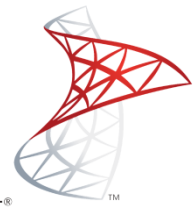




ORACLE



Microsoft®  
SQL Server®



# *Systemes de gestion de bases de données*

## NoSQL DATABASES



## NewSQL DATABASES



## GRAPH DBs



Cours conçu par **Razvan BIZOI**

razvan@bizoi.fr

Reproduction interdite

- *Projection des données*
- *La valeur NULL*
- *Opérateurs arithmétiques et de concaténation*
- *Tri*

# 2

## ***L'interrogation des données***



### **Objectifs**

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

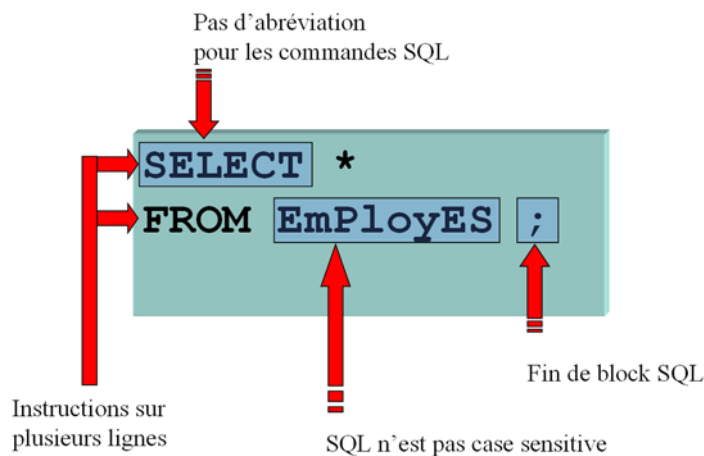
- Extraire d'une table les colonnes souhaitées.
- Traiter les colonnes contenant des valeurs NULL.
- Effectuer des opérations arithmétiques avec les colonnes de type numériques et les colonnes de type date.
- Afficher les résultats des requêtes triées.



### **Contenu**

Grammaire SQL	2-2	La valeur NULL	2-9
Projection	2-2	Tri du résultat d'une requête	2-11
Les constantes	2-5	La pseudo-colonne ROWNUM	2-15
Opérateur de concaténation	2-6	La limitation d'enregistrements	2-16
Opérateurs arithmétiques	2-7		

# Grammaire SQL



Voici quelques exigences de syntaxe à garder à l'esprit lorsque vous travaillez avec **SQL** :

- Chaque instruction **SQL** se termine par un point-virgule.
- Une instruction **SQL** peut être saisie sur une ligne ou, par souci de clarté, répartie sur plusieurs. La plupart des exemples de ce livre comprennent des instructions fractionnées en portions lisibles.
- Vous ne pouvez pas abréger une commande **SQL**.
- **SQL** ne tient pas compte de la casse; vous pouvez combiner les majuscules et les minuscules lorsque vous vous référez aux mots clés **SQL**, tels que « **SELECT** » ou « **INSERT** », aux noms de tables et aux noms de colonnes. Toutefois, l'utilisation des majuscules ou minuscules a son importance lorsque vous vous référez au contenu d'une colonne. Si vous demandez tous les clients dont le nom commence par 'a' et que tous les noms de clients sont stockés en majuscules, la requête n'extrait aucun enregistrement.
- Une seule instruction **SQL** peut être stockée dans le tampon mémoire **SQL\*Plus**. Si deux instructions **SQL** sont exécutées une après l'autre le tampon mémoire contient seulement la dernière et c'est uniquement celle-ci qui peut être éditée.

## Projection

L'opération de projection permet de retenir certaines ou toutes les colonnes d'une table et retourne l'intégralité des enregistrements de la table.

Une projection s'exprime à l'aide du langage **SQL** par la clause « **SELECT** ».

Des quatre instructions du LMD, « **SELECT** » est celle qui est exécutée le plus souvent dans une application réelle, car les enregistrements sont plus souvent lus qu'ils ne sont modifiés.

L'instruction « **SELECT** » est un outil puissant et sa syntaxe est compliquée en raison des nombreuses possibilités qui vous sont offertes pour former une instruction valide en combinant les tables, les colonnes, les fonctions et les opérateurs. Par conséquent, au lieu d'examiner la syntaxe complète de cette instruction, on va commencer par découvrir la syntaxe au fur et à mesure de son utilisation.

```
SELECT [ALL | DISTINCT] { [{ NOM_TABLE | ALIAS2 }.]*,  
  [ [{NOM_TABLE | ALIAS2}.]COLONNE1 [AS] ALIAS1[,...]
```

**FROM NOM\_TABLE ALIAS2;**

<b>ALL</b>	La requête extrait l'intégralité des enregistrements de la table. C'est l'option par défaut.
<b>DISTINCT UNIQUE</b>	La requête extrait les enregistrements de la table qui sont uniques, la règle d'unicité s'applique à l'ensemble des colonnes sélectionnées.
<b>*</b>	La projection totale, permet d'extraire l'ensemble des colonnes pour la table mentionné dans la clause « <b>FROM</b> ».
<b>COLONNE</b>	Une liste des noms de colonnes séparées par virgule, de la table mentionnée dans la clause « <b>FROM</b> », que vous souhaitez extraire dans la projection. Le nom de chaque colonne peut être préfixé par le nom de la table ou par l'alias défini pour cette table. Attention, si vous avez défini un alias pour le nom de la table, vous ne pouvez plus utiliser le nom de la table, mais uniquement l'alias.
<b>[AS] ALIAS</b>	Si l'en-tête de colonne n'est pas assez significatif, il est possible de définir un alias qui se déclare immédiatement après la colonne ; il peut être précédé par « <b>AS</b> », sous la forme d'une chaîne de caractères placée ou non entre guillemets. Il est également possible de définir un alias pour le nom de la table.
<b>FROM</b>	La table d'où vous souhaitez extraire les données.

La requête suivante est une projection totale de la table CATEGORIES.



```
SQL> DESC CATEGORIES
Nom                                NULL ?    Type
-----
CODE_CATEGORIE                    NOT NULL  NUMBER(6)
NOM_CATEGORIE                     NOT NULL  VARCHAR2(25)
DESCRIPTION                        NOT NULL  VARCHAR2(100)
```

```
SQL> SELECT * FROM CATEGORIES ;
```

```
CODE_CA  NOM_CATEGORIE                DESCRIPTION
-----
1 Boissons                               Boissons, cafés, thés, bières
2 Condiments                           Sauces, assaisonnements et épices
3 Desserts                             Desserts et friandises
4 Produits laitiers                     Fromages
...
```

Dans l'exemple précédent, la requête extrait l'ensemble des colonnes et des enregistrements de la table CATEGORIES. En pratique on utilise très rarement la projection totale car les informations dont a besoin portent sur une partie des colonnes de la table. Une projection partielle est plus appropriée du point de vue de la lisibilité du rapport ainsi que des traitements sur le serveur et les transferts de données à travers le réseau.

