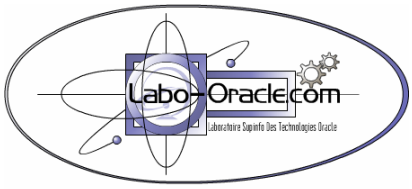

Cours Oracle - SQLA

Module n°
SQL Avancé

ORACLE

Programme de Formation de Supinfo

Laboratoire Supinfo des Technologies Oracle
Auteur : Thibault Blanchard
Date : 01/01/1601 01:00 - Version 1.2
Nombre de Page : 32
http://www.labo-oracle.com

Ecole Supérieure d'Informatique
23, rue Château Landon
75010 PARIS
http://www.supinfo.com

1 UTILISATION DE FONCTION DE GROUPE AVEC LES OPERATEURS CUBE ET ROLLUP

1.1 Rappel sur les fonctions de groupe

Cf. Cours SQLP « Module 2 : Techniques de récupération de données » § 2 « Les fonctions de groupe »

Commentaire : Il serait intéressant de faire pointer ceci vers le document adéquat sur le site du labo.
21/08 : Impossible car il faut un compte pour rentrer sur le site.

1.2 Fonctions de groupes avancées

1.2.1 GROUP BY avec les opérateurs ROLLUP ET CUBE

Les fonctions CUBE et ROLLUP sont utilisées avec GROUP BY pour obtenir des super agrégats de lignes par références croisées aux colonnes.
Les opérations de ROLLUP et de CUBE sont spécifiées dans la clause GROUP BY d'une requête.
Le groupement ROLLUP retourne le même résultat q'une requête avec un GROUP BY, mais il retourne également des lignes de résultat prenant en compte tous les sur ensembles.
Le groupement CUBE retourne le même résultat que le groupement ROLLUP mais il retourne également des lignes de résultat prenant en compte tous les sous-ensemble.

1.2.2 L'opérateur ROLLUP

L'opérateur ROLLUP est une extension de la clause GROUP BY qui permet de produire des agrégats cumulatifs tels que des sous totaux.

```
SELECT          column, group_function
FROM            table
[WHERE          condition]
[GROUP BY       [ROLLUP] (group_by_expression)];
```

L'opérateur ROLLUP crée des groupements en parcourant dans une direction, « de la droite vers la gauche », la liste de colonnes spécifiée dans la clause GROUP BY. Il effectue ensuite la fonction de groupe à ces groupements.

1.2.3 Exemple d'utilisation de ROLLUP

L'opérateur ROLLUP crée des sous totaux allant du niveau le plus détaillé à un total général, suivant la liste de groupement spécifié. Il calcule d'abord les valeurs standards de la fonction de groupe pour les groupements spécifiés dans la clause GROUP BY, puis il crée des sous totaux pour les sur ensembles, en parcourant la liste des colonnes de droite a gauche.
Pour deux arguments dans l'opérateur ROLLUP d'un GROUP BY, la requête retournera *n*+1=2+1=3 groupements. Les lignes résultant des valeurs des *n* premiers arguments sont appelées lignes originales et les autres sont appelées lignes de grand ensemble.

Exemple:

```
SQL> SELECT       deptno,job, SUM(sal)
  2  FROM         emp
  3  GROUP BY     ROLLUP(deptno,job);

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        10                 8750
        20 ANALYST         6000
        20 CLERK           1900
        20 MANAGER         2975
        20                10875
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        5600

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        30                 9400
                          29025

13 ligne(s) sélectionnée(s).
```

-> Cette requête affiche la somme des salaires pour chaque fonction dans chaque département ainsi que la somme des salaires pour chaque département et pour tous les départements.

1.2.4 L'opérateur CUBE

L'opérateur CUBE est une extension de la clause GROUP BY qui permet de retourner des valeurs de sous ensembles avec un ordre SELECT et qui peut être utilisé avec toutes les fonctions de groupe.

```
SELECT          column, group_function
FROM            table
[WHERE          condition]
[GROUP BY       [CUBE] (group_by_expression)];
```

Alors que ROLLUP ne retourne qu'une partie des combinaisons de sous totaux possibles, CUBE retourne toutes les combinaisons possibles des groupes spécifiés dans le GROUP BY ainsi qu'un total. Toutes les valeurs retournées sont calculées à partir de la fonction de groupe spécifiée dans la liste de SELECT.

1.2.5 Exemple d'utilisation de CUBE

L'opérateur CUBE retourne donc le même résultat que ROLLUP mais il y a en plus la fonction de groupe appliquée au sous groupe.
Le nombre de groupes supplémentaires dans le résultat est déterminé par le nombre de colonnes inclues dans la clause GROUP BY, car chaque combinaison de colonnes est utilisée pour produire de grands ensembles. Donc si il y a *n* colonnes ou expressions dans le GROUPY BY, il y aura 2^n combinaisons de grands ensembles possibles.

Exemple :

```
SQL> SELECT       deptno, job, SUM(sal)
  2  FROM         emp
  3  GROUP BY CUBE(deptno, job);

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        10                 8750
        20 ANALYST         6000
        20 CLERK           1900
        20 MANAGER         2975
        20                10875
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        5600
        30                 9400
           ANALYST         6000
           CLERK           4150
           MANAGER         8275
           PRESIDENT       5000
           SALESMAN        5600
                          29025
```

->Cette requête retourne le même résultat qu'avec l'opérateur ROLLUP en y ajoutant la somme des salaires pour chaque job.

1.2.6 La fonction GROUPING

Lors de l'utilisation des opérateurs ROLLUP et CUBE, des champs vides apparaissent dans la présentation du résultat. Pour ne pas confondre ces champs avec des valeurs NULL il existe la fonction GROUPING. Elle permet également de déterminer le niveau du sous total, c'est-à-dire le ou les groupes à partir desquels sont calculés les sous totaux.

```
SELECT          column, group_function, GROUPING(expr)
FROM            table
[WHERE          condition]
[GROUP BY       [ROLLUP][CUBE] (group_by_expression)];
```

La fonction GROUPING ne peut recevoir qu'une seule colonne en argument. Cet argument doit être le même qu'une des expressions de la clause GROUP BY.

1.2.7 Exemple d'utilisation de GROUPING

GROUPING se comporte comme une fonction booléenne.
Elle renvoie 0 quand :

- ➔ Le champ a été utilisé pour calculer le résultat de la fonction
- ➔ La valeur NULL dans le champ correspond à une valeur NULL dans la table.

Elle renvoie 1 quand :

- ➔ Le champ n'a pas été utilisé pour calculer le résultat de la fonction
- ➔ La valeur NULL dans le champ a été créée par ROLLUP/CUBE, à la suite d'un groupement

Exemple :

```
SQL> SELECT       deptno, job, SUM(sal), GROUPING(deptno),
..2               GROUPING(job)
  3  FROM         emp
  4  GROUP BY ROLLUP(deptno, job);

    DEPTNO JOB         SUM(SAL) GROUPING(DEPTNO) GROUPING(JOB)
---------- --------- ---------- ---------------- -------------
        10 CLERK           1300                0             0
        10 MANAGER         2450                0             0
        10 PRESIDENT       5000                0             0
        10                 8750                0             1
        20 ANALYST         6000                0             0
        20 CLERK           1900                0             0
        20 MANAGER         2975                0             0
        20                10875                0             1
        30 CLERK            950                0             0
        30 MANAGER         2850                0             0
        30 SALESMAN        5600                0             0

    DEPTNO JOB         SUM(SAL) GROUPING(DEPTNO) GROUPING(JOB)
---------- --------- ---------- ---------------- -------------
        30                 9400                0             1
                          29025                1             1

13 ligne(s) sélectionnée(s).
```

-> Cette requête affiche deux colonnes GROUPING qui indiquent si les champs deptno et job ont été calculés avec l'opérateur ROLLUP. Lorsque que la valeur renvoyée est 1 cela signifie que la colonne en argument n'a pas été prise en compte et donc que ce n'est pas une valeur NULL.

