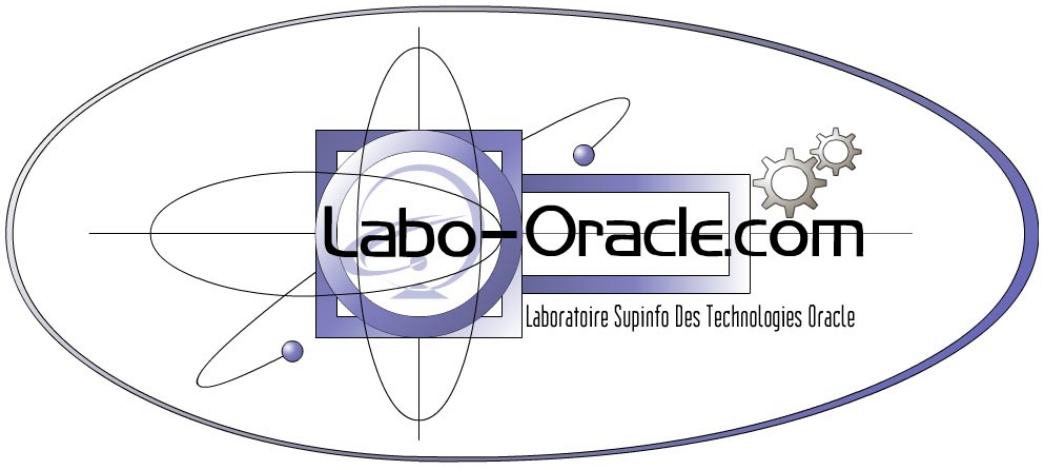


Module n°4

SQL AVANCE

1Z0-007

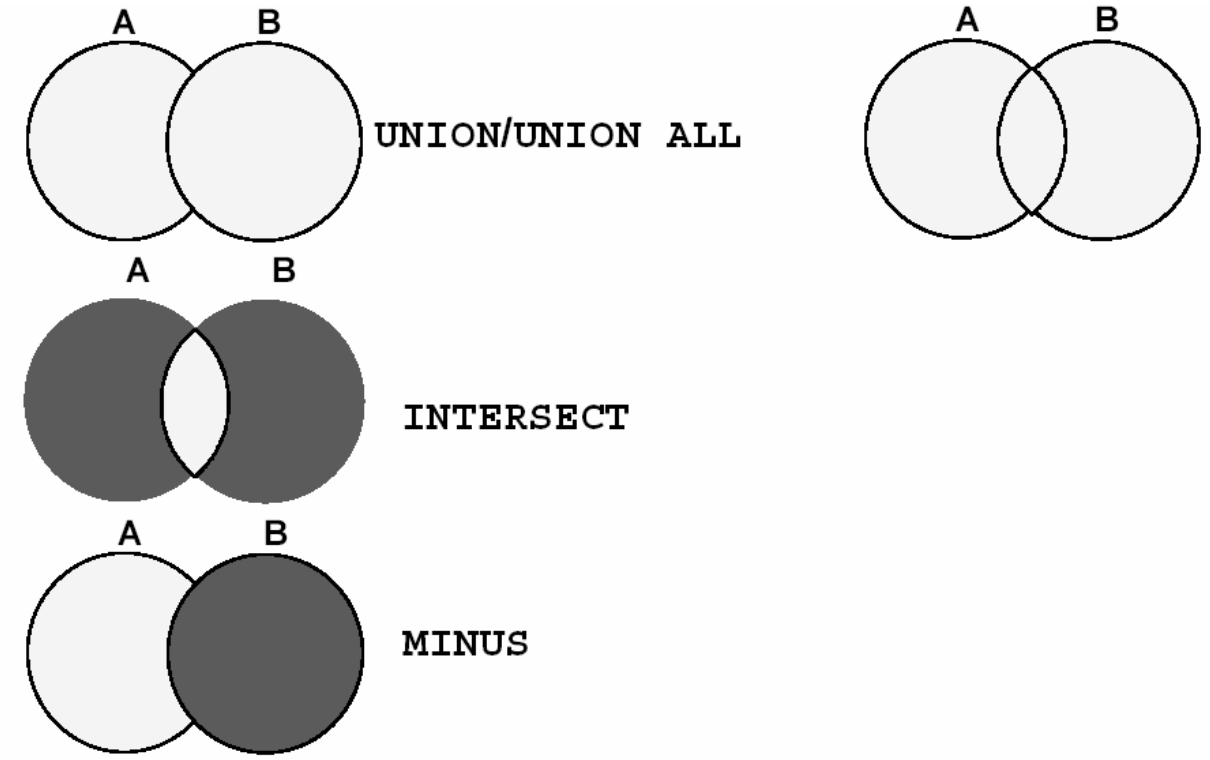
Auteur : Andrei Langéac
Version 1.0 – 12 octobre 2004
Nombre de pages : 48

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com

Sommaire

1. Utilisation des opérateurs d'ensemble

1.1. 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.*

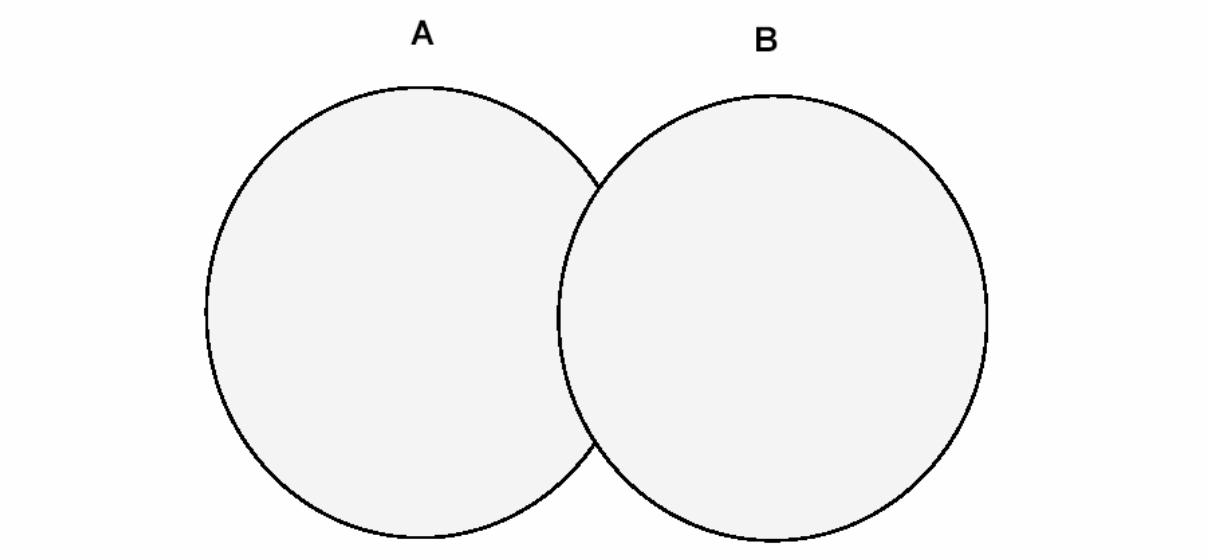
Opérateur	Résultat
INTERSECT	Sélectionne toutes les lignes similaires retournées par les requêtes (**INTERSECT** combine deux requêtes et retourne uniquement le valeurs du premier **SELECT** qui sont identiques à au moins une de celles du second **SELECT**)
UNION	Renvoie toutes les lignes sélectionnées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

1.2.UNION et UNION ALL

1.2.1. 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**.