La requête suivante est une projection partielle des tables CATEGORIES et COMMANDES.



```
SQL> SELECT CODE_CATEGORIE CODE, NOM_CATEGORIE "Catégorie de produits"
2 FROM CATEGORIES ;
```

```
CODE  Catégorie de produits
-----
1 Boissons
2 Condiments
```

```

3 Desserts
4 Produits laitiers
5 Pâtes et céréales
6 Viandes
7 Produits secs
...

SQL> SELECT CODE_CLIENT, DATE_COMMANDE, NO_EMPLOYE, PORT FROM COMMANDES ;

CODE_  DATE_COMMA NO_EMPLOYE      PORT
-----
ANTON  18/05/2010      12      87,3
TRAIH  18/05/2010      79      51,9
FOLIG  18/05/2010     110      80,4
GALED  18/05/2010      69      98,5
...

```

Dans les deux exemples précédents, vous avez remarquées que **SQL\*Plus** applique une certaine mise en forme aux données qu'il présente.

La présentation de résultats se fait sous forme tabulaire où :

- Il convertit tous les noms de colonnes ou les alias, qui ne sont pas placées entre les guillemets, en majuscules.
- Les en-têtes de colonnes ne peuvent pas être plus longs que la longueur définie des colonnes.

### Attention

Les guillemets « " » sont utilisés seulement pour définir l'alias d'une colonne, par exemple "Alias de Colonne". Le symbole de délimitation des chaînes de caractères est la simple cote « ' », par exemple : 'Chaîne de caractères'.

La requête suivante est une projection partielle de la table EMPLOYES, pour extraire les différentes fonctions des employés.

```

SQL> SELECT FONCTION FROM EMPLOYES;

FONCTION
-----
Représentant(e)
Représentant(e)
Assistante commerciale
...

111 ligne(s) sélectionnée(s).

SQL> SELECT DISTINCT FONCTION FROM EMPLOYES;

FONCTION
-----
Assistante commerciale
Chef des ventes
Président
Représentant(e)
Vice-Président

```

Dans le premier exemple, on peut remarquer que la clause « **ALL** » et la requête extraient l'intégralité des enregistrements de la table.

Dans le deuxième exemple la requête extrait les enregistrements de la table qui sont uniques, la règle d'unicité s'applique à la seule colonne sélectionnée.



```
SQL> SELECT C.* FROM CATEGORIES C;
```

CODE_CATEGORIE	NOM_CATEGORIE	DESCRIPTION
1	Boissons	Boissons, cafés, thés, bières
...		

L'exemple précédent illustre l'utilisation de l'alias pour le nom de la table CATEGORIES.



### Attention

Attention une fois que vous avez défini un alias pour la table, vous ne pouvez plus utiliser le nom de la table dans la requête.

L'alias de la table peut être utilisé pour préfixer chaque colonne ; dans les requêtes multi-tables il est également utilisé pour préfixer la projection totale « \* » d'une table.

## Les constantes

Une constante est une variable dont la valeur, fixée au moment de sa définition, n'est pas modifiable.

### Constante numérique

Une constante numérique définit un nombre contenant éventuellement un signe, un point décimal et un exposant, puissance de dix. Le point décimal ne peut être défini que par le caractère point « . ». Les caractères numériques doivent être contigus (sans espaces pour milliers par exemple).



```
SQL> SELECT 0, -1234567890, +1234567890, -123.456, +123.456, -1E+123
2 FROM DUAL;
```

0	-1234567890	+1234567890	-123.456	+123.456	-1E+123
0	-1,235E+09	1234567890	-123,456	123,456	-1,00E+123

```
SQL> SELECT -123,456, +123,456,-1E+123 FROM DUAL;
```

-123	456	+123	456	-1E+123
-123	456	123	456	-1,00E+123

### Constante chaîne de caractère

Une constante chaîne de caractère est représentée par une chaîne de caractères entre cotes « ' » où les lettres en majuscules et en minuscules sont considérées comme deux caractères différents. Il est possible d'insérer une apostrophe à l'intérieur d'une chaîne de caractères en la représentant par deux apostrophes consécutives.



```
SQL> SELECT 'Bonjour aujourd'hui c'est le : ' "Aujourd'hui",
2 SYSDATE "Date" FROM DUAL;
```

Aujourd'hui	Date
Bonjour aujourd'hui c'est le :	11/02/2011

11g

À partir de la version Oracle 11g il est possible de personnaliser le caractère délimiteur d'une constante chaîne de caractères. La constante chaîne de caractère peut être définie à l'aide de la syntaxe suivante : **q 'délimiteur chaîne délimiteur'**

<b>q</b>	L'opérateur indique que le mécanisme de remplacement de cotation sera utilisé.
<b>délimiteur</b>	Le caractère délimiteur est tout caractère unique, sauf le caractère espace ou tabulation. Dans tous les cas, le caractère délimiteur de l'ouverture et de clôture doit être le même caractère.
<b>chaîne</b>	Une chaîne de caractères qui peut également contenir le caractère délimiteur tant qu'il n'est pas immédiatement suivi par un guillemet simple.

Voici l'exemple précédent cette fois-ci avec un mécanisme de remplacement de cotation avec le délimiteur « { } ».



```
SQL> SELECT q'{Bonjour aujourd'hui c'est le :}' "Aujourd'hui",
2  SYSDATE "Date" FROM DUAL;

Aujourd'hui          Date
-----
Bonjour aujourd'hui c'est le : 11/02/2011

SQL> SELECT q'"Voici le délimiteur " dans la chaîne!"' FROM DUAL;

Q' "VOICILEDÉLIMITEUR"DANSLACHAÎNE!" '
-----
Voici le délimiteur " dans la chaîne!

SQL> SELECT q'~chaîne~chaîne~' q1, q'#chaîne#chaîne#' q2,
2  q'!chaîne!chaîne!' q3, q'|chaîne|chaîne|' q4
3  FROM DUAL;

Q1          Q2          Q3          Q4
-----
chaîne~chaîne chaîne#chaîne chaîne!chaîne chaîne'|chaîne

SQL> SELECT q'<chaîne<['|']>chaîne>' q5,
2  q'%chaîne[%|chaîne%' q6, q'aaaa' q7 FROM DUAL;

Q5          Q6          Q7
-----
chaîne<['|']>chaîne chaîne[%|chaîne aa
```

## Opérateur de concaténation

La concaténation est le seul opérateur disponible des chaînes de caractères. Le résultat d'une concaténation est la chaîne de caractères obtenue en mettant bout à bout les deux chaînes de caractères passées en arguments.