Quand la fonction GROUPING est utilisée avec l'opérateur CUBE, la colonne GROUPING(deptno) contiendra un 1 et la colonne GROUPING(job) un 0 car la colonne deptno n'est pas prise en compte pour calculer la somme du sous groupe

1.3 Les fonctions analytiques

1.3.1 Description des fonctions analytiques

Pour faciliter la programmation avancée en SQL, Oracle8*i* release 2 a introduit des fonctions analytiques pour faciliter les calculs tels que les moyennes variables et les classements.
Les groupes définis par la clause GROUP BY d'un ordre SELECT sont appelés *partition*. Le résultat d'une requête peut se composé d'une partition contenant toutes les lignes, quelques grandes partition, ou de nombreuses petites partitions contenant chacune peu de lignes. Les fonctions analytiques s'appliquent à chaque ligne dans chaque partition.

1.3.2 La fonction RANK

La fonction RANK crée un classement de lignes en commençant à 1.

```
SELECT         column,group_function(argument),
               RANK() OVER([PARTITION BY column] ORDER BY
               group_function(argument [DESC])
FROM           table
GROUP BY       column
```

Exemple :

```
SQL> SELECT       deptno, job, SUM(sal),
  2               RANK() OVER(PARTITION BY deptno
  3               ORDER BY SUM(sal) DESC)
  4               AS rank_of_job_per_dep,
  5               RANK() OVER(ORDER BY SUM(sal) DESC)
  6               AS rank_of_sumsal
  7  FROM         emp
  8  GROUP BY     deptno, job
  9  ORDER BY     deptno;


    DEPTNO JOB         SUM(SAL) GROUPING(DEPTNO) GROUPING(JOB)
---------- --------- ---------- ---------------- -------------
        10 CLERK           1300                0             0
        10 MANAGER         2450                0             0
        10 PRESIDENT       5000                0             0
        10                 8750                0             1
        20 ANALYST         6000                0             0
        20 CLERK           1900                0             0
        20                 7900                0             1
        30 CLERK            950                0             0
        30 SALESMAN        2750                0             0
        30                 3700                0             1
                          20350                1             1

11 ligne(s) sélectionnée(s).
```

-> Cette requête affiche le classement de la somme des salaires par département et pour tous les départements.

Le classement s'effectue sur les colonnes spécifiées dans le ORDER BY, si aucune partition n'est spécifiée le classement se fait sur toutes les colonnes. RANK assigne un rang de 1 a la plus petite valeur sauf lorsque que l'ordre DESC est utilisé.

1.3.3 La fonction CUME_DIST

La fonction de distribution cumulative calcule la position relative d'une valeur par rapport aux autres valeurs de la partition.

```
SELECT            column, group_function(argument),
                  CUME_DIST() OVER([PARTITION BY column]
                  ORDER BY group_function(argument [DESC])
FROM              table
GROUP BY          column
```

La fonction CUME_DIST détermine la partie des lignes de la partition qui sont inférieures ou égales à la valeur courante. Le résultat est une valeur décimale entre zéro et un inclus. Par défaut, l'ordre est ascendant, c'est-à-dire que la plus petite valeur de la partition correspond au plus petit CUME_DIST.

Exemple :

```
SQL> SELECT       deptno, job, SUM(sal),
  2               CUME_DIST() OVER(PARTITION BY deptno
  3               ORDER BY SUM(sal) DESC)
  4               AS cume_dist_per_dep
  5  FROM         emp
  6  GROUP BY     deptno, job
  7  ORDER BY     deptno, SUM(sal);

    DEPTNO JOB         SUM(SAL) CUME_DIST_PER_DEP
---------- --------- ---------- -----------------
        10 CLERK           1300                 1
        10 MANAGER         2450        ,666666667
        10 PRESIDENT       5000        ,333333333
        20 CLERK           1900                 1
        20 MANAGER         2975        ,666666667
        20 ANALYST         6000        ,333333333
        30 CLERK            950                 1
        30 MANAGER         2850        ,666666667
        30 SALESMAN        5600        ,333333333

9 ligne(s) sélectionnée(s).
```

-> Cette requête affiche le CUME_DIST de la somme des salaires pour chaque job dans chaque département.

2 RECUPERATION HIERARCHIQUE.

2.1 Aperçu des requêtes hiérarchiques

2.1.1 Dans quel cas utiliser une requête hiérarchique ?

Les requêtes hiérarchiques permettent de récupérer des données basées sur une relation hiérarchique naturelle entre les lignes dans une table.
Une base de donnée relationnelle ne stocke pas les données de manière hiérarchique. Cependant, quand une relation hiérarchique existe entre les lignes d'une seule table, il est possible de faire du *tree walking* qui permet de construire la hiérarchie. Une requête hiérarchique est un moyen d'afficher dans l'ordre les branches d'un arbre.

2.1.2 Structure en arbre

Une structure en arbre est constituée d'un nœud parent qui se divise en branches enfants qui elles-mêmes se divisent en branches et ainsi de suite. Par exemple la table EMP, qui représente le management d'une entreprise, a une structure en arbre avec aux nœuds les managers et sur les branches tous les employés qui leurs sont affectés. Cette hiérarchie est possible grâce aux colonnes EMPNO et MGR. La relation de hiérarchie est produite par un self-join : le numéro de manager d'un employé correspond au numéro d'employé de son manager.
La relation parent-enfant d'une structure en arbre permet de contrôler :

- ➔ La direction dans laquelle la hiérarchie est parcourue
- ➔ Le point de départ dans la hiérarchie

2.1.3 Requêtes hiérarchiques

Les requêtes hiérarchiques sont des requêtes contenant les clauses CONNECT BY et START WITH.

```
SELECT            [LEVEL], column, expr...
FROM              table
[WHERE            condition(s)]
[START WITH       condition(s)]
[CONNECT BY PRIOR condition(s)];
```

LEVEL est une pseudo colonne qui retourne 1 pour le nœud racine, 2 pour la branche de la racine et ainsi de suite. LEVEL permet de calculer jusqu'à quel niveau l'arbre a été parcouru.
START WITH permet de spécifier le nœud racine de la hiérarchie. Pour effectuer une vraie requête hiérarchique, cette clause est obligatoire.
CONNECT BY PRIOR permet de spécifier les colonnes pour lesquelles il existe une relation de parenté entre les lignes.
La requête SELECT ne doit pas contenir de jointure.

2.2 Parcourir l'arbre

2.2.1 Point de départ

Le point de départ du parcours est précisé dans la clause START WITH de la requête hiérarchique. START WITH peut être utilisée avec toutes les conditions valides et elle peut contenir une sous requête.
Si la clause START WITH n'est pas précisée dans la requête, le parcours de l'arbre est commencée avec toutes les lignes de la table en tant que nœud racine. Si la requête comporte une clause WHERE, toutes les

lignes satisfaisant la condition sont considérées comme nœuds racines. Cette structure ne reflète plus un arbre hiérarchique.
Les clauses START WITH et CONNECT BY PRIOR ne font pas partie du langage ANSI SQL standard.