1.2.2. Utilisation de l'opérateur UNION

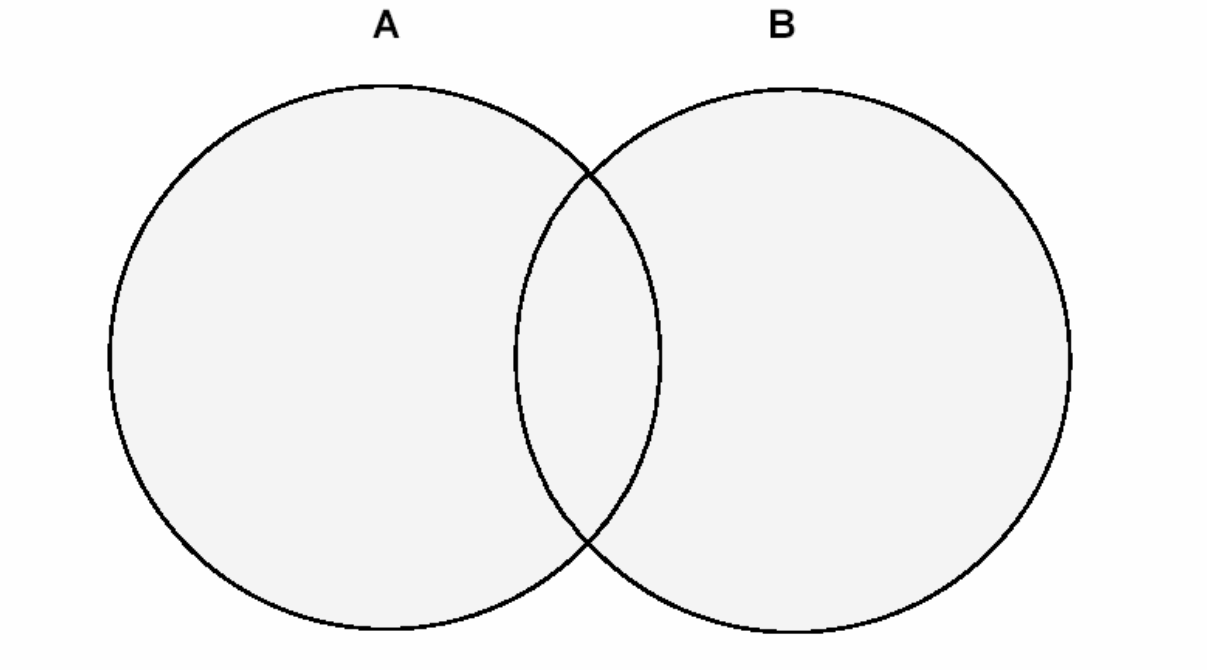
L'opérateur **UNION** élimine les doublons.

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
…
23 lignes sélectionnées
```

Explication: 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.

1.2.3. 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.

1.2.4. L'utilisation de l'opérateur UNION ALL

L'opérateur **UNION ALL** retourne toutes les lignes d'une requête multiple sans éliminer les doublons.

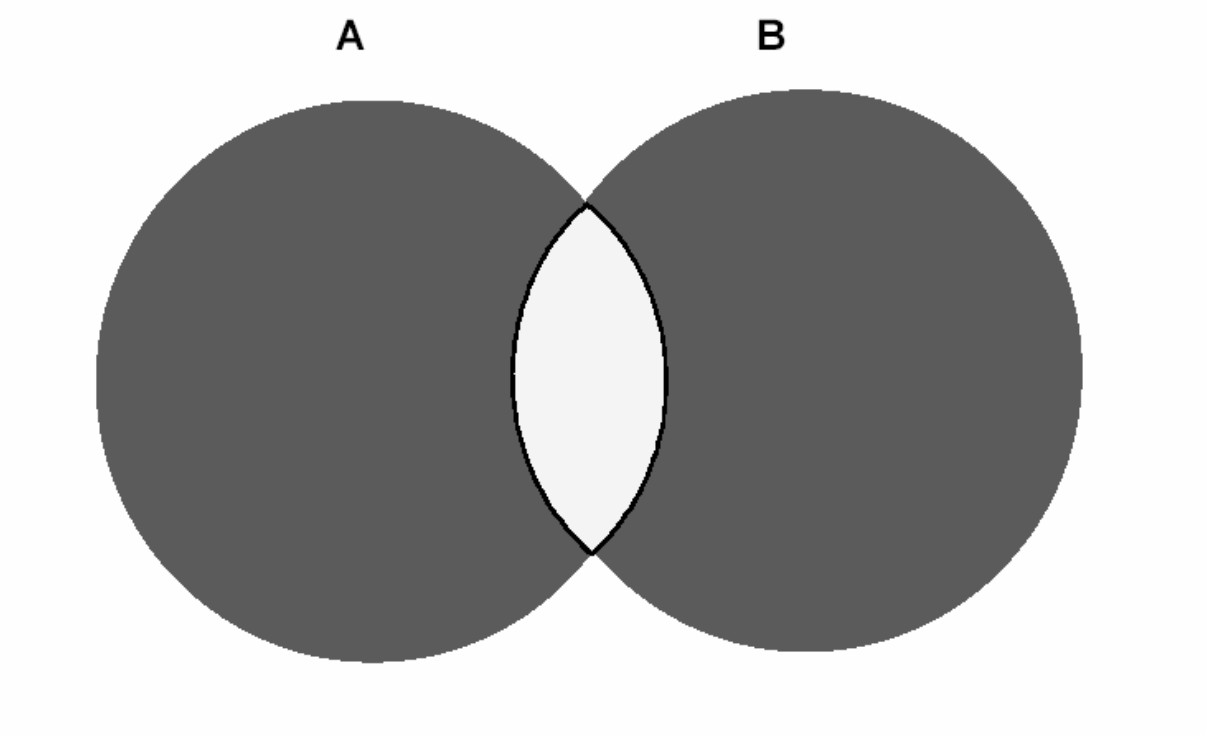
Exemple:

```
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 lignes sélectionnées.
```

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

1.3. INTERSECT

1.3.1. 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**.

1.3.2. Utilisation de l'opérateur INTERSECT

Exemple:

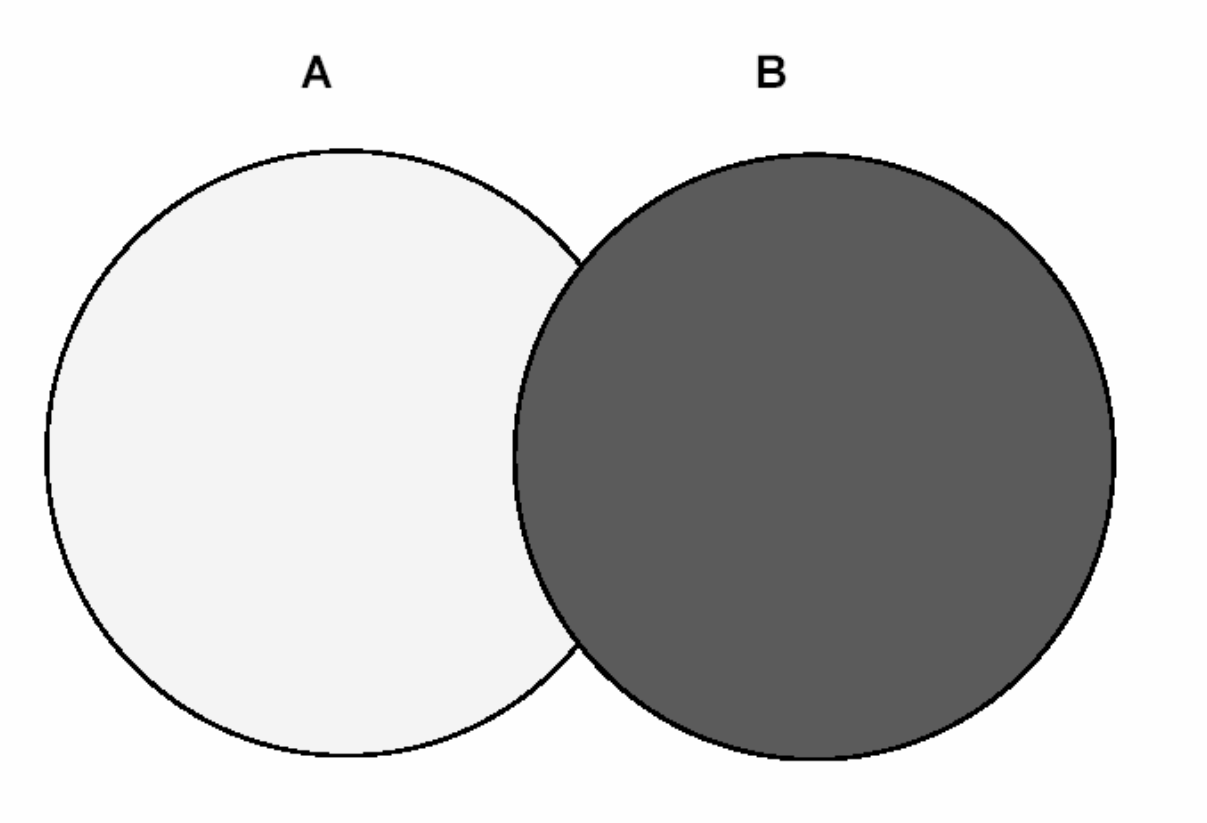
```
SQL>  SELECT      ename, empno, job
  2   FROM        emp
  3   INTERSECT
  4   SELECT      name, empid, title
  5   FROM        emp_history;