Cet opérateur se note au moyen de deux caractères barre verticale accolés « || », selon la syntaxe présente dans l'exemple suivant. Une projection partielle de la table EMPLOYES, pour extraire une

chaîne de caractères qui résulte de la concaténation du numéro employé, nom, prénom et date de naissance.



```
SQL> SELECT NO_EMPLOYE || ' -- ' || NOM || ' -- ' || PRENOM ||  
2 ' -- ' || DATE_NAISSANCE "Liste des employés"  
3 FROM EMPLOYES ;
```

Liste des employés

```
-----  
70 -- Berlioz -- Jacques -- 12/10/1967  
71 -- Nocella -- Guy -- 19/11/1976  
72 -- Herve -- Didier -- 19/09/1979  
73 -- Mangeard -- Jocelyne -- 27/12/1979  
74 -- Cazade -- Anne-Claire -- 19/09/1979  
75 -- Devie -- Thérèse -- 26/03/1968  
76 -- Peacock -- Margaret -- 20/03/1969  
77 -- Idesheim -- Annick -- 29/09/1982  
78 -- Rollet -- Philippe -- 09/03/1985  
79 -- Silberreiss -- Albert -- 18/08/1978  
80 -- Weiss -- Sylvie -- 08/01/1978  
81 -- Delorgue -- Jean -- 11/07/1981  
82 -- Zonca -- Virginie -- 12/03/1980  
83 -- Twardowski -- Colette -- 13/01/1979  
84 -- Coutou -- Myriam -- 31/08/1985  
85 -- King -- Robert -- 10/02/1975  
86 -- Ragon -- André -- 21/05/1989  
87 -- Dohr -- Sylvie -- 30/12/1982  
88 -- Maurousset -- James -- 26/09/1978  
89 -- Pouetre -- Camille-Hélène -- 16/03/1987  
90 -- Montesinos -- Aline -- 31/07/1972  
91 -- Aubert -- Maria -- 02/11/1984  
92 -- Thomas -- Patricia -- 13/12/1983  
93 -- Falatik -- Bernard -- 24/10/1981  
94 -- Marielle -- Michel -- 11/12/1990  
...
```

### Conseil



L'opérateur de concaténation peut travailler avec des expressions qui retournent une chaîne de caractère, un numérique ou une date, des constantes de type chaînes de caractères et numériques; les conversions entre ces différents types sont effectuées implicitement.

## Opérateurs arithmétiques

Une expression arithmétique est une combinaison de noms de colonnes, de constantes et de fonctions arithmétiques (les fonctions arithmétiques sont traitées plus loin) combinées au moyen des **opérateurs** arithmétiques addition « + », soustraction « - », multiplication « \* » ou division « / ».

Une expression arithmétique peut comporter plusieurs opérateurs. Dans ce cas, le résultat de l'expression peut varier selon l'ordre dans lequel les opérations sont effectuées.

La priorité des opérateurs :

- La multiplication et la division sont prioritaires par rapport à l'addition et à la soustraction.



- Les opérateurs de même priorité sont évalués de la gauche vers la droite.
- Les parenthèses sont utilisées pour forcer la priorité de l'évaluation et pour clarifier les instructions SQL.

L'exemple suivant illustre une projection de la table `PRODUIT` pour extraire le nom du produit, la valeur du stock et la valeur commandée.



```
SQL> SELECT NOM_PRODUIT "Produit", PRIX_UNITAIRE*UNITES_STOCK "Stock",
2 UNITES_COMMANDEES*12 "Commandes" FROM PRODUITS;
```

Produit	Stock	Commandes
Chai	3510	
Chang	1615	480
Aniseed Syrup	650	840
Chef Anton's Cajun Seasoning	5830	
Grandma's Boysenberry Spread	15000	
Uncle Bob's Organic Dried Pears	2250	
Northwoods Cranberry Sauce	1200	
Mishi Kobe Niku		
Ikura	4805	
Queso Cabrales	2310	360
Queso Manchego La Pastora	16340	
Konbu	720	
Alice Mutton		
Teatime Chocolate Biscuits	1150	
Sir Rodney's Marmalade	16200	
Sir Rodney's Scones	150	480
...		

### Attention



Les constantes numériques sont saisies avec ou sans signe sans espace entre les caractères et le caractère de séparation des décimales est le point « . ».

L'exemple suivant illustre une projection de la table `EMPLOYES` pour extraire le nom de l'employé et une prévision de salaire à la suite d'une augmentation de 10%.



```
SQL> SELECT NOM || ' ' || PRENOM "Employé", SALAIRE * 1.1 "Nouveau Salaire"
2 FROM EMPLOYES ;
```

Employé	Nouveau Salaire
Berlioz Jacques	10340
Nocella Guy	8360
Herve Didier	7370
Mangeard Jocelyne	9020
Cazade Anne-Claire	7920
Devie Thérèse	1694
Peacock Margaret	6710
Idesheim Annick	8030
Rollet Philippe	5500
Silberreiss Albert	10120
Weiss Sylvie	7260
Delorgue Jean	6490
...	

SQL propose deux opérations possibles des expressions de type date.

L'ajout d'un nombre de jours à une date, le résultat étant une expression de type date.

**DATE1 (+ ou -) NOMBRE = DATE2**

Le calcul du nombre de jours séparant les deux dates, le résultat étant une expression de type numérique.

**DATE1 - DATE2 = NOMBRE**

Le résultat peut être exprimé sous forme de valeur décimale si les valeurs de DATE1 et/ou de DATE2 contiennent une notion d'heure.

« **SYSDATE** » est une pseudocolonne que l'on peut utiliser dans une expression de type date et qui a pour valeur la date et l'heure courantes du système d'exploitation hôte.

La requête suivante est une projection de la table EMPLOYES pour extraire le nom de l'employé, sa date de naissance et son âge.



```
SQL> SELECT NOM,DATE_NAISSANCE,DATE_NAISSANCE+1,
2 (SYSDATE-DATE_NAISSANCE)/365 FROM EMPLOYES ;
```

NOM	DATE_NAISS	DATE_NAISS	(SYSDATE-DATE_NAISSANCE)/365
-----	-----	-----	-----
Berlioz	12/10/1967	13/10/1967	43,3634437
Nocella	19/11/1976	20/11/1976	34,2511149
Herve	19/09/1979	20/09/1979	31,4182382
Mangeard	27/12/1979	28/12/1979	31,1470054
Cazade	19/09/1979	20/09/1979	31,4182382
Devie	26/03/1968	27/03/1968	42,9086492
Peacock	20/03/1969	21/03/1969	41,9250876
Idesheim	29/09/1982	30/09/1982	28,3881012
Rollet	09/03/1985	10/03/1985	25,9442656
Silberreiss	18/08/1978	19/08/1978	32,5059095
...			

## La valeur NULL

Une valeur « **NULL** » en SQL est une valeur non définie. Lorsque l'un des termes d'une expression a la valeur « **NULL** », l'expression entière prend la valeur « **NULL** ». D'autre part, un prédicat comportant une comparaison avec une expression ayant la valeur « **NULL** » prendra toujours la valeur FAUX.

La requête suivante est une projection de la table EMPLOYES pour extraire le nom de l'employé, son salaire, sa commission et la somme perçue.



```
SQL> SELECT NOM, SALAIRE, COMMISSION, SALAIRE+COMMISSION FROM EMPLOYES ;
```

NOM	SALAIRE	COMMISSION	SALAIRE+COMMISSION
-----	-----	-----	-----
...			
Cazade	7200	550	7750
<b>Devie</b>	<b>1540</b>		
Peacock	6100	930	7030
...			