2.2.2 Sens du parcours

Le sens de parcours de l'arbre est déterminé par les arguments de la clause CONNECT BY PRIOR. L'arbre peut être parcouru dans deux sens : de l'enfant vers le parent ou du parent vers l'enfant. L'opérateur PRIOR fait référence à la ligne parent.
Pour trouver l'enfant d'une ligne parent, le serveur Oracle examine le premier argument de PRIOR pour avoir la ligne parent et la compare avec toutes les lignes désignées par le deuxième argument. Les lignes pour lesquelles la condition est vérifiée sont le parent et l'enfant.
Le serveur Oracle choisit toujours les enfants en évaluant la condition du CONNECT BY et en respectant la ligne parent actuelle.
Par exemple pour parcourir la table EMP de haut en bas, il faut définir une relation hiérarchique dans laquelle la valeur EMPNO de la ligne parent est égale à la valeur MGR de la ligne enfant.

```
… CONNECT BY PRIOR       empno=mgr
```

La clause CONNECT BY ne peut pas contenir de sous requête.

2.2.3 Exemple de parcours

Exemple :

```
SQL> SELECT       empno, ename, job, mgr
  2  FROM         emp
  3  START WITH   empno = 7698
  4  CONNECT BY PRIOR mgr = empno;

     EMPNO ENAME      JOB              MGR
---------- ---------- --------- ----------
      7698 BLAKE      MANAGER         7839
      7839 KING       PRESIDENT
```

-> Cette requête reflète un parcours de bas en haut de la table EMP en commençant à l'employé 7698.

Les expressions en argument peuvent porter sur plusieurs colonnes. Mais dans ce cas l'opérateur PRIOR n'est valable que pour le premier argument.

Exemple :

```
SQL> SELECT       empno, ename, job, mgr
  2  FROM         emp
  3  START WITH   empno = 7698
  4  CONNECT BY PRIOR empno = mgr AND sal > NVL(comm,0);

     EMPNO ENAME      JOB              MGR
---------- ---------- --------- ----------
      7698 BLAKE      MANAGER         7839
      7499 ALLEN      SALESMAN        7698
      7521 WARD       SALESMAN        7698
      7844 TURNER     SALESMAN        7698
      7900 JAMES      CLERK           7698
```

-> Cette requête renvoie les lignes dont le numéro de manager est égal au numéro d'employé de la ligne parent et dont le salaire est supérieur à la commission.

2.3 Organiser les données

2.3.1 Classer les lignes avec la pseudo colonne LEVEL

La pseudo colonne LEVEL permet d'afficher explicitement le rang ou niveau d'une ligne dans la hiérarchie.
Cela permet de créer des rapports plus faciles à exploiter et à comprendre.
Dans un arbre l'endroit où une ou plusieurs branches se séparent d'une branche supérieure est appelée nœud et la fin d'une branche est appelée feuille ou nœud feuille. La valeur de la pseudo colonne LEVEL dépend du nœud sur lequel se trouve la ligne. La valeur de LEVEL sera 1 pour le nœud parent, puis 2 pour un enfant et ainsi de suite.
Le premier nœud d'un arbre est appelé nœud racine, alors que tous les autres sont appelés nœud enfant. Un nœud parent est un nœud ayant au moins un enfant. Un nœud feuille est un nœud ne possédant pas d'enfant.

2.3.2 Formatage d'un rapport hiérarchique à l'aide de LEVEL et LPAD

Il est possible de refléter la structure de la hiérarchie dans une seule colonne en combinant la pseudo colonne avec un LPAD (cf. cours SQLP « Module 1 : Ordres SELECT Basiques » § 4.2.2 « Les fonctions de manipulation de caractère »).
Cette représentation sera sous la forme d'un arbre indenté.

Exemple :

```
SQL> COLUMN org_chart FORMAT A15
SQL> SET PAGESIZE 20
SQL> SELECT       LPAD(' ', 3 * LEVEL-3)||ename AS org_chart,
  2               LEVEL, empno, mgr, deptno
  3  FROM         emp
  4  START WITH    mgr IS NULL
  5  CONNECT BY PRIOR empno = mgr;
SQL> CLEAR COLUMN

ORG_CHART            LEVEL      EMPNO        MGR     DEPTNO
--------------- ---------- ---------- ---------- ----------
KING                     1       7839                    10
   JONES                 2       7566       7839         20
      SCOTT              3       7788       7566         20
         ADAMS           4       7876       7788         20
      FORD               3       7902       7566         20
         SMITH           4       7369       7902         20
   BLAKE                 2       7698       7839         30
      ALLEN              3       7499       7698         30
      WARD               3       7521       7698         30
      MARTIN             3       7654       7698         30
      TURNER             3       7844       7698         30
      JAMES              3       7900       7698         30
   CLARK                 2       7782       7839         10
      MILLER             3       7934       7782         10

14 ligne(s) sélectionnée(s).
```

-> Cette requête ajoute des espaces devant les noms des employés en fonction de leur niveau dans la hiérarchie.

2.3.3 Eliminer une branche

Dans une requête hiérarchique il est possible de supprimer une branche ou un nœud de l'arbre.
Pour éliminer une branche il faut utiliser une condition WHERE. Ainsi la branche concernée n'est pas prise en compte mais les branches enfants situées après sont quand même affichées.

Exemple :

```
SQL> SELECT       deptno, empno, ename, job, sal
  2  FROM         emp
  3  WHERE        ename != 'SCOTT'
  4  START WITH   mgr IS NULL
  5  CONNECT BY PRIOR empno = mgr;

    DEPTNO      EMPNO ENAME      JOB              SAL
---------- ---------- ---------- --------- ----------
        10       7839 KING       PRESIDENT       5000
        20       7566 JONES      MANAGER         2975
        20       7876 ADAMS      CLERK           1100
        20       7902 FORD       ANALYST         3000
        20       7369 SMITH      CLERK            800
        30       7698 BLAKE      MANAGER         2850
        30       7499 ALLEN      SALESMAN        1600
        30       7521 WARD       SALESMAN        1250
        30       7654 MARTIN     SALESMAN        1250
        30       7844 TURNER     SALESMAN        1500
        30       7900 JAMES      CLERK            950
        10       7782 CLARK      MANAGER         2450
        10       7934 MILLER     CLERK           1300
```

-> Cette requête parcours l'arbre de haut en bas à partir du noeud racine sans afficher l'employé SCOTT, tout en conservant les branches enfants (ADAMS).

Pour éliminer un nœud il faut ajouter la condition dans la clause CONNECT BY. Ainsi ni le nœud concerné, ni les branches enfants de ce nœud ne sont affichées.

2.3.4 Ordonner les données

Il est possible d'ordonner les données d'une requête hiérarchique en utilisant la clause ORDER BY. Cependant ceci n'est pas conseillé car l'ordre naturel de l'arbre sera brisé.

2.3.5 La fonction ROW_NUMBER()

Par défaut, les valeurs NULL sont affichées en premier lors d'un classement décroissant. Avec Oracle8*i* release 2 il est possible de forcer les valeurs NULL à apparaître en dernier en ajoutant NULLS LAST dans la clause ORDER BY.
La fonction ROW_NUMBER donne à chaque ligne de la partition un numéro suivant la séquence définie dans la clause ORDER BY.

