si n>0
SQRT(n) racine carrée de n
TRUNC(n[, p]) tronque n à la précision p (0 par défaut)
GREATEST(n1, n2,...) maximum de n1, n2,...
LEAST(n1, n2,...) minimum de n1, n2,...
TO_CHAR(n, format) convertit n en chaîne de caractères
TO_NUMBER(chaîne) convertit la chaîne de caractères en numérique
Exemple
Calcul du salaire journalier :

SELECT NOME, ROUND(SAL/22, 2) FROM EMP

5.11. Clause ORDER BY

Les lignes constituant le résultat d'un SELECT sont obtenues dans un ordre indéterminé. La clause ORDER BY précise l'ordre dans lequel la liste des lignes sélectionnées sera donnée.
ORDER BY exp1 [DESC], exp2 [DESC], ...
L'option facultative DESC donne un tri par ordre décroissant. Par défaut, l'ordre est croissant. Le tri se fait d'abord selon la première expression, puis les lignes ayant la même valeur pour la première expression sont triées selon la deuxième, etc.
Les valeurs nulles sont toujours en tête quel que soit l'ordre du tri (ascendant ou descendant). Pour préciser lors d'un tri sur quelle expression va porter le tri, il est possible de donner le rang relatif de la colonne dans la liste des colonnes, plutôt que son nom. Il est aussi possible de donner un nom d'en-tête de colonne du SELECT.
Exemple
À la place de : **SELECT DEPT, NOMD FROM DEPT ORDER BY NOMD**
on peut taper : **SELECT DEPT, NOMD FROM DEPT ORDER BY 2**
Cette nouvelle syntaxe doit être utilisée pour les interrogations exprimées à l'aide d'un opérateur booléen UNION, INTERSECT ou MINUS.
Elle permet aussi de simplifier l'écriture d'un tri sur une colonne qui contient une expression complexe.
Exemples
(a) Liste des employés et de leur poste, triée par département et dans
chaque département par ordre de salaire décroissant :
SELECT NOME, POSTE FROM EMP ORDER BY DEPT, SAL DESC
(b) **SELECT DEPT, SUM(SAL) "Total salaires"**
FROM EMP GROUP BY DEPT
ORDER BY 2
(c) **SELECT DEPT, SUM(SAL) "Total salaires"**
FROM EMP
GROUP BY DEPT
ORDER BY SUM(SAL)
(d) **SELECT DEPT, SUM(SAL) "Total salaires" FROM EMP**
GROUP BY DEPT
ORDER BY "Total salaires"

5.12. Opérateurs ensemblistes

Pour cette section on suposera que deux tables EMP1 et EMP2 contiennent les informations sur deux filiales de l'entreprise.

5.12.1. Opérateur UNION

L'opérateur UNION permet de fusionner deux sélections de tables pour

obtenir un ensemble de lignes égal à la réunion des lignes des deux sélections.
Les lignes communes n'apparaîtront qu'une fois.
Exemple
Liste des ingénieurs des deux filiales :
SELECT * FROM EMP1 WHERE POSTE='INGENIEUR' UNION SELECT * FROM EMP
WHERE POSTE='INGENIEUR'

5.13. Vues

Une vue est une vision partielle ou particulière des données d'une ou plusieurs tables de la base.
La définition d'une vue est donnée par un SELECT qui indique les données de la base qui seront vues.
Les utilisateurs pourront consulter la base, ou modifier la base (avec certaines restrictions) à travers la vue, c'est-à-dire manipuler les données renvoyées par la vue comme si c'était des données d'une table réelle.
Seule la définition de la vue est enregistrée dans la base, et pas les données de la vue. On peut parler de table virtuelle.

5.13.1. 5.3.1 CREATE VIEW

La commande CREATE VIEW permet de créer une vue en spécifiant le
SELECT constituant la définition de la vue :
CREATE VIEW vue (col1, col2...) AS SELECT ...
La spécification des noms des colonnes de la vue est facultative : par défaut, les colonnes de la vue ont pour nom les noms des colonnes résultat du SELECT. Si certaines colonnes résultat du SELECT sont des expressions sans nom, il faut alors obligatoirement spécifier les noms de colonnes de la vue.
Le SELECT peut contenir toutes les clauses d'un SELECT, sauf la clause ORDER BY.
Exemple
Vue constituant une restriction de la table EMP aux employés du département 10 :
CREATE VIEW EMP10 AS SELECT * FROM EMP WHERE DEPT = 10
Remarque
Dans l'exemple ci-dessus il aurait été plus prudent et plus souple d'éviter d'utiliser "*" et de le remplacer par les noms des colonnes de la table EMP. En effet, si la définition de la table EMP est modifiée, il y aura une erreur à l'exécution si on ne reconstruit pas la vue EMP10.