L'exemple montre qu'une valeur « **NULL** » ne peut pas être utilisée dans un calcul; ainsi que nous l'avons indiqué plus haut, cette valeur n'est pas égale à zéro; il faut plutôt la considérer comme étant une valeur inconnue.

Lorsque l'un des termes d'une expression a la valeur « **NULL** », l'expression entière prend la valeur « **NULL** » ; pour pouvoir travailler avec des champs qui contiennent des valeurs « **NULL** », il faut une fonction qui puisse gérer cette valeur.

### NVL

La fonction « **NVL** » permet de remplacer une valeur « **NULL** » par une valeur significative.

**NVL (EXPRESSION1, EXPRESSION2) = VALEUR\_DE\_RETOUR**

<b>EXPRESSION1</b>	Une expression qui peut retourner la valeur « <b>NULL</b> ».
<b>EXPRESSION2</b>	La valeur de remplacement dans le cas où EXPRESSION1 est égale à « <b>NULL</b> ». EXPRESSION2 doit être de même type que EXPRESSION1.
<b>VALEUR_DE_RETOUR</b>	Est égale à EXPRESSION2 si EXPRESSION1 est égale à « <b>NULL</b> » si non EXPRESSION1.

Tous les types de données caractères, numériques et dates peuvent être utilisés.

La requête suivante est une sélection des neuf premiers enregistrements de la table **CLIENTS** pour extraire la société et le numéro de fax.



```
SQL> SELECT SOCIETE, NVL(FAX, 'Non affecté') FROM CLIENTS;
```

SOCIETE	NVL(FAX, 'NONAFFECTÉ')
...	
Eastern Connection	(71) 555-3373
Ernst Handel	7675-3426
<b>Familia Arquibaldo</b>	<b>Non affecté</b>
...	

La requête suivante est une projection de la table **EMPLOYES** pour extraire le nom de l'employé, son salaire, sa commission et la somme perçue.

```
SQL> SELECT NOM, SALAIRE, COMMISSION, SALAIRE+NVL(COMMISSION,0) R
2 FROM EMPLOYES ;
```

NOM	SALAIRE	COMMISSION	R
-----	-----	-----	-----
Berlitz	9400	980	10380
Nocella	7600	910	8510
Herve	6700	1170	7870
Mangard	8200	190	8390
Cazade	7200	550	7750
<b>Devie</b>	<b>1540</b>		<b>1540</b>
Peacock	6100	930	7030
Idesheim	7300	600	7900
Rollet	5000	570	5570
Silberreiss	9200	1100	10300
...			

### NVL2

La fonction « **NVL2** » permet de remplacer une valeur « **NULL** » par une valeur significative.

**NVL (EXPRESSION1, EXPRESSION2, EXPRESSION3) = VALEUR\_DE\_RETOUR**

<b>VALEUR_DE_RETOUR</b>	Est égale à EXPRESSION3 si EXPRESSION1 est égale à « <b>NULL</b> » si non EXPRESSION2.
-------------------------	--



```
SQL> SELECT NOM, SALAIRE, COMMISSION,
2 NVL2( COMMISSION, SALAIRE+COMMISSION, SALAIRE) R FROM EMPLOYES ;
```

NOM	SALAIRE	COMMISSION	R
-----	-----	-----	-----
Berlioz	9400	980	10380
Nocella	7600	910	8510
Herve	6700	1170	7870
Mangeard	8200	190	8390
Cazade	7200	550	7750
<b>Devie</b>	<b>1540</b>		<b>1540</b>
Peacock	6100	930	7030
Idesheim	7300	600	7900
Rollet	5000	570	5570
Silberreiss	9200	1100	10300
...			

## Tri du résultat d'une requête

Les lignes constituant le tableau résultat d'un ordre « **SELECT** » sont affichées dans un ordre indéterminé qui dépend des algorithmes internes du moteur du système de gestion de bases de données relationnelles.

En revanche, on peut, dans l'ordre « **SELECT** », demander que le résultat soit trié avant l'affichage selon un ordre ascendant ou descendant, en fonction d'un ou de plusieurs critères. Il est possible d'utiliser jusqu'à 16 critères de tri.

Les critères de tri sont spécifiés dans une clause « **ORDER BY** », figurant en dernière position de l'ordre « **SELECT** ».

La syntaxe de l'instruction « **SELECT** » :

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
FROM NOM_TABLE
ORDER BY
    [NOM_COLONNE1 | ALIAS1 | POSITION1 | EXPRESSION1][ASC|DESC],
    [NOM_COLONNE2 | ALIAS2 | POSITION2 | EXPRESSION2][ASC|DESC]
[,...] ;
```

**NOM\_COLONNE**

Le nom de la colonne qui fournit la valeur qui entre en ligne de compte pour le tri. La colonne peut ou non faire partie des colonnes extraites par la requête mais elle doit être une des colonnes des tables mentionnées dans « **FROM** ».

**ALIAS**

L'alias de l'expression ou de la colonne qui fournit la valeur qui entre en ligne de compte pour le tri.

**EXPRESSION**

L'expression qui fournit la valeur qui entre en ligne de compte pour le tri.

**POSITION**

L'expression ou la colonne, identifiés par la position dans la clause « **SELECT** », qui fournit la valeur qui entre en ligne de compte pour le tri.

## ASC

Le critère de tri est ascendant pour NOM\_COLONNE ou EXPRESSION ou POSITION qui précède le critère. Les critères sont définis pour chaque expression si vous ne le précisez pas. Par défaut, il est ascendant.

## DESC

Le critère de tri est descendant pour NOM\_COLONNE ou EXPRESSION ou POSITION qui précède le critère. Par défaut, il est ascendant.

La requête suivante est une sélection de la table EMPLOYES pour extraire le nom, prénom et fonction ; les résultats doivent être triés par le nom de l'employé.



```
SQL> SELECT NOM, PRENOM, FONCTION FROM EMPLOYES ORDER BY NOM;
```

NOM	PRENOM	FONCTION
Alvarez	Marcel	Représentant(e)
Arrambide	Jean Pierre	Représentant(e)
Aubert	Maria	Représentant(e)
Aubry	Jean-Claude	Représentant(e)
Barre	Jean-Paul	Représentant(e)
Bazart	Régis	Représentant(e)
Belin	Chantal	Chef des ventes
Berlioz	Jacques	Représentant(e)
Besse	José	Représentant(e)
Bettan	Henri-Michel	Représentant(e)
Blard	Jean-Benoît	Représentant(e)
Bodard	René	Représentant(e)
Brasseur	Hervé	Vice-Président
Brunet	Jean-Luc	Représentant(e)
...		