Exemple :

```
SQL> SELECT       empno, job, comm,
  2               ROW_NUMBER() OVER(ORDER BY comm DESC NULLS LAST)
  3               AS rnum
  4  FROM         emp;

     EMPNO JOB              COMM       RNUM
---------- --------- ---------- ----------
      7654 SALESMAN         1400          1
      7521 SALESMAN          500          2
      7499 SALESMAN          300          3
      7844 SALESMAN            0          4
      7369 CLERK                          5
      7566 MANAGER                        6
      7900 CLERK                          7
      7934 CLERK                          8
      7902 ANALYST                        9
      7876 CLERK                         10
      7698 MANAGER                       11
      7782 MANAGER                       12
      7788 ANALYST                       13
      7839 PRESIDENT                     14

14 ligne(s) sélectionnée(s).
```

-> Cette requête numérote les lignes en se basant sur la valeur de la commission et en plaçant les valeurs NULL à la fin.

3 ECRITURE DE SOUS REQUETES CORRELEES

3.1 Sous requêtes

Cf. Cours SQLP « Module 2 : Techniques de récupération de données » § 3 « Les sous requêtes »

3.2 Sous requêtes corrélées

3.2.1 Description des sous requêtes corrélées

Une sous requête corrélée est une sous requête imbriquée qui est évaluée pour chaque ligne traitée par la requête principale, et qui, lors de son exécution, utilise les valeurs d'une colonne retournée par la requête principale. La corrélation est obtenue en utilisant un élément de la requête externe dans la sous requête.
Etapes d'exécution d'une requête corrélée :

- ➔ Récupère la ligne à utilisée (renvoyée par la requête externe)
- ➔ Exécute la requête interne en utilisant la valeur de la ligne récupérée
- ➔ Utilise la valeur retournée par la requête interne pour conserver ou éliminer la ligne
- ➔ Recommence jusqu'à ce qu'il n'y ait plus de lignes

Les sous requêtes sont principalement utilisée dans des ordres SELECT mais elles peuvent aussi s'appliquer aux ordres UPDATE et DELETE.

3.2.2 Utilisation de requêtes corrélées

Une requête corrélée est un moyen de lire chaque ligne d'une table et de comparer la valeur trouvée avec une donnée correspondante d'une autre table.

Commentaire : Cette définition n'est pas claire et contient un « la » en trop. A reformuler.
21/08 : Corrigé

```
SELECT          outer1, outer2, ...
FROM            table1 alias1
WHERE           outer1 operator
                        (SELECT      inner1
                         FROM        table2 alias2
                         WHERE       alias1.expr1 =
                                     alias2.expr2);
```

Elle est utilisée quand une sous requête doit retourner différents résultats selon la ligne considérée par la requête principale, c'est-à-dire qu'on l'utilise pour répondre à une question complexe dont la réponse dépend de chaque ligne considérée par la requête principale.
Le serveur Oracle effectue une requête corrélée quand la sous requête fait appel à une colonne d'une table de la requête parent.

Exemple :

```
SQL> SELECT    empno, sal, deptno
  2  FROM         emp outer
  3  WHERE        sal > (SELECT       AVG(sal)
  4                      FROM         emp inner
  5                      WHERE        outer.deptno = inner.deptno);

     EMPNO         SAL     DEPTNO
---------- ---------- ----------
      7499       1600         30
      7566       2975         20
      7698       2850         30
      7788       3000         20
      7839       5000         10
      7902       3000         20

6 ligne(s) sélectionnée(s).
```

→ Cette requête affiche tous les employés qui ont un salaire supérieur au salaire moyen de leur département.

Quand les requêtes internes et externes font appel à la même table il faut utiliser des alias afin que les résultats ne soient pas faussés.

3.2.3 L'opérateur EXISTS

Dans les ordres SELECT imbriqués, tous les opérateurs logiques peuvent être utilisés mais en plus lorsqu'il s'agit d'ordres SELECT imbriqués, on peut utiliser l'opérateur EXISTS. Cet opérateur teste l'existence de lignes dans la sous requête.
Si la sous requête renvoie une ligne :

- ➔ La recherche ne continue pas dans la sous requête.
- ➔ La condition est évaluée TRUE

Si la sous requête ne renvoie pas de valeurs pour une ligne :

- ➔ La condition est évaluée FALSE
- ➔ La recherche se poursuit à la ligne suivante dans la sous requête

L'opérateur NOT EXISTS quant à lui teste si la valeur n'est pas trouvée.

Exemple :

```
SQL> SELECT    empno, ename, job, deptno
  2  FROM      emp outer
  3  WHERE     EXISTS (SELECT  'X'
  4                    FROM   emp inner
  5                    WHERE  inner.mgr = outer.empno);

     EMPNO ENAME      JOB           DEPTNO
---------- ---------- --------- ----------
      7566 JONES      MANAGER           20
      7698 BLAKE      MANAGER           30
      7782 CLARK      MANAGER           10
      7788 SCOTT      ANALYST           20
      7839 KING       PRESIDENT         10
      7902 FORD       ANALYST           20

6 ligne(s) sélectionnée(s).
```

→ Cette requête affiche tous les employés ayant au moins une personne à ses ordres.

Comme elle effectue uniquement un test, la requête interne n'a pas besoin de retourner de valeur donc on peut utiliser n'importe quel valeur littérale. Il est d'ailleurs conseillé d'utiliser une constante à la place d'une colonne afin d'accélérer la requête.

3.2.4 L'opérateur NOT EXISTS

Exemple :

```
SQL> SELECT    deptno, dname
  2  FROM      dept d
  3  WHERE     NOT EXISTS (SELECT     'X'
  4                        FROM       emp e
  5                        WHERE      d.deptno = e.deptno);

    DEPTNO DNAME
---------- --------------
        40 OPERATIONS
```

→ Cette requête affiche les départements dans lesquels il n'y a aucun employés.

Une structure NOT IN peut être utilisée en alternative à un opérateur NOT EXISTS. Cependant il faut faire attention car la condition sera évaluée FALSE si elle rencontre une valeur NULL.

Commentaire : Il serait judicieux d'ajouter ici deux exemples, l'un avec « not exists » et l'autre « not in » pour montrer dans quel cas il est trompeur d'utiliser ce dernier
22-08 : Corrigé

Exemple :

```
SQL> SELECT    o.ename
  2  FROM      emp o
  3  WHERE     NOT EXISTS (SELECT     'X'
  4                        FROM       emp i
  5                        WHERE      i.mgr = o.empno);

ENAME
----------
SMITH
ALLEN
WARD
MARTIN
TURNER
ADAMS
JAMES
MILLER

8 ligne(s) sélectionnée(s).
```

-> Cette requête affiche tous les employés n'ayant personne sous leurs ordres.

La requête de l'exemple peut être également construite avec NOT IN seulement le résultat sera totalement différent puisque les valeurs NULL seront prises en compte.

Exemple :

```
SQL> SELECT    o.ename
  2  FROM      emp o
  3  WHERE     o.empno NOT IN (SELECT       i.mgr
  4                            FROM         emp i);

Aucune ligne sélectionnée
```

→ Cette requête ne renvoie aucun résultat car la sous requête retourne une valeur NULL et comme toute comparaison avec une valeur NULL renvoie une valeur NULL, le résultat est complètement différent.

Il est donc conseillé, lorsque la sous requête retourne des valeurs NULL, de ne pas utiliser NOT IN en alternative de NOT EXISTS.