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

Explication : 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.

1.4.MINUS

1.4.1. 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**.

1.4.2. Utilisation de l'opérateur MINUS

Exemple:

```
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 lignes sélectionnées.
```

Explication: 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*)

1.5.Règles syntaxiques des opérateurs d'ensemble

1.5.1. 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**.

1.5.2. 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
  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 lignes sélectionnées.
```

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

1.6. Le serveur Oracle et les opérateurs d'ensemble

Lorsque vous utilisez les opérateurs d'ensemble le serveur Oracle élimine automatiquement les doublons sauf lorsque l'opérateur **UNION ALL** est utilisé.
Par défaut le résultat est retourné dans un ordre ascendant par rapport à la première colonne contenue dans **SELECT**.

1.7.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       'sing' AS "My dream", 3 a_dummy
  2         FROM         dual
  3         UNION
  4         SELECT       'I'd like to teach', 1
  5         FROM         dual
  6         UNION
  7         SELECT       'the world to', 2
  8         FROM         dual
  9         ORDER BY     2;

My dream
--------------------
I'd like to teach
the world to
sing
```

Explication: 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.

2. LE SERVEUR ORACLE ET LES FONCTIONS DE DATE

2.1. Les fonctions de date

2.1.1. TZ_OFFSET

La fonction **TZ_OFFSET** retourne le décalage de fuseau horaire correspondant à la valeur écrite. La valeur retournée par cette fonction dépend de la date courante lorsque la requête est exécutée.

Par exemple si la fonction retourne -08:00, la valeur peut être interprétée comme le fuseau horaire d'où la commande a été exécutée, ce qui correspond à huit heures après UTC.

Il est possible d'entrer le nom d'un fuseau horaire ou les mots clés SESSIONTIMEZONE et DBTIMEZONE.

TZ_OFFSET *(['time_zone_name'] '[+ | -] hh:mm'] [**SESSIONTIMEZONE] [DBTIMEZONE])*;

Exemple:

- Le fuseau horaire 'US/Eastern' est quatre heures avant UTC.

```
SQL>        SELECT TZ_OFFSET('US/Eastern')
            FROM DUAL;

TZ_OFFS
-------
-04:00
```

- Le fuseau horaire 'Canada/Yukon' est sept heures avant UTC.

```
SQL>        SELECT TZ_OFFSET('Canada/Yukon')
            FROM DUAL;

TZ_OFFS
-------
-07:00
```

Vous pouvez interroger la vue V$TIMEZONE_NAMES pour voir la liste des fuseaux existants.

```
SQL> DESCRIBE v$timezone_names;

 Nom                                       NULL ?   Type
 ----------------------------------------- -------- ----------------

 TZNAME                                             VARCHAR2(64)
 TZABBREV                                           VARCHAR2(64)
```

2.1.2. CURRENT_DATE

La fonction **CURRENT_DATE** retourne la date actuelle dans le fuseau horaire de la session.
Noter que la valeur de la date est la date dans le calendrier grégorien

Exemple :

```
SQL>       ALTER SESSION
  2        SET NLS_DATE_FORMAT = 'DD-MON-YYY HH24:MI:SS';

Session modifiée.

SQL>       ALTER SESSION
  2        SET TIME_ZONE ='-5:0';

Session modifiée.

SQL>       SELECT SESSIONTIMEZONE, CURRENT_DATE
  2        FROM DUAL;

SESSIONTIMEZONE         CURRENT_DATE
------------------      ---------------------
-05:00                  31-AOU-004 04:29:20


SQL>       ALTER SESSION SET TIME_ZONE = '-8:0';

Session modifiée.

SQL>       SELECT SESSIONTIMEZONE, CURRENT_DATE
  2        FROM DUAL;

SESSIONTIMEZONE         CURRENT_DATE
--------------------    ---------------------
-08:00                  31-AOU-004 01:29:52
```

Explication : Cet exemple montre que la fonction **CURRENT_DATE** est sensible au fuseau horaire de la session. Dans la première requête, la session est modifiée et la valeur **TIME_ZONE** est mise à -5:0
TIME_ZONE est un paramètre de session et non le paramètre d'initialisation qui se configure grâce à la commande :

TIME_ZONE = '*[+ | -] **hh:mm';***

'*[+ | -] **hh:mm';*** indique les heures et les minutes après ou avant UTC

La valeur **CURRENT_DATE** change lorsque le paramètre **TIME_ZONE** est changé

2.1.3. CURRENT_TIMESTAMP

Cette fonction retourne la date et l'heure de la session comme une valeur de type **TIMESTAMP WITH TIME ZONE**

CURRENT_TIMESTAMP (*precision);*

Precision: L'argument qui spécifie les secondes fractionnelles. Par défaut cette valeur est à 6.

Exemple :

```
SQL>       ALTER SESSION
  2        SET TIME_ZONE='-5:0';

Session modifiée.

SQL>       SELECT SESSIONTIMEZONE,CURRENT_TIMESTAMP
  2        FROM DUAL;

SESSIONTIMEZONE         CURRENT_TIMESTAMP
--------------------    -------------------------------
-05:00                  31/08/04 04:39:23,817000 -05:00

SQL>       ALTER SESSION
  2        SET TIME_ZONE='-8:0';

Session modifiée.

SQL>       SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP
  2        FROM DUAL;

SESSIONTIMEZONE         CURRENT_TIMESTAMP
------------------      -------------------------------
-08:00                  31/08/04 01:40:28,319000 -08:00
```

Explication : Cet exemple montre que **CURRENT_TIMESTAMP** est sensible au fuseau horaire de la session.

Noter que la valeur de **CURRENT_TIMESTAMP** change lorsque le paramètre **TIME_ZONE** est modifié

2.1.4. LOCALTIMESTAMP

La fonction **LOCALTIMESTAMP** retourne la date et l'heure actuelle da la session comme une valeur de type **TIMESTAMP.**
La différence entre **LOCALTIMESTAMP** et **CURRENT_TIMESTAMP** est que **LOCALTIMESTAMP** retourne la valeur **TIMESTAMP**.

LOCAL_TIMESTAMP (*TIMESTAMP_precision***);**

TIMESTAMP_precision:	L'argument qui spécifie les secondes fractionnelles de la valeur TIMESTAMP

Exemple :

```
SQL>       ALTER SESSION
  2        SET TIME_ZONE='-5:0';

Session modifiée.

SQL>       SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP
  2        FROM DUAL;

CURRENT_TIMESTAMP                    LOCALTIMESTAMP
-----------------------------        -----------------------------
31/08/04 04:47:23,217000 -05:        0031/08/04 04:47:23,217000


SQL>       ALTER SESSION
  2        SET TIME_ZONE='-8:0';

Session modifiée.

SQL>       SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP
  2        FROM DUAL;

CURRENT_TIMESTAMP                    LOCALTIMESTAMP
-----------------------------        -----------------------------
31/08/04 01:48:00,304000 -08:00      31/08/04 01:48:00,304000
```

Explication : Cet exemple vous montre la différence entre **LOCALTIMESTAMP** et **CURRENT_TIMESTAMP.**

Noter que **LOCALTIMESTAMP** n'affiche pas la valeur du fuseau horaire.