```
SQL> SELECT NOM||' '||PRENOM "Employé", FONCTION FROM EMPLOYES
2 ORDER BY "Employé" ASC;
```

Employé	FONCTION
Alvarez Marcel	Représentant(e)
Arrambide Jean Pierre	Représentant(e)
Aubert Maria	Représentant(e)
Aubry Jean-Claude	Représentant(e)
Barre Jean-Paul	Représentant(e)
Bazart Régis	Représentant(e)
Belin Chantal	Chef des ventes
Berlioz Jacques	Représentant(e)
Besse José	Représentant(e)
Bettan Henri-Michel	Représentant(e)
Blard Jean-Benoît	Représentant(e)
Bodard René	Représentant(e)
Brasseur Hervé	Vice-Président
Brunet Jean-Luc	Représentant(e)
Buchanan Steven	Chef des ventes
...	

```
SQL> SELECT NOM||' '||PRENOM "Employé", FONCTION FROM EMPLOYES
2 ORDER BY 1 DESC;
```

Employé	FONCTION
-----	-----
Zonca Virginie	Représentant(e)
Ziliox Francoise	Assistante commerciale
Weiss Sylvie	Représentant(e)
Viry Yvan	Représentant(e)
Valot Alain	Représentant(e)
Urbaniak Isabelle	Représentant(e)
Twardowski Colette	Représentant(e)
Tourtrel Nicole	Représentant(e)
Thomas Patricia	Représentant(e)
Thimoleon Georges	Représentant(e)
Teixeira Claudia	Représentant(e)
...	

### Note

Le tri se fait d'abord selon le premier critère spécifié dans la clause « **ORDER BY** », puis les lignes ayant la même valeur pour le premier critère sont triées selon le deuxième critère de la clause « **ORDER BY** », etc.

La requête suivante est une sélection de la table PRODUIT pour extraire les produits, les fournisseurs et les catégories de produits avec les résultats ordonnés par fournisseur et catégorie produits.

```
SQL> SELECT NOM_PRODUIT, NO_FOURNISSEUR, CODE_CATEGORIE FROM PRODUITS
2 ORDER BY NO_FOURNISSEUR, CODE_CATEGORIE DESC;
```

NOM_PRODUIT	NO_FOURNISSEUR	CODE_CATEGORIE
-----	-----	-----
Aniseed Syrup	1	2
Chai	1	1
Chang	1	1
Chef Anton's Cajun Seasoning	2	2
Chef Anton's Gumbo Mix	2	2
Louisiana Hot Spiced Okra	2	2
Louisiana Fiery Hot Pepper Sauce	2	2
Uncle Bob's Organic Dried Pears	3	7
Grandma's Boysenberry Spread	3	2
Northwoods Cranberry Sauce	3	2
Ikura	4	8
Longlife Tofu	4	7
Mishi Kobe Niku	4	6
...		

### Attention

Si un attribut sur lequel porte un critère de tri contient la valeur « **NULL** », les lignes correspondantes sont affichées en dernier. Oracle traite les valeurs « **NULL** » comme si elles étaient des valeurs infinies.

La requête suivante est une sélection de la table EMPLOYEES pour extraire le nom, le prénom le salaire et la commission avec les résultats ordonnés par commission.

```
SQL> SELECT NOM,PRENOM,SALAIRE,COMMISSION
2 FROM EMPLOYEES
3 ORDER BY COMMISSION;
```

NOM	PRENOM	SALAIRE	COMMISSION
-----	-----	-----	-----

```

...
Frederic      Jean Marie      8100      1870
Letertre      Sylvie          5600      1890
Tourtel       Nicole          8100      1920
Menetrier     Alexandra       8700      1970
Burst         Jean-Yves       7700      1980
Ragon         André           13000     5980
Belin         Chantal         10000     10640
Leger         Pierre          19000     11150
Chambaud      Axelle          12000     11600
Buchanan      Steven          13000     12940
Splingart     Lydia           16000     16480
Guerdon       Béatrice        1700
Grangirard    Patricia        1700
Etienne       Brigitte        2000
Giroux        Jean-Claude     150000
Devie         Thérèse         1540
Callahan      Laura           1200
Lampis        Gabrielle       1300
...

```

**10g**

Oracle traite les valeurs « **NULL** » comme si elles étaient des valeurs infinies; on peut donc ajouter une clause « **ORDER BY** » avec les mots clés « **NULLS FIRST** » et « **NULLS LAST** ».

```

SQL> SELECT NOM, PRENOM, SALAIRE, COMMISSION FROM EMPLOYES
2 ORDER BY COMMISSION NULLS FIRST;

```

NOM	PRENOM	SALAIRE	COMMISSION
-----	-----	-----	-----
Ziliox	Francoise	1900	
Guerdon	Béatrice	1700	
Grangirard	Patricia	1700	
Devie	Thérèse	1540	
Maurer	Véronique	1400	
Poupard	Claudette	1800	
Fuller	Andrew	96000	
Etienne	Brigitte	2000	
Lampis	Gabrielle	1300	
Callahan	Laura	1200	
Brasseur	Hervé	147000	
Pouetre	Camille-Hélène	2000	
Giroux	Jean-Claude	150000	
Petit	Michel	6300	20
Gregoire	Renée	8000	30
...			

```

SQL> SELECT NOM, PRENOM, SALAIRE, COMMISSION FROM EMPLOYES
2 ORDER BY COMMISSION NULLS LAST;

```

NOM	PRENOM	SALAIRE	COMMISSION
-----	-----	-----	-----
...			
Chambaud	Axelle	12000	11600
Buchanan	Steven	13000	12940
Splingart	Lydia	16000	16480
Guerdon	Béatrice	1700	
Grangirard	Patricia	1700	

Etienne	Brigitte	2000
Giroux	Jean-Claude	150000
...		

## La pseudo-colonne ROWNUM

« **ROWNUM** » retourne une valeur numérique entière qui indique l'ordre de sélection de la ligne au moment de l'exécution de la requête. La valeur « **ROWNUM** » est associée à chaque ligne avant la prise en compte d'une éventuelle clause « **ORDER BY** ».

La requête suivante est une sélection de la table **PRODUIT** pour extraire les dix premiers enregistrements affichant les produits, les fournisseurs et les catégories de produits avec un ordre de tri par fournisseur et catégorie produits.



```
SQL> SELECT NOM_PRODUIT, NO_FOURNISSEUR NO, CODE_CATEGORIE CODE, ROWNUM
2 FROM PRODUITS WHERE ROWNUM <= 10
3 ORDER BY NO_FOURNISSEUR, CODE_CATEGORIE DESC;
```

NOM_PRODUIT	NO	CODE	ROWNUM
Aniseed Syrup	1	2	3
Chai	1	1	1
Chang	1	1	2
Chef Anton's Cajun Seasoning	2	2	4
Uncle Bob's Organic Dried Pears	3	7	6
Grandma's Boysenberry Spread	3	2	5
Northwoods Cranberry Sauce	3	2	7
Ikura	4	8	9
Mishi Kobe Niku	4	6	8
Queso Cabrales	5	4	10

Dans l'exemple précédent vous pouvez remarquer les valeurs de la pseudocolonne « **ROWNUM** » qui ont été attribuées avant le traitement de tri.

## Table DUAL

Oracle fournit une petite table appelée « **DUAL** » qui se compose d'une ligne et d'une colonne qui est utilisée pour tester des fonctions ou effectuer des calculs rapides.