3.3 UPDATE corrélés

Commentaire : Un exemple concret ne ferait pas de mal ici non plus après la description de la syntaxe. Ce qu'on recherche souvent dans une doc ce sont les exemples donc donnons leurs des exemples.
22-08 : Corrigé

Une sous requête corrélée peut être utilisée dans le cas d'un ordre UPDATE pour mettre à jour les lignes d'une table en fonction des lignes d'une autre table.

```
UPDATE          table1 alias1
SET             column = (SELECT expression
                          FROM table2 alias2
                          WHERE alias1.column = alias2.column);
```

Exemple :

```
SQL> ALTER TABLE emp
  2  ADD        (dname VARCHAR2(14));

SQL> UPDATE     emp e
  2  SET        dname = (SELECT              dname
  3                      FROM         dept d
  4                      WHERE        e.deptno = d.deptno);

14 ligne(s) mise(s) à jour.
```

3.4 DELETE corrélés

Commentaire : Un exemple concret ne ferait pas de mal ici non plus après la description de la syntaxe. Ce qu'on recherche souvent dans une doc ce sont les exemples donc donnons leurs des exemples.
22-08 : Corrigé

Une sous requête corrélée peut être utilisée dans le cas d'un ordre DELETE pour effacer uniquement les lignes existantes dans une autre table.

```
DELETE FROM     table1 alias1
WHERE           column operator
                   (SELECT expression
                   FROM table2 alias2
                   WHERE alias1.column = alias2.column);
```

Exemple :

```
SQL> DELETE FROM emp E
  2  WHERE     empno =
  3                (SELECT      empno
  4                FROM         emp_history EH
  5                WHERE        E.empno = EH.empno);
```

→ Cette requête supprime de la table EMP uniquement les lignes qui sont déjà présentes dans la table EMP_HISTORY.

4 UTILISATION DES OPERATEURS D'ENSEMBLE

4.1 Les opérateurs d'ensemble

Un opérateur d'ensemble combine le résultat de deux requêtes ou plus en un seul résultat. Les requêtes utilisant ces opérateurs sont appelées *requêtes composées.*

Opérateur	Résultat
INTERSECT	Sélectionne toutes les lignes similaires retournées par les requêtes (INTERSECT combine deux requêtes et retourne uniquement le valeurs du premier SELECT qui sont identiques à au moins une de celles du second SELECT)
UNION	Renvoie toutes les lignes sélectionnées par les deux requêtes en excluant les lignes identiques
UNION ALL	Renvoie toutes les lignes sélectionnées par les deux requêtes en incluant les lignes identiques
MINUS	Renvoie toutes les lignes retournées par le premier SELECT qui ne sont pas retournées par le second SELECT

```
SELECT          column1, column2, ...
FROM            table1
SET OPERATOR
SELECT          column1, column2, ...
FROM            table2 ;
```

Tous les opérateurs d'ensemble ont une priorité égale. Si un ordre SQL contient plusieurs de ces opérateurs, le serveur les exécute de la droite vers la gauche (du plus imbriqué au moins imbriqué) si aucune parenthèse ne définit d'ordre explicite.
Les opérateurs INTERSECT et MINUS sont spécifiques à Oracle.

4.2 UNION et UNION ALL

Commentaire : Un exemple concret. Ce qu'on recherche souvent dans une doc ce sont les exemples donc donnons leurs des exemples.
22-08 : Corrigé

4.2.1 L'opérateur UNION

L'opérateur UNION combine le résultat de plusieurs requêtes en éliminant les lignes retournées par les deux requêtes.
Le nombre de colonnes et le type de données doivent être identiques dans les deux ordres SELECT. Les noms de colonnes peuvent être différents.
UNION agit sur toutes les colonnes sélectionnées
Les valeurs NULL ne sont pas ignorées lors de la vérification des doublons.
Les requêtes utilisant UNION dans la clause WHERE doivent avoir le même nombre de colonnes et le même type de données dans la liste SELECT.
Par défaut, le résultat sera trié selon un ordre ascendant en fonction de la première colonne de la liste de SELECT.

Exemple :

```
SQL> SELECT    ename, job, sal
  2  FROM      emp
  3  UNION
  4  SELECT    name, title, 0
  5  FROM      emp_history;

ENAME      JOB              SAL
---------- --------- ----------
ADAMS      CLERK           1100
ALLEN      SALESMAN           0
ALLEN      SALESMAN        1600
BALFORD    CLERK              0
BLAKE      MANAGER         2850
…
23 ligne(s) sélectionnée(s).
```

→ Cette requête affiche tous les résultats des deux requêtes sans afficher les lignes communes aux deux. De plus elle affiche un 0 lorsque la ligne provient de la table qui n'as pas de colonne correspondant au salaire.

Commentaire : Un exemple concret. Ce qu'on recherche souvent dans une doc ce sont les exemples donc donnons leurs des exemples.
22-08 : Corrigé

4.2.2 L'opérateur UNION ALL

L'opérateur UNION ALL retourne toutes les lignes d'une requête multiple sans éliminer les doublons. Contrairement à l'opérateur UNION, les résultats de la requête ne sont pas triés par défaut. Le mot clé DISTINCT ne peut être utilisé.

Exemple :

```
SQL> SELECT    ename, empno, job
  2  FROM      emp
  3  UNION ALL
  4  SELECT    name, empid, title
  5  FROM      emp_history
  6  ORDER BY  ename;

ENAME           EMPNO JOB
---------- ---------- ---------
ADAMS            7876 CLERK
ALLEN            7499 SALESMAN
ALLEN            7499 SALESMAN
BALFORD          6235 CLERK
BLAKE            7698 MANAGER
BRIGGS           7225 PAY CLERK
…
23 ligne(s) sélectionnée(s)
```

→ Cette requête affiche tous les résultats des deux requêtes, y compris les lignes présentent dans les deux tables.

4.2.3 Utilisation de UNION et UNION ALL

```
SQL> SELECT       ename, job, deptno
  2  FROM         emp
  3  UNION
  4  SELECT       name, title, deptid
  5  FROM         emp_history;

ENAME      JOB           DEPTNO
---------- --------- ----------
ADAMS      CLERK             20
ALLEN      SALESMAN          20
ALLEN      SALESMAN          30
BALFORD    CLERK             20
BLAKE      MANAGER           30
…
21 ligne(s) sélectionnée(s).
```

Commentaire : Je ne me souviens plus si cette table a été définie dans un de nos modules. En tous les cas le lecteur ne consultant que ce module ne connaît pas cette table. Lister donc son contenu.
22-08 : En annexe

→ Cette requête affiche le nom, la fonction et le numéro de département de tous les employés sans répétition. On peut noter que Allen est présent deux fois car il fait partie de deux départements différents donc la ligne n'est pas considérée comme identique.

Si l'on remplace UNION par UNION ALL, la requête renvoie 23 lignes puisqu'elle affiche tous les résultats en incluant les doublons.

4.3 INTERSECT

4.3.1 L'opérateur INTERSECT

L'opérateur INTERSECT retourne uniquement les résultats communs aux deux requêtes.
Le nombre de colonnes et le type des données doivent être identiques dans les deux ordres SELECT, mais les noms peuvent être différents.
L'inversion des tables dont on fait l'intersection ne change pas le résultat.
Comme UNION, INTERSECT n'ignore pas les valeurs NULL.
Les requêtes qui utilisent INTERSECT dans la clause WHERE doivent avoir le même nombre et le même type de colonnes que dans la liste de SELECT.

4.3.2 Utilisation de l'opérateur INTERSECT

Exemple :

```
SQL>  SELECT      ename, empno, job
  2   FROM        emp
  3   INTERSECT
  4   SELECT      name, empid, title
  5   FROM        emp_history;

ENAME           EMPNO JOB
---------- ---------- ---------
ALLEN            7499 SALESMAN
CLARK            7782 MANAGER
SCOTT            7788 ANALYST
```

→ Cette requête affiche les lignes ayant les mêmes ENAME, EMPNO et JOB dans les tables EMP et EMP_HISTORY

Si l'on ajoute une colonne dans la liste des SELECT il se peut que le résultat de la requête soit différent.