2.1.5. DBTIMEZONE et SESSIONTIMEZONE

Le fuseau horaire par défaut est le fuseau horaire du système d'exploitation
Ce fuseau horaire est configuré en spécifiant la clause **SET TIME_ZONE** dans la requête **CREATE DATABASE**.

La fonction **DBTIMEZONE** retourne la valeur du fuseau horaire de la base de données.
La valeur retournée est le décalage du fuseau horaire ou le nom du fuseau horaire qui dépend de la manière dont l'utilisateur a spécifié la valeur du fuseau horaire de la base de donnée dans la requête **CREATE DATABASE** ou la plus récente requête **ALTER DATABSE**.

Exemple :

```
SQL> SELECT DBTIMEZONE FROM DUAL;

     DBTIME
     ------
     +00:00
```

La fonction **SESSIONTIMEZONE** retourne la valeur du fuseau horaire de la session courante.

La valeur retournée est le décalage du fuseau horaire ou le nom du fuseau horaire qui dépend de la manière dont l'utilisateur a spécifié la valeur du fuseau horaire de la base de donnée dans la plus récente requête **CREATE DATABASE** ou **ALTER DATABSE**.

Notez que le fuseau horaire de la base de données est différent du fuseau horaire de la session.

Exemple :

```
SQL> SELECT SESSIONTIMEZONE
  2  FROM DUAL;

SESSIONTIMEZONE
---------------
-08:00
```

2.1.6. EXTRACT

L'expression **EXTRACT** extrait et retourne la valeur de date depuis une expression

SELECT EXTRACT (*[YEAR] [MONTH] [DAY] [HOUR] [MINUTE]*

[SECOND] [TIMEZONE_HOUR]
[TIMEZONE_MINUTE] [TIMEZONE_REGION]
[TIMEZONE_ABBR]
FROM *[datetime_value_expression]*
*[interval_value_expression]***);**

Lorsque vous extrayez **TIMEZONE_REGION** ou **TIMEZONE_ABBR**, la valeur retournée est une chaîne de caractères qui contient le nom du fuseau horaire ou son abréviation

Lorsque vous extrayez d'autres valeurs, les valeurs retournées seront au format UTC

Exemple :

```
SQL> SELECT EXTRACT (YEAR FROM SYSDATE) FROM DUAL;

EXTRACT(YEARFROMSYSDATE)
------------------------
                    2004
```

Exemple :

```
SQL> SELECT     ename, hiredate,
  2             EXTRACT(MONTH FROM hiredate)
  3  FROM       emp;

ENAME      HIREDATE            EXTRACT(MONTHFROMHIREDATE)
---------- ------------------- --------------------------
SMITH      17-DEC-980 00:00:00                         12
ALLEN      20-FEV-981 00:00:00                          2
WARD       22-FEV-981 00:00:00                          2
JONES      02-AVR-981 00:00:00                          4
MARTIN     28-SEP-981 00:00:00                          9
BLAKE      01-MAI-981 00:00:00                          5
CLARK      09-JUN-981 00:00:00                          6
SCOTT      09-DEC-982 00:00:00                         12
KING       17-NOV-981 00:00:00                         11
TURNER     08-SEP-981 00:00:00                          9
ADAMS      12-JAN-983 00:00:00                          1
```

```
ENAME      HIREDATE            EXTRACT(MONTHFROMHIREDATE)
---------- ------------------- --------------------------
JAMES      03-DEC-981 00:00:00                         12
FORD       03-DEC-981 00:00:00                         12
MILLER     23-JAN-982 00:00:00                          1
GEROGES    26-AOU-004 11:24:58                          8

15 lignes sélectionnées.
```

Explication : La fonction **EXTRACT** est utilisée pour extraire le mois de la colonne *HIREDATE* de la table EMP pour tous les employées.

2.2.Conversions

2.2.1. Conversion TIMESTAMP en utilisant FROM_TZ

La fonction **FROM_TZ** est utilisée pour convertir la valeur de **TIMESTAMP** en **TIMESTAMP WITH TIME ZONE**.

FROM_TZ (TIMESTAMP *timestamp_value, time_zone_value);*

Time_zone_value: Est une chaîne de caractères au format 'TZH: TZM' ou une expression qui retourne une chaîne en **TRZ (time zone region)**.

Exemple:

```
SQL> SELECT FROM_TZ(TIMESTAMP
  2                     '2000-03-28 08:00:00',
  3                     'Australia/North')
  4  FROM   DUAL;


FROM_TZ(TIMESTAMP'2000-03-2808:00:00','AUSTRALIA/NORTH')
---------------------------------------------------------------------
28/03/00 08:00:00,000000000 AUSTRALIA/NORTH
```

2.2.2. STRING en TIMESTAMP en utilisant TO_TIMESTAMP et TO_TIMESTAMP_TZ

La fonction **TO_TIMESTAMP** convertit une chaîne de caractères de type CHAR, VARCHAR, NCHAR, ou NVARCHAR2 en type de données **TIMESTAMP**

TO_TIMESTAMP (*CHAR, [fmt], ['nlsparam']***);**

Fmt: Spécifie le format de CHAR. Si vous ne le spécifiez pas la chaîne sera au format par défaut

Nlsparam: Spécifie la langue dans laquelle les mois, les jours et les abréviations seront retournés.

Voici la syntaxe: 'NLS_DATE_LANGUAGE= language'.

La fonction utilisera la langue par défaut de votre session si ce paramètre n'est pas spécifié

Exemple :

```
SQL> SELECT     TO_TIMESTAMP('2000-12-01 11:00:00',
  2                          'YYYY-MM-DD HH:MI:SS') AS tmp_conversion
  3  FROM       DUAL;

TMP_CONVERSION
---------------------------------------------------------------------
01/12/00 11:00:00,000000000
```

La fonction **TO_TIMESTAMP_TZ** convertit une chaîne de caractères de type CHAR, VARCHAR, NCHAR, ou NVARCHAR2 en type de données **TIMESTAMP WITH TIME ZONE**

TO_TIMESTAMP_TZ (*CHAR, [fmt], ['nlsparam']***);**

Fmt: Spécifie le format de CHAR. Si vous ne le spécifiez pas la chaîne sera au format par défaut .

Nlsparam: Spécifie la langue dans laquelle les mois, les jours et les abréviations seront retournés.

Voici la syntaxe: 'NLS_DATE_LANGUAGE= language'.
La fonction utilisera la langue par défaut de votre session si ce paramètre n'est pas spécifié

Exemple :

```
SQL> SELECT TO_TIMESTAMP_TZ('1999-12-01 11:00:00 -8:00',
  2             'YYYY-MM-DD HH:MI:SS TZH:TZM') AS tz_conversion
  3  FROM DUAL;

TZ_CONVERSION
---------------------------------------------------------------------------
01/12/99 11:00:00,000000000 -08:00
```

2.2.3. Conversion de l'intervalle de temps avec TO_YMINTERVAL

La fonction **TO_YMINTERVAL** convertie la chaîne de caractères de type CHAR, VARCHAR, NCHAR, ou NVARCHAR2 en type de données **INTERVAL YEAR TO MONTH.**
Ce type de données stock une période de temps en utilisant les champs de date **YEAR** et **MONTH**

TO_YMINTERVAL (*char***);**

Char: Chaîne de caractères à convertir.