```
SQL> DESC DUAL
```

Nom	NULL ?	Type
DUMMY		VARCHAR2(1)

```
SQL> SELECT * FROM DUAL;
```

```
D
-
X
```

```
SQL> INSERT INTO DUAL VALUES ('A');
INSERT INTO DUAL VALUES ('A')
*
ERREUR à la ligne 1 :
```



ORA-01031: privilèges insuffisants

Il faut remarquer que dans la table « **DUAL** » vous ne pouvez effectuer aucune opération de mise à jour des données.

Étant donné que les nombreuses fonctions d'Oracle peuvent opérer sur les colonnes et les littéraux, l'emploi de « **DUAL** » permet d'observer l'agissement des fonctions simplement en utilisant des chaînes.

Les colonnes qui existent dans « **DUAL** » n'ont aucune importance. Vous pouvez donc facilement expérimenter les formats et les calculs de date au moyen de cette table et des fonctions spéciales, afin d'en comprendre le fonctionnement avant de les appliquer sur des données de tables réelles.

Exemple : Dans ces exemples, l'instruction « **SELECT** » ne tient pas compte des colonnes de la table, et une seule ligne suffit à démontrer un fonctionnement. Par exemple, supposons que vous souhaitiez rapidement afficher la date de demain, le nom d'utilisateur et calculer  $(2434 / 3.14) * 16.24$ .



```
SQL> SELECT USER, SYSDATE+1, ( 2434 / 3.14 ) * 16.24 FROM DUAL;
```

USER	SYSDATE+1	( 2434 / 3.14 ) * 16.24
STAG	11/02/2011	12588,586

Il existe plusieurs pseudocolonnes qui peuvent être utilisées dans les requêtes SQL ou tout simplement pour afficher leur valeur à l'aide de la table « **DUAL** ». Toutes les pseudocolonnes de l'exemple suivant sont détaillées dans les modules suivants.



```
SQL> SELECT UID, USER, SYSDATE, SYSTIMESTAMP, ROWID FROM DUAL;
```

UID	USER	SYSDATE
80	STAGIAIRE	10/02/2011
10/02/2011 18:00:11,071000 +01:00		
AAAAB0AAAAAAOAAAA		

## La limitation d'enregistrements



A partir de la version Oracle 12c il est possible de limiter le nombre d'enregistrements retournés par une requête. Elle retourne uniquement les premiers n enregistrements. La syntaxe que vous devez ajouter impérativement après la clause « **ORDER BY** » est :

```
SELECT ... ORDER BY
```

```
[ OFFSET valeur ROW[S]] [ FETCH { FIRST | NEXT }
[ valeur [PERCENT]] ROW[S] { ONLY | WITH TIES } ] ;
```

**FIRST / NEXT** La syntaxe permet d'utiliser un ou l'autre sans aucune différence. La valeur qui suit cette clause permet de définir combien d'enregistrements on veut lire.

**PERCENT** La valeur précédente ne représente pas un nombre d'enregistrements mais un pourcentage.

**WITH TIES**

Dans le cas où vous définissez un ordre de tri et plusieurs enregistrements qui ont une valeur pour la ou les colonnes triées égale au dernier enregistrement, tous ces enregistrements sont retournés.

**OFFSET**

La valeur suivante détermine le nombre d'enregistrements à partir des quels on commence la lecture.



```
STAG@topaze>SELECT CODE_CATEGORIE,NOM_CATEGORIE FROM CATEGORIES ;
```

```
CODE_CATEGORIE  NOM_CATEGORIE
```

```
-----
1 Boissons
2 Condiments
3 Desserts
4 Produits laitiers
5 Pâtes et céréales
6 Viandes
7 Produits secs
8 Poissons et fruits de mer
9 Conserves
10 Viande en conserve
```

```
STAG@topaze>SELECT CODE_CATEGORIE,NOM_CATEGORIE FROM CATEGORIES
2  FETCH FIRST 3 ROWS ONLY;
```

```
CODE_CATEGORIE  NOM_CATEGORIE
```

```
-----
1 Boissons
2 Condiments
3 Desserts
```

```
STAG@topaze>SELECT CODE_CATEGORIE,NOM_CATEGORIE FROM CATEGORIES
2  OFFSET 5 ROWS FETCH NEXT 3 ROWS ONLY;
```

```
CODE_CATEGORIE  NOM_CATEGORIE
```

```
-----
6 Viandes
7 Produits secs
8 Poissons et fruits de mer
```

```
STAG@topaze>SELECT NOM, PRENOM, FONCTION
2  FROM EMPLOYES FETCH NEXT 1 PERCENT ROWS ONLY;
```

```
NOM      PRENOM  FONCTION
```

```
-----
Berlioz  Jacques  Représentant(e)
Nocella  Guy      Représentant(e)
```

```
STAG@topaze>SELECT NOM_PRODUIT, UNITES_STOCK FROM PRODUITS
2  ORDER BY UNITES_STOCK DESC NULLS LAST
3  FETCH NEXT 5 ROWS ONLY;
```

```
NOM_PRODUIT                                UNITES_STOCK
-----
Potato Chips                                200
Rhönbräu Klosterbier                        125
```

```

Green Tea                                     125
Boston Crab Meat                             123
Grandma's Boysenberry Spread                 120      <-----

5 lignes sélectionnées.

STAG@topaze>
STAG@topaze>SELECT NOM_PRODUIT, UNITES_STOCK FROM PRODUITS
  2  ORDER BY UNITES_STOCK DESC NULLS LAST
  3  FETCH NEXT 5 ROWS WITH TIES;                                     <-----

NOM_PRODUIT                                UNITES_STOCK
-----
Potato Chips                               200
Rhönbräu Klosterbier                       125
Green Tea                                  125
Boston Crab Meat                           123
Grandma's Boysenberry Spread               120
Long Grain Rice - Gnocchi                120
Crab Meat                               120

```

Cette option est intéressante pour ne pas tenir compte de la taille des tables pendant la conception des requêtes, car en production, les tables ne sont pas aussi petites que le jeu d'exemples présentés ici.



```

STAG@topaze>SET TIMING ON
STAG@topaze>SELECT COMMANDE,REF_PRODUIT,NO_EMPLOYE,QUANTITE,PRIX_UNITAIRE
  2  FROM INDICATEURS FETCH NEXT 5 ROWS ONLY;

COMMANDE    REF_PRODUIT NO_EMPLOYE    QUANTITE    PRIX_UNITAIRE
-----
24/06/1991      16         101         116         68,4
24/06/1991     116         101         199         90,96
24/06/1991      17         101          62         57,12
24/06/1991      98         101         118         56,4
24/06/1991      10         101          33         74,04

5 lignes sélectionnées.