4.4 MINUS

4.4.1 L'opérateur MINUS

L'opérateur MINUS retourne les lignes résultantes de la première requête et qui ne se trouvent pas dans les résultats de la seconde (la première requête MOINS la seconde).
Comme pour les autres opérateurs il faut veiller à respecter le nombre de colonne et le type de données. Il faut également faire attention lors de l'utilisation avec la clause WHERE.

4.4.2 Utilisation de l'opérateur MINUS

```
SQL> SELECT       name, empid, title
  2  FROM         emp_history
  3  MINUS
  4  SELECT       ename, empno, job
  5  FROM         emp;

NAME            EMPID TITLE
---------- ---------- ---------
BALFORD          6235 CLERK
BRIGGS           7225 PAY CLERK
JEWELL           7001 ANALYST
SPENCER          6087 OPERATOR
VANDYKE          6185 MANAGER
WILD             7356 DIRECTOR

6 ligne(s) sélectionnée(s).
```

→ Cette requête affiche le nom, le numéro d'employé et la fonction des employés ayant quittés la compagnie (les employés présents dans la table EMP_HISTORY MOINS les employés de la table EMP)

4.5 Règles syntaxiques des opérateurs d'ensemble

4.5.1 Règles sur les opérateurs d'ensemble

Les expressions dans les listes de SELECT doivent avoir le même nombre de colonnes et le même type de données. Les noms des colonnes affichés dans le résultat sont ceux du premier SELECT.
Les lignes doublons sont automatiquement retirées excepté lors de l'utilisation de l'opérateur UNION ALL.
Les résultats sont triés en ordre ascendant par défaut sauf dans le cas de UNION ALL. La clause ORDER BY peut être utilisée mais uniquement en fin de requête. Comme les noms de colonnes affichés sont ceux de la première liste SELECT, l'argument de ORDER BY doit être un nom du premier SELECT.
Des parenthèses peuvent être utilisées pour modifier l'ordre d'exécution, qui par défaut va de l'opérateur le plus imbriqué au moins imbriqué.
Les requêtes utilisant un opérateur d'ensemble dans leur clause WHERE doivent avoir le même nombre et type de colonnes dans leur liste SELECT.

4.5.2 Faire correspondre la syntaxe des SELECT

Pour afficher une colonne qui n'a pas d'équivalent dans une des tables, on peut faire appel à la table DUAL et aux fonctions de conversions de données pour respecter la syntaxe. Dans le résultat de la requête, les lignes issues de la table n'ayant pas la colonne équivalente renverront l'expression littérale dans un des champs.

Exemple :

```
SQL> SELECT       deptno, TO_CHAR(null) AS location, hiredate
  2  FROM         emp
  3  UNION
```

Commentaire : Attention, l'exemple ne correspond pas au paragraphe. Où est DUAL dans cet exemple ?
22-08 : Dans l'exemple je démontre l'utilisation des fonctions de conversions de données. La table DUAL est utilisée dans l'exemple suivant.

```
  4  SELECT       deptno, loc, TO_DATE(null)
  5  FROM         dept;

    DEPTNO LOCATION      HIREDATE
---------- ------------- --------
        10 NEW YORK
        10               09/06/81
        10               17/11/81
        10               23/01/82
        20 DALLAS
        20               17/12/80
…
        30               28/09/81
        30               03/12/81

    DEPTNO LOCATION      HIREDATE
---------- ------------- --------
        40 BOSTON

18 ligne(s) sélectionnée(s).
```

→ Cette requête affiche le numéro de département, la location, et la date d'embauche pour tous les employés.

4.6 Contrôler l'ordre des lignes

Par défaut, le résultat est trié selon un ordre ascendant. Pour changer l'ordre de tri on peut utiliser la clause ORDER BY. Cette clause ne peut être utilisée qu'une seule fois dans une requête composée et doit être placée à la fin de la requête. La clause ORDER BY accepte en argument le nom de colonne, l'alias ou la position de la colonne.

Exemple :

```
SQL> COLUMN       a_dummy NOPRINT
SQL> SELECT       'chanter' AS "Mon rêve", 3 a_dummy
  2  FROM         dual
  3  UNION
  4  SELECT       'Je voudrais apprendre', 1
  5  FROM         dual
  6  UNION
  7  SELECT       'au monde à', 2
  8  FROM         dual
  9  ORDER BY     2;

Mon rêve
--------------------
Je voudrais apprendre
au monde à
chanter
```

→ L'utilisation de la colonne A_DUMMY permet de trier le résultat afin d'afficher une phrase correcte. La commande NOPRINT permet de ne pas afficher la colonne contenant le critère de tri.

5 ECRIRE DES SCRIPTS AVANCES

5.1 Utilisation de SQL pour générer du SQL

Le SQL est un outil puissant pour générer d'autres ordres SQL. Dans la plupart des cas cela implique l'écriture d'un script. Il est possible de générer du SQL dans SQL pour :

- ➔ Eviter le codage répétitif
- ➔ S'aider du dictionnaire de données
- ➔ Dropper ou recréer des objets de la base de données
- ➔ Générer des attributs dynamiques qui prennent en compte des paramètres du script en cours d'exécution.

La plupart du temps pour générer des ordres SQL on utilise des informations contenus dans le dictionnaire de données.
Le dictionnaire de données est une collection de tables et de vues qui contiennent des informations sur la base de données. Cette collection est créée et mise à jour par le serveur Oracle. L'utilisateur SYS est le propriétaire de ces tables.
Le dictionnaire de données stocke les noms d'utilisateurs du serveur Oracle, les privilèges accordés aux utilisateurs, le nom des objets de la base de données, les contraintes sur les tables et des informations d'audit.
Il y a quatre catégories de vues du dictionnaire de données. Chaque catégorie à un préfixe différent qui reflète son utilisation.

Préfixe	Description
USER_	Liste tous les objets dont l'utilisateur est propriétaire
ALL_	Liste tous les objets sur lesquels l'utilisateur a des droits d'accès, en plus des objets dont il est propriétaire
DBA_	Autorise les utilisateurs possédant des privilèges DBA à lister tous les objets de la base de données
V$_	Affiche les performances du serveur de base de données ; initialement disponible uniquement pour le DBA

5.2 Création d'un script basique

On utilise un script SQL pour générer des ordres SQL en se basant sur les résultats renvoyés par la requête. Cette requête peut se baser sur une table ou encore sur le dictionnaire de données.

Exemple :

```
SQL> SELECT       'CREATE TABLE ' || table_name || '_test '
  2               || 'AS SELECT * FROM ' || table_name
  3               ||' WHERE 1=2;'
  4               AS "Create Table Script"
  5  FROM         user_tables;

Create Table Script
--------------------------------------------------------------------------------
CREATE TABLE ARTICLES_test AS SELECT * FROM ARTICLES WHERE 1=2;
CREATE TABLE BONUS_test AS SELECT * FROM BONUS WHERE 1=2;
CREATE TABLE CLIENTS_test AS SELECT * FROM CLIENTS WHERE 1=2;
CREATE TABLE COMMANDES_test AS SELECT * FROM COMMANDES WHERE 1=2;
CREATE TABLE CUSTOMER_test AS SELECT * FROM CUSTOMER WHERE 1=2;
CREATE TABLE DEPT_test AS SELECT * FROM DEPT WHERE 1=2;
CREATE TABLE DUMMY_test AS SELECT * FROM DUMMY WHERE 1=2;
CREATE TABLE EMP_test AS SELECT * FROM EMP WHERE 1=2;
CREATE TABLE EMP_HISTORY_test AS SELECT * FROM EMP_HISTORY WHERE 1=2;
CREATE TABLE LIGNES_test AS SELECT * FROM LIGNES WHERE 1=2;
CREATE TABLE SALGRADE_test AS SELECT * FROM SALGRADE WHERE 1=2;

11 ligne(s) sélectionnée(s).
```

→ Cette requête génère une ligne de création d'une nouvelle table pour chaque table dont l'utilisateur est propriétaire.