Exemple:

```
SQL>       SELECT       hiredate,
  2                     hiredate + TO_YMINTERVAL('01-02') AS
  3                     hire_date_YMIN
  4        FROM         emp
  5        WHERE        deptno=20;

HIREDATE                HIRE_DATE_YMIN
------------------      ------------------
17-DEC-1980 00:00:00    17-FEV-1982 00:00:00
02-AVR-1981 00:00:00    02-JUN-1982 00:00:00
09-DEC-1982 00:00:00    09-FEV-1984 00:00:00
12-JAN-1983 00:00:00    12-MAR-1984 00:00:00
03-DEC-1981 00:00:00    03-FEV-1983 00:00:00
```

Explication : Cette requête calcule la date qui se situe un an et deux mois après la date d'embauche pour les employées du département 20.

3. GROUP BY avec les opérateurs ROLLUP et CUBE

3.1. Rappels sur GROUP BY et HAVING

3.1.1. Rappel sur les fonctions de groupe

Voir le cours SQLP Module 2 **« Techniques de récupération des données »**

3.1.2. Rappel sur GROUP BY

Voir le cours SQLP Module 2 **« Techniques de récupération des données »**

3.1.3. Rappel sur HAVING

Voir le cours SQLP Module 2 **« Techniques de récupération des données »**

3.2. ROLLUP

3.2.1. 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.

3.2.2. Exemple

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 à 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 appelés 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 lignes sélectionnées.
```

Explication: 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.

3.3.CUBE

3.3.1. 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**.

3.3.2. Exemple

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 des 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
```

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

3.4. Les fonctions analytiques

3.4.1. 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 composer d'une partition contenant toutes les lignes, quelques grandes partitions, ou de nombreuses petites partitions contenant chacune peu de lignes. Les fonctions analytiques s'appliquent à chaque ligne dans chaque partition.

3.4.2. 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) RANK_OF_JOB_PER_DEP RANK_OF_SUMSAL
---------- --------- ---------- -------------------- --------------
        10 ANALYST          5000                    1              3
        10 MANAGER          2450                    2              6
        10 CLERK            1300                    3              8
        20 ANALYST          6000                    1              1
        20 MANAGER          2975                    2              4
        20 CLERK            1900                    3              7
        30 SALESMAN         5600                    1              2
        30 MANAGER          2850                    2              5
        30 CLERK             950                    3              9

9 lignes sélectionnées.
```

Explication: 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.

3.4.3. 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 lignes sélectionnées.
```

Explication: Cette requête affiche le **CUME_DIST** de la somme des salaires pour chaque job dans chaque département.

3.5.GROUPING

3.5.1. 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**.

3.5.2. Exemple

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 lignes sélectionnées.
```

Explication: 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

3.5.3. La fonction GROUPING SETS

Vous pouvez utilisez la fonction **GROUPING SETS** pour définir des groupements multiples dans la même requête
GROUPING SETS est une extension de la clause **GROUP BY** qui vous permet de spécifier plusieurs groupements.
Vous pouvez écrire une requête **SELECT** avec **GROUPING SETS** plutôt que plusieurs requêtes **SELECT** reliées avec l'opérateur **UNION ALL**.

3.5.4. Utilisation de GROUPING SETS

```
SQL>        SELECT     deptno,job,mgr,AVG(sal)
  2         FROM       emp
  3         GROUP BY   GROUPING SETS
  4                      ((deptno,job), (job,mgr));

    DEPTNO JOB              MGR   AVG(SAL)
---------- --------- ---------- ----------
        10 CLERK                      1300
        10 MANAGER                    2450
        10 PRESIDENT                  5000
        20 ANALYST                    3000
        20 CLERK                       950
        20 MANAGER                    2975
        30 CLERK                       950
        30 MANAGER                    2850
        30 SALESMAN                   1400
        60
           ANALYST         7566       3000

    DEPTNO JOB               MGR   AVG(SAL)
---------- --------- ---------- ----------
           CLERK           7698        950
           CLERK           7782       1300
           CLERK           7788       1100
           CLERK           7902        800
           MANAGER         7839 2758,33333
           PRESIDENT                   5000
           SALESMAN        7698       1400


19 lignes sélectionnées.
```

Explication: Cette requête calcule des agrégats de plus de deux groupements
La table est divisée en deux groupes : deptno/job et job/mgr
Le salaire moyen pour chaque groupe est calculé et affiché

3.6.Les colonnes composées

3.6.1. Les colonnes composées

Une colonne composée est une collection de colonnes traitées comme une unité pendant le calcul des groupements. Pour utiliser les colonnes composées vous devez spécifier les colonnes entre les parenthèses.

ROLLUP (column A, (column B, column C))

Ici (column B, column C) sont traités comme une unité et ROLLUP ne sera pas appliqué pour (column B, column C).

Les colonnes composées sont souvent utilisées avec **ROLLUP, CUBE,** et **GROUPING SETS.**

Requêtes GROUPIN SETS	Equivalent en GROUP BY
GROUP BY GROUPING SETS (a, b, c)	**GROUP BY** a **UNION ALL** **GROUP BY** b **UNION ALL** **GROUP BY** c
GROUP BY GROUPING SETS (a, b, (b, c)) (**GROUPIN SET** a une colonne composée)	**GROUP BY** a **UNION ALL** **GROUP BY** b **UNION ALL** **GROUP BY** b, c
GROUP BY GROUPING SETS ((a, b, c))	**GROUP BY** a, b, c
GROUP BY GROUPING SETS (a, (b), ())	**GROUP BY** a **UNION ALL** **GROUP BY** b **UNION ALL** **GROUP BY** ()

3.6.2. Exemple

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

    DEPTNO JOB              MGR   SUM(SAL)
---------- --------- ---------- ----------
        10 CLERK           7782       1300
        10 MANAGER         7839       2450
        10 PRESIDENT                  5000
        10                            8750
        20 ANALYST         7566       6000
        20 CLERK           7788       1100
        20 CLERK           7902        800
        20 MANAGER         7839       2975
        20                           10875
        30 CLERK           7698        950
        30 MANAGER         7839       2850

    DEPTNO JOB              MGR   SUM(SAL)
---------- --------- ---------- ----------
        30 SALESMAN        7698       5600
        30                            9400
        60
        60
                                     29025

16 lignes sélectionnées.
```

Explication: Cette requête calcule les groupements suivants:

- (deptno, job, mgr)
- (deptno)
- ()

Et affiche:

- Le salaire total pour chaque département.
- Le salaire total pour chaque département job et manager
- Total.

3.7.Groupes concaténés

3.7.1. Les groupes concaténés

Les groupes concaténés offrent une manière concise de produire des combinaisons utiles des groupements. Ils sont spécifiés en utilisant plusieurs **GROUPING SETS**, **CUBE** et **ROLLUP** en les séparant par les virgules

GROUP BY GROUPING SETS (a, b), **GROUPING SETS** (c, d)

La requête précédente définie les groupes suivants

(a, c), (a, d), (b, c), (b, d).

3.7.2. Exemple

```
SQL> SELECT     deptno,job,mgr,AVG(sal)
  2  FROM       emp
  3  GROUP BY   deptno,
  4             ROLLUP(job),
  5             CUBE(mgr);

    DEPTNO JOB              MGR   AVG(SAL)
---------- --------- ---------- ----------
        10 CLERK           7782       1300
        10 MANAGER         7839       2450
        10 PRESIDENT                  5000
        20 ANALYST         7566       3000
        20 CLERK           7788       1100
        20 MANAGER         7839       2975
        20 CLERK           7902        800
        30 CLERK           7698        950
        30 SALESMAN        7698       1400
        30 MANAGER         7839       2850
        60
           ...

35 lignes sélectionnées
```

Explication: Cette requête calcule les groupements suivants:

- (deptno, mgr, job)
- (deptno, mgr)
- (deptno, job)
- (deptno).