5.13.2. DROP VIEW

DROP VIEW vue
supprime la vue "vue".

5.13.3. Utilisation des vues

Une vue peut être référencée dans un SELECT de la même façon qu'une table. Ainsi, il est possible de consulter la vue EMP10. Tout se passe comme s'il existait une table EMP10 des employés du département 10 :

SELECT * FROM EMP10

Mise à jour avec une vue

Sous certaines conditions, il est possible d'effectuer des DELETE, INSERT et des **UPDATE** à travers des vues.

Les conditions suivantes doivent être remplies :

- pour effectuer un **DELETE**, le select qui définit la vue ne doit pas comporter de jointure, de **group by**, de distinct, de fonction de groupe ;
- pour un **UPDATE**, en plus des conditions précédentes, les colonnes modifiées doivent être des colonnes réelles de la table sous-jacente ;
- pour un **INSERT**, en plus des conditions précédentes, toute colonne "not null" de la table sous-jacente doit être présente dans la vue.

Ainsi, il est possible de modifier les salaires du département 10 à travers la vue EMP10. Toutes les lignes de la table EMP avec DEPT = 10 seront modifiées :

UPDATE EMP10 SET SAL = SAL * 1.1

Une vue peut créer des données qu'elle ne pourra pas visualiser. On peut ainsi ajouter un employé du département 20 avec la vue EMP10. Si l'on veut éviter cela il faut ajouter "WITH CHECK OPTION" dans l'ordre de création de la vue après l'interrogation définissant la vue. Il est alors interdit de créer au moyen de la vue des lignes qu'elle ne pourrait relire. Ce dispositif fonctionne également pour les mises à jour.

Exemple

```
CREATE VIEW EMP10 AS
SELECT * FROM EMP
WHERE DEPT = 10
WITH CHECK OPTION
```

5.13.4. Utilité des vues

De façon générale, les vues permettent de dissocier la façon dont les utilisateurs voient les données, du découpage en tables. On sépare l'aspect externe (ce que voit un utilisateur particulier de la base) de l'aspect conceptuel (comment a été conçu l'ensemble de la base). Ceci favorise l'indépendance entre les programmes et les données. Si la structure des données est modifiée, les programmes ne seront pas à modifier si l'on a pris la précaution d'utiliser des vues (ou si on peut se ramener à travailler sur des vues). Par exemple, si une table est découpée en plusieurs tables après l'introduction de nouvelles données, on peut introduire une vue, jointure des nouvelles tables, et la nommer du nom de l'ancienne table pour éviter de réécrire les programmes qui utilisaient l'ancienne table. Une vue peut aussi être utilisée pour restreindre les droits d'accès à certaines colonnes et à certaines lignes d'une table : un utilisateur peut ne pas avoir accès à une table mais avoir les autorisations pour utiliser une vue qui ne contient que certaines colonnes de la table ; on peut de plus ajouter des restrictions d'utilisation sur cette vue Dans le même ordre d'idées, une vue peut être

utilisée pour implanter une contrainte d'intégrité grâce à l'option "**WITH CHECK OPTION**".
Une vue peut également simplifier la consultation de la base en enregistrant des SELECT complexes.
Exemple
En créant la vue
CREATE VIEW EMPSAL AS
SELECT NOME, SAL + NVL(COMM, 0) GAINS, NOMD

5.6 Procédure stockée

Une procédure stockée est un programme qui comprend des instructions SQL précompilées et qui est enregistré dans la base de données (plus exactement dans le dictionnaire des données, notion étudiée en 5.8).

Le plus souvent le programme est écrit dans un langage spécial qui contient à la fois des instructions procédurales et des ordres SQL. Ces instructions ajoutent les possibilités habituelles des langages dits de troisième génération comme le langage C ou le Pascal (boucles, tests, fonctions et procédures,...).

Oracle o#re ainsi le langage PL/SQL qui se rapproche de la syntaxe du langage Ada et qui inclut des ordres SQL. Malheureusement aucun de ces langages (ni la notion de procédure stockée) n'est standardisé et ils sont donc liés à chacun des SGBD. Les procédures stockées d'Oracle peuvent aussi être écrites en Java.

Les procédures stockées o#rent des gros avantages pour les applications client/serveur, surtout au niveau des performances :

le tra#c sur le réseau est réduit car les clients SQL ont seulement à envoyer l'identi#cation de la procédure et ses paramètres au serveur sur lequel elle est stockée.

les procédures sont précompilées une seule fois quand elles sont enregistr ées. L'optimisation a lieu à ce moment et leurs exécutions ultérieures n'ont plus à passer par cette étape et sont donc plus rapides. De plus les erreurs sont repérées dès la compilation et pas à l'exécution.