La requête de l'exemple utilise la vue USER_TABLE afin de connaître le nom des tables de l'utilisateur. Parmi les vues couramment utilisées il y a également USER_OBJECTS, USER_TAB_PRIVS_MADE et USER_COL_PRIVS_MADE.

5.3 Contrôler l'environnement

Pour pouvoir exécuter les requêtes SQL qui ont été générées il faut les envoyer vers un fichier en utilisant la fonction SPOOL. Mais pour que l'on puisse réutiliser ce fichier en l'exécutant directement il faut que le résultat envoyé respecte un certain format. Pour cela il faut contrôler l'environnement, c'est-à-dire redéfinir les paramètres d'affichage sur le terminal. Il faut principalement éliminer les en-têtes de colonnes, les commentaires et les titres de pages.
Le contrôle de l'environnement se fait grâce aux commandes suivantes :

Variable Système	Description
TERMOUT	Contrôle l'affichage du résultat généré par l'exécution d'un fichier de commande
PAGESIZE	Contrôle le nombre de lignes par page (La définir à 0 supprime les en-têtes, les sauts de page, les titres)
FEEDBACK	Contrôle l'affichage du nombre de lignes renvoyées par la requête
ECHO	Active ou non l'affichage de la liste des commandes exécutées lors du lancement d'un script

5.4 Un script complet

```
SET ECHO OFF
SET FEEDBACK OFF
SET PAGESIZE 0

SPOOL dropem.sql

SELECT            'DROP TABLE ' || object_name || ';'
FROM              user_objects
WHERE             object_type = 'TABLE';

SPOOL OFF

SET FEEDBACK ON
SET PAGESIZE 24
SET ECHO ON
```

→ Cette requête SQL écrit dans un fichier en générant des ordres pour toutes les tables dont l'utilisateur est le propriétaire. Le résultat envoyé dans le fichier ne contiendra que les ordres DROP grâce à l'utilisation des commandes de contrôle d'environnement.

Pour ensuite exécuter ce script il suffit de lancer la commande START DROPEM.
Par défaut, les fichiers sont spoolé dans le dossier indiqué lors de l'exécution de la commande HOST.

5.5 Renvoyer le contenu d'une table vers un fichier

Bien que les données existent déjà dans la table, il peut être utile d'avoir les valeurs des lignes dans un fichier texte sous la forme d'un ordre INSERT INTO VALUES.
Une des difficultés majeures de ce genre de script est de bien gérer les quotes. Car dans l'ordre INSERT il faut inclure des simples quotes autour de chaque valeur à insérer tout en faisant attention de ne pas les confondre avec les quotes autour du texte à concaténer dans la requête.
Le tableau suivant montre les correspondances entre la syntaxe dans la requête et le résultat dans le fichier :

Source	Résultat
'''X'''	'X'
''''	'
''''||dname||''''	'BOSTON'
''','''	','
'');'	');
'l''apostrophe'	l'apostrophe

Exemple :

```
SET HEADING OFF ECHO OFF FEEDBACK OFF
SET PAGESIZE 0
SPOOL          c:\data.sql
SELECT
               'INSERT INTO dept VALUES
               (' || deptno || ', ''' || dname || ''', ''' || loc ||
                ''');'
               AS "Insert Statements Script"
FROM           dept
/
SPOOL OFF
SET PAGESIZE 24
SET HEADING ON ECHO ON FEEDBACK ON

INSERT INTO dept VALUES (10, 'ACCOUNTING', 'NEW YORK');
INSERT INTO dept VALUES (20, 'RESEARCH', 'DALLAS');
INSERT INTO dept VALUES (30, 'SALES', 'CHICAGO');
INSERT INTO dept VALUES (40, 'OPERATIONS', 'BOSTON');
```

-> Ce script génère une ligne INSERT avec les valeurs contenues dans la table DEPT

5.6 Générer un attribut dynamique

En SQL il est possible de générer des scripts avec un attribut dynamique. Un attribut dynamique est un paramètre non définit dans la requête et dont la valeur est demandée à l'utilisateur. Cette valeur est ensuite utilisée pour construire la requête résultante de la première requête.

Exemple :

```
SQL> COLUMN my_col NEW_VALUE dyn_where_clause

SQL> SELECT       DECODE('&&deptno', null,
  1               DECODE ('&&hiredate', null, ' ',
  2                     'WHERE hiredate=TO_DATE('''||'&&hiredate'',
  3                     ''DD/MM/RR'')'),
  4                DECODE ('&&hiredate', null,
  5                     'WHERE deptno = ' || '&&deptno',
  6                     'WHERE deptno = ' || '&&deptno' ||
  7                     ' AND hiredate = TO_DATE ('''||'&&hiredate'',
  8                     ''DD/MM/RR'')')
  9               ) AS my_col
  10 FROM         dual;

SQL> SELECT ename FROM emp &dyn_where_clause;
ancien   1 : select ename from emp &dyn_where_clause
nouveau    1 : select ename from emp WHERE deptno = 20 AND hiredate = TO_DATE
('12/01/83',  'DD/MM/RR')

ENAME
----------
ADAMS
```

→ Cette requête génère un ordre SELECT qui renvoie les données des employés d'un département et qui on été embauchés un certain jour. La clause WHERE de cet ordre est générée automatiquement selon les données fournies par l'utilisateur.

6 CREATION RAPPORTS AVEC SQL*PLUS

6.1 La commande SET

6.1.1 Les variables de la commande SET

La commande SET est une variable système qui affecte la façon dont SQL*Plus exécute les commandes. Elle permet de contrôler l'affichage des résultats d'une requête SQL.
Les variables de la commande SET sont utilisées pour :

- Contrôler le nombre de lignes vides entre les résultats
- Contrôler le nombre d'espaces entres les colonnes
- Créer des rapports avec des noms évocateurs, des groupes et des sous totaux
- Etablir une valeur pour remplacer les valeurs NULL
- Afficher des titres de colonnes spécifiques

La commande SHOW ALL permet de voir les paramètres actuels de toutes les variables. SHOW suivi du nom du paramètre pour n'afficher que ce paramètre.