Le salaire total pour chaque groupe est calculé

Cet exemple affiche :

- Le salaire total pour chaque département manager et job.
- Le salaire total pour chaque département et manager.
- Le salaire total pour chaque département et job.
- Le salaire total pour chaque département.

4. Les sous-requêtes avancées

4.1. Les sous-requêtes

4.1.1. Qu'est-ce qu'une sous-requête ?

Voir le cours SQLP Module 2 **« Techniques de récupération des données »**

4.1.2. Les sous-requêtes multiple colonnes

Voir le cours SQLP Module 2 **« Techniques de récupération des données »**

4.1.3. Utilisation des sous-requêtes

Voir le cours SQLP Module 2 **« Techniques de récupération des données »**

4.2. Les comparaisons entre les colonnes

4.2.1. La comparaison pairwise

La requête SQL ci dessous contient une sous-requête multiple-colonne car la sous-requête retourne plus d'une colonne.
Cela compare tous les valeurs de la colonne MGR et tous les valeurs de la colonne DEPTNO, avec les valeurs des colonnes MGR et DEPTNO pour les employées qui ont les numéros 7369 ou 7499.

Premièrement la sous-requête retrouve les valeurs de MGR et DEPTNO pour les employées 7369 ou 7499. Ensuite ces valeurs sont comparées avec les valeurs des colonnes MGR et DEPTNO de la table EMP.
Si les valeurs correspondent les données seront affichées. Les employées qui possèdent EMPNO 7369 et 7499 de feront pas partie de cet affichage.

Exemple:

```
SQL> SELECT     empno, mgr, deptno
  2  FROM       emp
  3  WHERE      (mgr, deptno) IN
  4                             (SELECT mgr, deptno
  5                             FROM    emp
  6                             WHERE empno IN (7369,7499))
  7  AND        empno NOT IN (7369,7499);


     EMPNO        MGR     DEPTNO
---------- ---------- ----------
      7521       7698         30
      7844       7698         30
      7900       7698         30
      7654       7698         30
```

4.2.2. Les comparaisons nonpairwise

L'exemple ci-dessous montre une comparaison nonpairwise.

Cette requête affiche les colonnes EMPNO, MGR, et DEPTNO pour chaque employé dont le manager ID correspond au manager ID des employés numéro 7639 ou 7499 et dont le DEPTNO correspond au numéro de département des employés 7369 ou 7499
La première sous-requête retrouve les valeurs de MGR pour les employées dont EMPNO est 7360 ou 7499. La seconde sous-requête retrouve les valeurs de DEPTNO pour les même employées.
Les valeurs des colonnes MGR et DEPTNO retrouvées précédemment sont comparées aux valeurs de MGR et DEPTNO de la table EMP.
Si les valeurs de la colonne MGR de la table EMP correspondent aux valeurs retrouvées par la première sous-requête et si les valeurs de la colonne DEPTNO de la table EMP correspondent aux valeurs retrouvées par la seconde sous-requête des données sont affichées.

Exemple:

```
SQL> SELECT     empno, mgr, deptno
  2  FROM       emp
  3  WHERE      mgr IN
  4                     (SELECT mgr
  5                     FROM    emp
  6                     WHERE empno IN (7369,7499))
  7  AND        deptno IN
  8                     (SELECT deptno
  9                     FROM    emp
 10                     WHERE   empno IN (7369,7499))
 11  AND        empno NOT IN (7369,7499);

     EMPNO        MGR     DEPTNO
---------- ---------- ----------
      7521       7698         30
      7844       7698         30
      7900       7698         30
      7654       7698         30
```

4.3.Les sous-requêtes scalaires

4.3.1. Les sous-requêtes scalaires

Les sous-requêtes scalaires retournent une seule colonne d'une seule ligne.
Les sous-requêtes multiple-colonnes qui comparent deux colonnes ou plus en utilisant la clause **WHERE** et les opérateurs logiques ne sont pas considérées comme les sous-requêtes scalaires

La valeur de la sous-requête scalaire est la valeur de la liste de la requête **SELECT**
Si la sous-requête retourne zéro lignes la valeur de la sous-requête scalaire est **NULL**
Si cela retourne plus d'une ligne le serveur Oracle vous enverra une erreur.

Vous pouvez utiliser les sous-requêtes scalaires dans :

- Les conditions et les expressions faisant partie de **DECODE** et **CASE.**
- Toutes les clauses **SELECT** sauf **GROUP BY.**
- A gauche de l'opérateur dans la clause **SET** et dans la clause **WHERE** lors de la mise à jour (**UPDATE**)

Les sous-requêtes scalaires ne sont pas valides dans les cas suivants:

- Comme la valeurs par défaut pour les colonnes et les expression de hash pour les clusters.
- Dans la clause **RETURNING**.
- Comme la base pour les index bases sur la fonction.
- Dans la clause **GROUP BY** la contrainte **CHECK** et la condition **WHEN**.
- Dans la clause **HAVING**.
- Dans les clauses **START WITH** et **CONNECT BY**
- Dans des expressions independantes comme **CREATE PROFILE.**

4.3.2. Exemple

```
SQL> SELECT     empno,ename,
  2             (CASE
  3             WHEN deptno=
  4                     (SELECT deptno FROM dept
  5                     WHERE LOC='DALLAS')
  6             THEN 'Canada' ELSE 'USA' END) location
  7  FROM       emp;

     EMPNO ENAME      LOCATI
---------- ---------- ------
      7369 SMITH      Canada
      7499 ALLEN      USA
      7521 WARD       USA
      7566 JONES      Canada
      7654 MARTIN     USA
      7698 BLAKE      USA
      7782 CLARK      USA
      7788 SCOTT      Canada
      7839 KING       USA
      7844 TURNER     USA
      7876 ADAMS      Canada

     EMPNO ENAME      LOCATI
---------- ---------- ------
      7900 JAMES      USA
      7902 FORD       Canada
      7934 MILLER     USA
      7777 GEROGES    USA

15 lignes sélectionnées.
```

Explication : La sous-requête retourne la valeur 20 qui correspond à l'ID de département DALLAS. La clause **CASE** utilise cette valeur pour afficher l'ID des employées, et la valeur de CANADA ou USA en fonction de l'ID de département retourné par la sous-requête.

4.4.Des sous requêtes corrélées

4.4.1. 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 utiliser (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**.

4.4.2. Exemple

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.

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 lignes sélectionnées.
```

Explication 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.

4.5.Les opérateurs EXITS et NOT EXISTS

4.5.1. 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.

4.5.2. Utilisation de l'opérateur EXISTS

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 lignes sélectionnées
```

Explication: 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.

4.5.3. 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
```

Explication : Cette requête affiche les départements dans lesquels il n'y a aucun employé.

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**.

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).
```

Explication : 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
```

Explication : 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**.

4.6.UPDATE et DELETE corrélés

4.6.1. UPDATE corrélé

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 lignes sélectionnées.
```

4.6.2. DELETE corrélé

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);
```

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

4.7.La clause WITH

4.7.1. La clause WITH

Vous pouvez utiliser la clause **WITH** pour définir le block de requête avant de l'utiliser dans la requête.
Vous pouvez réutiliser la même requête lorsque vous utilisez plusieurs fois cette requête
Le serveur Oracle garde en mémoire le résultat de ce block de requête. Cela permet d'augmenter les performances

4.7.2. Exemple

```
SQL>        WITH
  2                dept_cost  AS(
  3                       SELECT  d.dname, SUM(e.sal) AS dept_total
  4                       FROM    emp e, dept d
  5                       WHERE   e.deptno=d.deptno
  6                       GROUP BY d.dname),
  7                avg_cost   AS(
  8                       SELECT SUM(dept_total)/COUNT(*) AS dept_avg
  9                       FROM    dept_cost)
 10         SELECT     *
 11         FROM       dept_cost
 12         WHERE      dept_total >
 13                                  (SELECT dept_avg
 14                                  FROM    avg_cost)
 15         ORDER BY   dname;