Les développeurs n'ont pas à connaître les détails de l'exécution des procédures. Une procédure fonctionne en #boîte noire#. L'écriture et la maintenance des applications sont donc facilitées.

la gestion et la maintenance des procédures sont facilitées car elles sont enregistrées sur le serveur et ne sont pas dispersées sur les postes clients.

Le principal inconvénient des procédures stockées est qu'elles impliquent une dépendance forte vis-à-vis du SGBD car chaque SGBD a sa propre syntaxe et son propre langage de programmation.

En Oracle, on peut créer une nouvelle procédure stockée par la commande CREATE PROCEDURE.

Exemple 5.8

Voici une procédure stockée Oracle qui prend en paramètre un num éro de département et un pourcentage, augmente tous les salaires des employés de ce département de ce pourcentage et renvoie dans un paramètre le coût total pour l'entreprise.

```
create or replace procedure augmentation
(unDept in integer, pourcentage in number,
cout out number) is
begin
select sum(sal) * pourcentage / 100
into cout
```

```
from emp
where dept = unDept;
update emp
set sal = sal * (1 + pourcentage / 100)
where dept = unDept;
end;
```

Remarques 5.4

(a) Voici un ordre SQL pour avoir, sous Oracle, les noms des procédures stockées et le nom de leur propriétaire :

```
select owner, object_name
from all_objects
where object_type = 'PROCEDURE'
order by owner, object_name
```

(b) Sous Oracle, on peut avoir une description des paramètres d'une procédure stockée par la commande "DESC nom-procédure'.

(c) ... et le code de la procédure est donné par :

```
select text
from user_source
where name = 'nom-procedure'
order by line
```

(d) Les erreurs de compilation des procédures stockées peuvent être vues sous Oracle par la commande SQL*PLUS # show errors#.

On peut aussi créer des fonctions que l'on peut ensuite utiliser dans un ordre SQL comme une fonction prédé#nie :

```
create function triple(i in integer)
return integer is
begin
return 3 * i;
end;
select matr, triple(matr) from emp
```

5.7 Trigger

Les triggers (déclencheurs en français) ressemblent aux procédures stock ées car ils sont eux aussi enregistrés dans le dictionnaire des données de la base et ils sont le plus souvent écrits dans le même langage. La di#érence est que leur exécution est déclenchée automatiquement par des événements liés à des actions sur la base. Les événements déclencheurs peuvent être les commandes LMD insert, update, delete ou les commandes LDD create, alter, drop.

Les triggers complètent les contraintes d'intégrité en permettant des contrôles et des traitements plus complexes. Par exemple, on peut implanter la règle qu'on ne peut pas baisser le salaire d'un employé (on peut toujours rêver).

Pour des contraintes très complexes, on peut utiliser des procédures stockées qui encapsulent les requêtes SQL.

Si un trigger génère une erreur, par exemple s'il viole une contrainte d'intégrité, la requête qui l'a déclenché est annulée (mais pas la transaction en cours).
Les triggers sont normalisés dans la norme SQL3.
Les triggers peuvent aussi être utilisés pour d'autres usages comme de mettre à jour des données de la base suite à une modi#cation d'une donnée.
Exemple 5.9
Le trigger suivant met à jour automatiquement une table cumul qui totalise les augmentations de salaire de chaque employé.

```
CREATE OR REPLACE TRIGGER totalAugmentation
AFTER UPDATE OF sal ON emp
FOR EACH ROW
update cumul
set augmentation = augmentation + :NEW.sal - :OLD.sal
where matricule = :OLD.matr
```

Il faudra aussi créer un autre trigger qui ajoute une ligne dans la table cumul quand un employé est créé :

```
CREATE OR REPLACE TRIGGER creetotal AFTER INSERT ON emp
for each row
insert into cumul (matricule, augmentation)
values (:NEW.matr, 0)
```

L'option #for each row# est facultative ; elle indique que le traitement du trigger (la commande update) doit être exécutée pour chaque ligne modi#ée par la commande update. Sinon, cette commande n'est exécutée qu'une seule fois par update qui a déclenché le trigger.
Les pseudo-variables :NEW et :OLD permettent de se référer aux anciennes et nouvelles valeurs des lignes. :NEW a la valeur NULL après une commande delete et :OLD a la valeur NULL après une commande insert.
Un trigger peut être lancé avant (BEFORE) ou après (AFTER) la commande déclenchante.
Sous Oracle, comme pour les procédures stockées, les erreurs de compilation des triggers s'a#chent avec la commande SQL*PLUS # show errors#.
Pour supprimer un trigger :

```
drop trigger nomTrigger;
```