6.1.2 Variables de la commande SET supplémentaires

Valeur de SET	Description
RECSEP {WR[APPED]|EA[CH]\OFF}	Contrôle l'affichage des séparateurs des enregistrement (WRAPPED affiche un séparateur uniquement après une ligne bouclée, EACH en affiche un après chaque ligne)
RECSEPCHAR {_|*c*}	Définit le caractère affiché entre les enregistrements
SPA[CE] {1|*n*}	Définit le nombre d'espace entre les colonnes
UND[ERLINE] {-|C|ON|OFF}	Définit le caractère à utiliser pour souligner les en-têtes
WRA[P] {ON|OFF}	Contrôle la troncature des données affichées
NULL *texte*	Définit le texte affiché lorsque la requête retourne une valeur NULL
HEADSEP {-|C|ON|OFF}	Spécifie le caractère à utiliser entre les en-têtes de colonnes
NEWP[AGE] {1|*n*}	Définit le nombre de lignes vides avant le haut de chaque page (0 = ajusté à la page)

6.2 La commande COLUMN

La commande COLUMN permet de contrôler l'affichage des colonnes et des en-têtes.

```
COL[UMN] [{ column| alias} [ option]]
```

Options de la commande COLUMN :

Option	Description
NEW_V[ALUE] *variable*	Définit une variable contenant la valeur d'une colonne et qui peut être utilisée dans la commande TTITLE
NOPRI[NT] PRI[NT]	Contrôle si une colonne est affichée ou non
CLE[AR] DEF[AULT]	Assigne aux paramètres d'affichage des colonnes les valeurs par défauts

6.3 La commande COMPUTE

6.3.1 Syntaxe de COMPUTE

La commande COMPUTE calcul et affiche des lignes de totaux.

```
COMP[UTE]         [function [LABEL labelname] …
                OF { expr| column| alias} …
                ON { expr| column| alias| REPORT| ROW}]
```

COMPUTE peut utiliser plusieurs fonctions de groupe :

Fonction	Opération	Type de données
AVG	Moyenne des valeurs non NULL	NUMBER
COUNT	Nombre de valeurs non NULL	Tous types
MAX[IMUM]	Valeur maximum	NUMBER, CHAR, VARCHAR2
MIN[IMUM]	Valeur minimum	NUMBER, CHAR, VARCHAR2
NUM[BER]	Nombre de lignes	Tous types
STD	Déviation standard des valeurs non NULL	NUMBER
SUM	Somme des valeurs non NULL	NUMBER
VAR[IANCE]	Variance des valeurs non NULL	NUMBER

Pour effacer tous les paramètres de COMPUTE sur les colonnes, on peut utiliser la commande CLEAR COMPUTE.

6.3.2 Utilisation de la commande COMPUTE

Pour pouvoir obtenir le résultat souhaité avec la commande COMPUTE il faut utilisé la commande BREAK (cf. Module 2 : Récupération de données, § 4.3.4 « La commande BREAK ») sur la ou les colonnes concernées par le COMPUTE.

Exemple :

```
SQL> BREAK ON deptno SKIP 1 ON job
SQL> COMPUTE SUM OF sal ON deptno job
SQL> SELECT       deptno, job, ename, sal
  2  FROM         emp
  3  WHERE        job IN ('ANALYST', 'SALESMAN')
  4  ORDER BY     deptno, job, sal;

    DEPTNO JOB       ENAME             SAL
---------- --------- ---------- ----------
        20 ANALYST   SCOTT            3000
                     FORD             3000
           *********              ----------
           sum                        6000
********** *                      ----------
sum                                   6000

        30 SALESMAN  WARD             1250
                     MARTIN           1250
                     TURNER           1500
                     ALLEN            1600

    DEPTNO JOB       ENAME             SAL
---------- --------- ---------- ----------
           *********              ----------
           sum                        5600
********** *                      ----------
sum                                   5600


6 ligne(s) sélectionnée(s).
```

→ Cette requête utilise la commande COMPUTE pour faire calculer le total des salaires pour chaque job listés dans la clause WHERE et pour chaque départements.

Commentaire : Dans ton exemple, c'est le maximum et non pas la somme que tu calcules. De plus je ne comprends pas l'intérêt d'un COMPUTE sur deptno et job, vu que le résultat est le même. Dans le cas d'un exemple d'illustration ce n'est pas très parlant.
22-08 : MAX remplacé par SUM, c'est peut être un peu plus parlant mais je ne vois pas trop comment faire un exemple plus parlant. Sinon il faut juste faire le COMPUTE sur une seule colonne, mais ici c'est juste pour montrer que l'on peut le faire sur plusieurs colonnes.

Pour effectuer un COMPUTE sur l'ensemble du rapport il faut utiliser l'option SKIP REPORT dans la commande BREAK et COMPUTE…ON REPORT dans la commande COMPUTE. Ainsi le résultat pour tout le rapport sera affiché en fin de page.
Il est possible de renommer la ligne contenant le COMPUTE en utilisant LABEL suivi du nom après la fonction de groupe du COMPUTE. Ainsi dans le rapport la ligne issue du COMPUTE sera renommée en utilisant le paramètre du LABEL.

7 ANNEXE : LES TABLES UTILISEES

7.1 La table EMP

```
SQL> set linesize 100
SQL> DESC emp;
 Nom                           NULL ?   Type
 ----------------------------- -------- -------------
 EMPNO                         NOT NULL NUMBER(4)
 ENAME                                  VARCHAR2(10)
 JOB                                    VARCHAR2(9)
 MGR                                    NUMBER(4)
 HIREDATE                               DATE
 SAL                                    NUMBER(7,2)
 COMM                                   NUMBER(7,2)
 DEPTNO                                 NUMBER(2)
 DNAME                                  VARCHAR2(14)

SQL> SELECT *
  2  FROM emp;

EMPNO ENAME      JOB              MGR HIREDATE       SAL     COMM   DEPTNO
------ --------- --------- --------- -------- --------- ------- --------
  7369 SMITH     CLERK          7902 17/12/80       800               20
  7499 ALLEN     SALESMAN       7698 20/02/81      1600     300        30
  7521 WARD      SALESMAN       7698 22/02/81      1250     500        30
  7566 JONES     MANAGER        7839 02/04/81      2975                20
  7654 MARTIN    SALESMAN       7698 28/09/81      1250    1400        30
  7698 BLAKE     MANAGER        7839 01/05/81      2850                30
  7782 CLARK     MANAGER        7839 09/06/81      2450                10
  7788 SCOTT     ANALYST        7566 09/12/82      3000                20
  7839 KING      PRESIDENT           17/11/81      5000                10
  7844 TURNER    SALESMAN       7698 08/09/81      1500        0       30
  7876 ADAMS     CLERK          7788 12/01/83      1100                20

 EMPNO ENAME     JOB              MGR HIREDATE       SAL     COMM   DEPTNO
------ --------- ------- ---------- -------- --------- -------- --------
  7900 JAMES     CLERK          7698 03/12/81       950                30
  7902 FORD      ANALYST        7566 03/12/81      3000                20
  7934 MILLER    CLERK          7782 23/01/82      1300                10
```

7.2 La table DEPT

```
SQL> DESC dept;
 Nom                    NULL ?   Type
 ---------------------- -------- -------------
 DEPTNO                          NUMBER(2)
 DNAME                           VARCHAR2(14)
 LOC                             VARCHAR2(13)

SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

7.3 La table EMP_HISTORY

```
SQL> DESC emp_history;
 Nom                                       NULL ?   Type
 ----------------------------------------- -------- -----------------
 EMPID                                     NOT NULL NUMBER(4)
 NAME                                               VARCHAR2(10)
 TITLE                                              VARCHAR2(9)
 DATE_OUT                                  NOT NULL DATE
 DEPTID                                    NOT NULL NUMBER(2)

SQL> SELECT *
  2  FROM emp_history;

     EMPID NAME       TITLE     DATE_OUT     DEPTID
---------- ---------- --------- -------- ----------
      6087 SPENCER    OPERATOR  27/11/81         20
      6185 VANDYKE    MANAGER   17/01/81         10
      6235 BALFORD    CLERK     22/02/80         20
      7788 SCOTT      ANALYST   05/05/81         20
      7001 JEWELL     ANALYST   10/06/81         30
      7499 ALLEN      SALESMAN  01/08/80         20
      7225 BRIGGS     PAY CLERK 27/11/81         10
      7782 CLARK      MANAGER   12/02/80         10
      7356 WILD       DIRECTOR  01/11/81         10

9 ligne(s) sélectionnée(s).
```