DNAME          DEPT_TOTAL
-------------- ----------
ACCOUNTING           8750
RESEARCH            10875
SALES                9400
```

Explication : L'exemple nous montre comment augmenter les performances de la requête et l'écrire plus simplement en utilisant la clause **WITH**.
La requête crée DEPT_COST et AVG_COST pour les réutiliser plus tard dans la requête principale.

5. Récupération hiérarchique

5.1. Aperçu des requêtes hiérarchiques

5.1.1. 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.

5.1.2. 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

5.1.3. 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.

5.2. Parcourir l'arbre

5.2.1. 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.

5.2.2. 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.

5.2.3. 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
```

Explication: 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
```

Explication: 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..

5.3. Organiser les données

5.3.1. 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.

5.3.2. 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**
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 lignes sélectionnées.
```

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

5.3.3. 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
```

Explication: 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).

5.3.4. 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.

5.3.5. 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 lignes sélectionnées.
```

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

6. Ordres DML et DDL avancés

6.1.La requête INSERT multitables

6.1.1. Types de la requête INSERT multitables

Vous pouvez insérer une ligne dans une table multiple en tant qu'élément d'une requête DML simple en employant **INSERT…SELECT**

Voici les différents types de requêtes INSERT multitables

- **INSERT ALL** inconditionnel**.**
- **INSERT ALL** conditionnel**.**
- **FIRST INSERT** conditionnel**.**
- Pivoting **INSERT.**

INSERT [ALL] [Conditional_insert_clause]
[insert_into_clause Values_clause] (Subquery)

Condtional_insert_clause:

[ALL] [FIRST]
[WHEN condition THEN] [insert_into_clause_values_clause]
[ELSE] [Insert_into_clause Values_clause]

Syntaxe	Description
INSERT:ALL into_clause	Utilise des multiples *insert_into_clauses* pour augmenter les performances de l'insertion inconditionnelle Le serveur Oracle exécute chaque *insert_into_clause* une fois pour chaque enregistrement retourné par la sous requête.
INSERT: conditional_insert_clause	Specifiez *conditional_insert_clause* pour augmenter les performances de l'insertion conditionnelle Le serveur Oracle filtre chaque *insert_into_clause* à travers la condition **WHEN** qui détermine quelle *insert_into_clause* est executée Une simple insertion multitables peut contenir plus de 127 conditions **WHEN**
FIRST: INSERT	Si vous spécifiez **FIRST,** le serveur Oracle évalue chaque clause WHEN dans l'ordre d'apparence Si le premier **WHEN** rencontré est évalué à vrai le serveur Oracle exécute la clause **INTO** correspondante et prend pas en compte les WHEN suivants.

INSERT: ELSE clause	Si aucune clause **WHEN** n'est évaluée à vrai et si vous avez spécifié la clause **ELSE**, le serveur Oracle exécute les clauses **INTO** associées à cette clause **ELSE**, dans le cas où la clause **ELSE** n'est pas spécifiée le Serveur Oracle ne fait aucune action.

6.1.2. INSERT ALL inconditionnel

L'exemple suivant insère des données dans les tables SAL_HISTORY et MGR_HISTORY. Grâce à SELECT on récupère les ID des employées, la date d'embauche, le salaire et l'ID des managers pour les employées dont l'ID est supérieur à 7400 dans la table EMP.

Les détails sur les ID des employées, la date d'embauche et le salaire sont insérés dans la table SAL_HISTORY.

Les détails sur les ID des employées, les ID des managers et le salaire sont insérés dans la table MGR_HISTORY.

La requête **INSERT** est considérée comme un **INSERT** inconditionnel car aucune restriction ne sera appliquée aux enregistrements retournés par **SELECT**. Tous les enregistrements seront insérés dans les tables.
La clause **VALUES** spécifie les colonnes à insérer dans les différentes tables.

```
SQL> INSERT ALL
  2     INTO sal_history        VALUES(empno,hiredate,sal)
  3     INTO mgr_history        VALUES(empno,mgr,sal)
  4     SELECT
  5             empno, hiredate, sal, mgr
  6     FROM    emp
  7     WHERE empno > 7400;

28 lignes crées.
```

6.1.3. INSERT ALL conditionnel

Cet exemple est similaire au précèdent.

La requête **INSERT** est référence comme un **INSERT ALL** conditionnel car des restrictions sont appliquées sur des enregistrements retournés par la requête **SELECT**. Seuls les enregistrements possédant une valeur SAL supérieure à 2000 seront insérés dans la table SAL_HISTORY. Les enregistrements dont la valeur de la colonne MGR est supérieure à 7400 seront insérés dans la table MGR_HISTORY.

```
INSERT ALL
     WHEN sal > 2000 THEN
            INTO sal_history VALUES(empno,hiredate,sal)
     WHEN mgr > 7400 THEN
            INTO mgr_history VALUES(empno,mgr,sal)
            SELECT empno, mgr, hiredate, sal
            FROM   emp
            WHERE  empno > 7400

18 lignes crées.
```

6.1.4. FIRST INSERT conditionnel

L'exemple suivant insère des enregistrements dans plusieurs tables en utilisant une seule requête **INSERT**.
La requête **SELECT** récupère des détails sur les ID des départements, le salaire total, et la date d'embauche la plus récente pour chaque département de la table EMP.

L'ordre **INSERT** est considéré comme **FIRST INSERT** conditionnel. La condition **WHEN ALL > 1000** est évaluée en premier. Si le salaire total pour un département est supérieur à 1000 un enregistrement sera inséré dans la table SPECIAL_SAL
Pour des enregistrements qui ne satisfont pas la première condition **WHEN**, le reste des conditions est évalué. Si un enregistrement ne satisfait aucune condition **WHEN** il sera inséré dans la table *hiredate_history*.

```
SQL> INSERT FIRST
  2             WHEN SAL > 1000         THEN
  3          INTO special_sal        VALUES (deptno,sal)
  4             WHEN hiredate LIKE('%00%') THEN
  5          INTO hiredate_history_00        VALUES(deptno,hiredate)
  6             WHEN hiredate LIKE ('%99%') THEN
  7          INTO hiredate_history_99        VALUES(deptno,hiredate)
  8             ELSE
  9          INTO hiredate_history VALUES(deptno,hiredate)
 10             SELECT  deptno,sum(sal) sal, MAX(hiredate) hiredate
 11             FROM    emp
 12             GROUP BY deptno;

4 lignes crées.
```

6.1.5. Pivoting INSERT

Pivoting est une opération pour laquelle vous avez besoin de créer une transformation tel que une table non relationnelle doit être convertie en plusieurs enregistrements de la table relationnelle

Dans l'exemple suivant les données des ventes sont récupérées depuis la table non relationnelle SALES_SOURCE_DATA laquelle contient les détails des ventes effectués par un représentant chaque jour de la semaine, pour chaque semaine ayant un Id donné.

```
SQL> INSERT ALL
  2     INTO sales_info VALUES(employee_id,week_id,sales_MON)

  3     INTO sales_info VALUES(employee_id,week_id,sales_TUE)

  4     INTO sales_info VALUES(employee_id,week_id,sales_WED)

  5     INTO sales_info VALUES(employee_id,week_id,sales_THUR)

  6     INTO sales_info VALUES(employee_id,week_id,sales_FRI)

  7     SELECT employee_id, week_id, sales_MON, sales_TUE, sales_WED
  8             ,sales_THUR, sales_FRI
  9     FROM    sales_source_data;

5 lignes crées.
```

6.2. Tables externes

6.2.1. Création des tables externes

Les tables externes sont des tables en lecture seule, dont les metadatas sont stockées dans la base de donnée locale mais les données se trouvent dans une base de données distante.