Ecoulé : 00 :00 :00.02
STAG@topaze>SELECT COUNT(*) "Nombre d'enregistrements" FROM INDICATEURS;

Nombre d'enregistrements
-----
                        7643024

```

# 3

- *LIKE et REGEXP\_LIKE*
- *BETWEEN et IN*
- *AND et OR*
- *IS NULL*
- *NOT*

## Les opérateurs logiques



### Objectifs

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- Utiliser les opérateurs logiques simples et des opérateurs multiples.
- Mettre en œuvre les opérateurs de recherche dans les chaînes de caractères.
- Traiter les valeurs NULL des colonnes avec les opérateurs.
- Combiner plusieurs expressions de type logique à l'aide des opérateurs logiques AND, OR ou NOT.



### Contenu

La sélection ou restriction	3-2	Le traitement de la case	3-16
L'opérateur égal à	3-2	L'opérateur BETWEEN	3-18
Les opérateurs logiques	3-5	L'opérateur IN	3-19
L'opérateur LIKE	3-8	Les opérateurs logiques	3-20
La fonction REGEXP_LIKE	3-10	Les expressions logiques	3-22

## La sélection ou restriction

---

Les requêtes peuvent être extrêmement détaillées, ce qui rend parfois difficile la lecture des informations dont vous avez immédiatement besoin. De plus, l'intégration de données supplémentaires à une requête augmente inutilement les temps de traitement de la base de données. En pratique on utilise très rarement la projection, car elle extrait l'intégralité des enregistrements de la table ; souvent les informations nécessaires portent seulement sur un nombre restreint des enregistrements qui respectent une ou plusieurs conditions.

L'opérateur de **sélection**, aussi appelé **restriction**, permet de ne conserver pour un affichage que les lignes de la table qui vérifient une **condition** (ou prédicat) de sélection définie sur les valeurs prises par une ou plusieurs colonnes de la table.

L'ordre « **SELECT** » permet de spécifier les lignes à sélectionner par utilisation de la clause « **WHERE** ». Cette clause est suivie de la condition de sélection, évaluée pour chaque ligne de la table. Seules les lignes pour lesquelles la condition est vérifiée sont sélectionnées.

La syntaxe de l'instruction « **SELECT** » :

```
SELECT [ALL | DISTINCT] { [{ NOM_TABLE | ALIAS2 }.]*,  
  [ [{NOM_TABLE | ALIAS2}.]COLONNE1 [AS] ALIAS1[,...]}  
FROM NOM_TABLE ALIAS2  
WHERE PREDICAT  
ORDER BY  
  [NOM_COLONNE1 | ALIAS1 | POSITION1 | EXPRESSION1] [ASC | DESC]  
  [, ...] ;
```

<b>EXPRESSION</b>	La requête peut extraire de la base soit une colonne soit le résultat d'une expression, elle peut aussi afficher une constante.
-------------------	---

<b>PREDICAT</b>	Une ou plusieurs conditions qui doivent être satisfaites par un enregistrement pour qu'il soit extrait par la requête.
-----------------	--

Le prédicat est une opération logique qui nécessite pour sa mise en œuvre un ensemble d'opérateurs. La mise en œuvre des opérateurs logiques est le sujet suivant.

## L'opérateur égal à

---

Les opérateurs logiques présents permettent de comparer des expressions qui retournent une valeur unique. Tous ces opérateurs sont utilisés de façon analogue.

### **Note**

Les expressions peuvent être de l'un des trois types suivants : numérique, caractère ou date. Les trois types d'expression peuvent être comparés au moyen des opérateurs égal à, inférieur à ou supérieur à. Pour le type **date**, la relation d'ordre est l'ordre **chronologique**. Pour le type **caractère**, la relation d'ordre est l'ordre **alphabétique**.



## Egal à

L'opérateur logique **égal à** compare la valeur retournée par l'expression de gauche avec la valeur retournée par l'expression de droite ; si les deux valeurs sont égales, il retourne VRAI, sinon FAUX.

**EXPRESSION1 = EXPRESSION2**

La requête suivante est une sélection de la table CLIENTS pour extraire la société et l'adresse des clients français.



```
SQL> SELECT * FROM CLIENTS WHERE PAYS='France ';
```

aucune ligne sélectionnée

```
SQL> SELECT * FROM CLIENTS WHERE PAYS='FRANCE' ;
```

aucune ligne sélectionnée

```
SQL> SELECT SOCIETE, ADRESSE, PAYS FROM CLIENTS WHERE PAYS='France';
```

SOCIETE	ADRESSE	PAYS
Du monde entier	67, rue des Cinquante Otages	France
Folies gourmandes	184, chaussée de Tournai	France
France restauration	54, rue Royale	France
La corne d'abondance	67, avenue de l'Europe	France
La maison d'Asie	1 rue Alsace-Lorraine	France
Paris spécialités	265, boulevard Charonne	France
Spécialités du monde	25, rue Lauriston	France
Victuailles en stock	2, rue du Commerce	France
...		

### Attention



Attention, pour les prédicats qui utilisent les expressions de types caractères, les comparaisons sont effectuées en tenant compte des majuscules ou minuscules.

L'opérateur logique **égal à** compare **exactement** les deux expressions ; ainsi, si les deux chaînes de caractères ne sont pas identiques, il retourne « **FAUX** ».

La requête suivante est une sélection de la table EMPLOYES pour extraire les employés embauchés en '03/04/1991'.



```
SQL> SELECT NOM, PRENOM, DATE_EMBAUCE FROM EMPLOYES
2 WHERE DATE_EMBAUCE = '03/04/1991';
```

NOM	PRENOM	DATE_EMBAU
Thimoleon	Georges	03/04/1991
Maillard	Corinne	03/04/1991

La requête suivante est une sélection de la table EMPLOYES pour extraire les employés encadrés par l'employé numéro 2.



```
SQL> SELECT NOM, PRENOM, REND_COMPTE FROM EMPLOYES WHERE REND_COMPTE = 18;
```

NOM	PRENOM	REND_COMPTE
Ragon	André	18
Leger	Pierre	18
Belin	Chantal	18

Chambaud      Axelle      18

La requête suivante est une sélection de la table EMPLOYES pour extraire les employés qui n'ont pas la colonne COMMISSION renseignée.

```
SQL> SELECT NOM, PRENOM, REND_COMPTE FROM EMPLOYES WHERE COMMISSION = NULL;
```

aucune ligne sélectionnée

```
SQL> SELECT NOM, PRENOM, REND_COMPTE FROM EMPLOYES
2 WHERE NVL(COMMISSION,-1) = -1;
```

NOM	PRENOM	REND_COMPTE
-----	-----	-----
Devie	Thérèse	
Pouetre	Camille-Hélène	
Ziliox	Francoise	
Lampis	Gabrielle	
...		

### Attention

Attention il faut se rappeler que vous ne pouvez effectuer des opérations avec des expressions qui ont la valeur « **NULL** ».

Il est de même pour toute opération de comparaison des deux expressions.

Oracle permet d'employer des opérateurs logiques, « **=** », « **!=** », etc., avec « **NULL** » mais ce type de comparaison ne retourne généralement pas des résultats très parlants.

## IS NULL

L'opérateur logique « **IS NULL** » vérifie si la valeur retournée par EXPRESSION est égale à « **NULL** » ; alors il retourne VRAI, sinon FAUX.

### EXPRESSION IS NULL

La requête suivante est une sélection de la table EMPLOYES pour extraire les employés qui n'ont pas la colonne COMMISSION renseignée.