Vous pouvez utiliser les données externes comme une table virtuelle. Ces données peuvent être sélectionnées et jointes directement et en parallèle sans qu'elles ne soient chargées dans la base de données locale.
Il est possible d'utiliser SQL, PL/SQL, et JAVA pour récupérer les données.
Etant donné que ces tables sont en lecture seule aucun ordre DML n'est possible sur ces tables

Les metadatas sont créées en utilisant la requête DDL **CREATE TABLE**

Le serveur Oracle fournit deux principaux drivers pour accéder à la table externe

Le premier est ORACLE_LOADER, il est utilise pour lire les données en utilisant la technologie « Oracle loader ».

Le deuxième est ORACLE_INTERNAL, il peut être utilisé pour importer et exporter les données en utilisant le format indépendant de la plateforme.

Vous pouvez créer une table externe en utilisant la clause **ORGANIZATION EXTERNAL** avec la requête **CREATE TABLE**.
Lorsque vous l'utilisez vous créez une metadata dans le dictionnaire de données que vous utilisez pour accéder aux données externes. Si vous spécifiez le mot clé **EXTERNAL** dans la clause **ORGANIZATION** vous indiquez que la table est en lecture seule et se trouve en dehors de la base locale.

TYPE access_driver_type indique le driver d'accès pour une table externe.

La clause **REJECT LIMIT** permet de spécifier le nombre maximum d'erreur avant que la requête soit annulée.
DEFAULT DIRECTORY Spécifie le répertoire où les données sources peuvent se trouver**.**
La clause **ACCESS PARAMETERS** vous permet d'assigner les paramètres pour le driver d'accès.

La clause LOCATION spécifie le repère externe pour chaque donnée externe.
Habituellement, **location_specifier** est un fichier

6.2.2. Exemple

Vous devez utiliser la requête **CREATE DIRECTORY** pour créer un répertoire d'objet, qui est l'alias pour le répertoire où se trouvent les données.
Noter que vous devez posséder les privilèges **CREATE ANY DIRECTORY** pour pouvoir créer des répertoires.
Une fois que le répertoire est crée, vous recevez automatiquement le droit **READ** et vous pouvez donner ce droit aux autres utilisateurs et rôles.

CREATE [OR REPLACE] DIRECTORY *repertoire* **AS '***path_name';*

Path_name: Spécifie le chemin complet du PATH de votre système d'exploitation.

Supposons que vous ayez un fichier qui possède des enregistrements suivants:

10, Jones, 11-DEC-1934
20, Smith, 12-JUN-1972

Les enregistrements sont délimités par de nouvelles lignes et les champs se terminent par une virgule. Le nom du fichier est : /flat_files/emp1.txt

Pour convertir ce fichier en données pour une table externe, dont les metadatas se trouveront dans la base de données locale vous devez:

1. Créer le répertoire d'objet emp_dir:

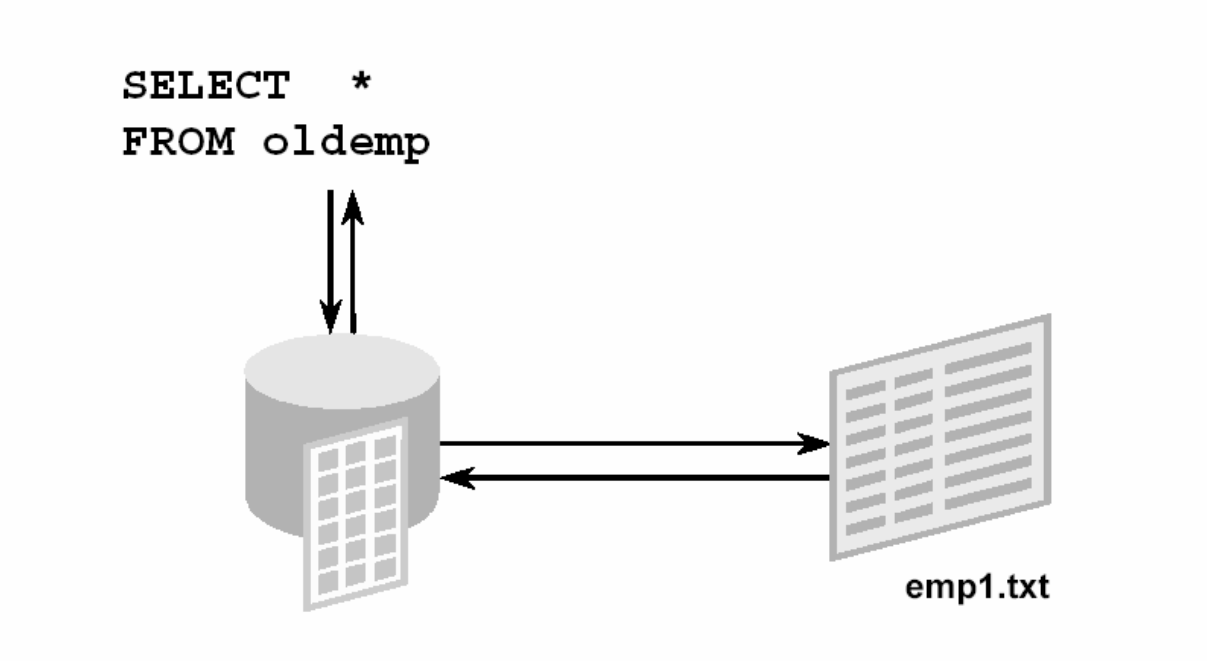
```
CREATE DIRECTORY emp_dir AS '/flat_files';
```

2. Executer la requête suivante.

```
SQL> CREATE TABLE       odlemp (
  2     empno NUMBER, ename CHAR (20), hiredate DATE)
  3     ORGANIZATION EXTERNAL
  4     (TYPE ORACLE_LOADER
  5     DEFAULT DIRECTORY emp_dir
  6     ACCESS PARAMETERS
  7     (RECORDS DELIMITED BY NEWLINE
  8     BADFILE         'bad_emp'
  9     LOGFILE         'log_emp'
 10     FIELDS TERMINATED BY ','
 11     (empno CHAR,
 12     ename  CHAR,
 13     hiredate CHAR date_format date mask "dd-mon-yyyy"))
 14     LOCATION        ('emp1.txt'))
 15     PARALLEL        5
 16     REJECT LIMIT    200;

Table crée.
```

Explication: La clause **PARALLEL** autorise cinq serveurs parallèles d'exécution pour scanner les tables externes lorsque la clause **INSERT INTO TABLE** est exécutée.

6.2.3. Interroger les tables externes



Une table externe ne décrit aucune donnée qui est stockée dans la base de données

Il décrit comment la couche externe de table doit présenter les données au serveur.

Quand la base de données doit accéder à des données dans une source extérieure, elle appelle le driver d'accès approprié pour obtenir les données d'une source extérieure sous une forme que le serveur de base de données s'attend.

Notez que la description des données est séparée de la définition de la table externe.
Les types de données pour les champs dans la source des données peuvent être différents des colonnes dans la table.
Le driver d'accès prend soin d'assurer que les données du point d'émission de données sont traitée de sorte qu'ils correspondent à la définition de la table externe.

6.3.CREATE INDEX avec la requête CREATE TABLE

Il est possible à un utilisateur qui possède de bons privilèges de créer un index lorsqu'il crée une table.

Exemple:

```
SQL> CREATE TABLE new_emp
  2     (empno NUMBER(6)
  3                     PRIMARY KEY USING INDEX
  4                     (CREATE INDEX emp_id_idx ON
  5                     new_emp(empno)),
  6     ename VARCHAR2(50));

Table crée
```

Explication: Dans cet exemple la clause **CREATE INDEX** est utilisée avec la requête **CREATE TABLE** pour créer l'index pour la clé primaire explicitement.
Vous pouvez nommer votre index lors de la création de la clé primaire.

Vous pouvez vérifier que votre index a bien été crée dans le dictionnaire de données USER_INDEX

```
SQL>       SELECT       index_name, table_name
  2        FROM         user_indexes
  3        WHERE        table_name='NEW_EMP';

INDEX_NAME                     TABLE_NAME
------------------------------ ------------------------------
EMP_ID_IDX                     NEW_EMP
```