```
SQL> SELECT NOM, PRENOM, SALAIRE, COMMISSION FROM EMPLOYES
2 WHERE COMMISSION IS NULL;
```

NOM	PRENOM	SALAIRE	COMMISSION
-----	-----	-----	-----
Devie	Thérèse	1540	
Pouetre	Camille-Hélène	2000	
Ziliox	Francoise	1900	
Lampis	Gabrielle	1300	
Fuller	Andrew	96000	
Brasseur	Hervé	147000	
Poupard	Claudette	1800	
Maurer	Véronique	1400	
Callahan	Laura	1200	
Giroux	Jean-Claude	150000	
Etienne	Brigitte	2000	
Grangirard	Patricia	1700	
Guerdon	Béatrice	1700	

## IS NOT NULL

L'opérateur logique « **IS NOT NULL** » est la négation de l'opérateur « **IS NULL** ».

La requête suivante est une sélection de la table EMPLOYES pour extraire les employés qui ont la colonne COMMISSION renseignée.



```
SQL> SELECT NOM, PRENOM, SALAIRE, COMMISSION FROM EMPLOYES
2 WHERE COMMISSION IS NOT NULL;
```

NOM	PRENOM	SALAIRE	COMMISSION
-----	-----	-----	-----
Berlioz	Jacques	9400	980
Nocella	Guy	7600	910
Herve	Didier	6700	1170
Mangeard	Jocelyne	8200	190
...			

### Conseil

Les opérateurs logiques « **IS NULL** » et « **IS NOT NULL** » peuvent être utilisés pour tous les types de données qui sont stockés dans la base.



## Les opérateurs logiques

### Supérieur à

L'opérateur logique **supérieur à** compare la valeur retournée par l'expression de gauche avec la valeur retournée par l'expression de droite ; si elle est supérieure, il retourne VRAI, sinon FAUX.

**EXPRESSION1 { > | >= } EXPRESSION2**

La requête suivante est une sélection de la table EMPLOYES pour extraire le nom et le prénom des employés qui ont un salaire supérieur à 13000.



```
SQL> SELECT NOM, PRENOM, SALAIRE FROM EMPLOYES WHERE SALAIRE > 13000;
```

NOM	PRENOM	SALAIRE
-----	-----	-----
Leger	Pierre	19000
Fuller	Andrew	96000
Brasseur	Hervé	147000
Splingart	Lydia	16000
Giroux	Jean-Claude	150000

La requête suivante est une sélection de la table EMPLOYES pour extraire le nom et le prénom des employés qui ont été embauchés après 30/06/2003.



```
SQL> SELECT NOM, PRENOM, DATE_Eмбаuche FROM EMPLOYES
2 WHERE DATE_Eмбаuche > '30/06/2003';
```

NOM	PRENOM	DATE_Eмбаuche
-----	-----	-----
Marielle	Michel	30/07/2003
Regner	Charles	07/09/2003
Fuller	Andrew	09/08/2003
Pagani	Hector	02/07/2003
Chaussende	Maurice	10/08/2003



Cremel	Brigitte	28/08/2003
Gregoire	Renée	19/07/2003
Di Clemente	Luc	06/07/2003
Jacquot	Philippe	30/07/2003

La requête suivante est une sélection de la table EMPLOYES pour extraire le nom et le prénom des employés qui ont un nom supérieur à 'Twardowski' dans l'ordre alphabétique.

SQL> **SELECT NOM,PRENOM FROM EMPLOYES WHERE NOM > 'Twardowski' ORDER BY NOM;**

NOM	PRENOM
-----	-----
Urbaniak	Isabelle
Valot	Alain
Viry	Yvan
Weiss	Sylvie
Ziliox	Francoise
Zonca	Virginie

### Inférieur à

L'opérateur logique **inférieur à** compare la valeur retournée par l'expression de gauche avec la valeur retournée par l'expression de droite ; si elle est inférieure, il retourne VRAI, sinon FAUX.

**EXPRESSION1 { < | <= } EXPRESSION2**

La requête suivante est une sélection de la table EMPLOYES pour extraire le nom et le prénom des employés qui étaient en service avant le 01/01/1990.

SQL> **SELECT NOM, PRENOM, DATE\_EмбаUCHE FROM EMPLOYES**  
**2 WHERE DATE\_EмбаUCHE < '01/01/1990';**

NOM	PRENOM	DATE_EмбаU
-----	-----	-----
Peacock	Margaret	16/04/1989
Burst	Jean-Yves	23/10/1988
Malejac	Yannick	29/11/1987
Maurer	Véronique	27/05/1989
Alvarez	Marcel	03/08/1988

La requête suivante est une sélection de la table PRODUITS pour extraire les noms du produit, le numéro du fournisseur des produits livrés par le fournisseur numéro un et le fournisseur numéro deux.

SQL> **SELECT NOM\_PRODUIT, NO\_FOURNISSEUR FROM PRODUITS**  
**2 WHERE NO\_FOURNISSEUR <= 2;**

NOM_PRODUIT	NO_FOURNISSEUR
-----	-----
Chai	1
Chang	1
Aniseed Syrup	1
Chef Anton's Cajun Seasoning	2
...	

### Attention

Toutes les valeurs de colonnes de type « **VARCHAR2** » et « **CHAR** » sont traitées comme des chaînes de caractères lors de comparaisons. Par conséquent, les nombres stockés dans ce type de colonne sont comparés en tant que chaînes de caractères, et non en tant que nombres. Si la colonne est de type « **NUMBER** », alors 12 est supérieur à 9, si elle est de type caractère, 9 est supérieur à 12, car le caractère '9' est supérieur au caractère '1'.

La requête suivante est une sélection de la table PRODUITS pour extraire les noms du produit et la quantité des produits qui ont une quantité supérieure à '50'.



```
SQL> SELECT NOM_PRODUIT, QUANTITE FROM PRODUITS WHERE QUANTITE >'50' ;
```

NOM_PRODUIT	QUANTITE
Manjimup Dried Apples	50 cartons (300 g)
Thüringer Rostbratwurst	50 sacs x 30 saucisses
Dried Apples	50 paquets de 300 g
<b>Amandes</b>	<b>Paquet de 5 kg</b>
Dried Plums	Sac de 450 g

```
SQL> SELECT ASCII(5), ASCII('P') FROM DUAL;
```

ASCII(5)	ASCII('P')
53	80

Comme vous pouvez voir dans l'exemple précédent le produit 'Amandes' et 'Dried Plums' ont été trouvés car la comparaison est faite par ordre alphabétique, en occurrence dans l'ordre du code du caractère dans la page des codes utilisée.

La requête suivante est une sélection de la table CLIENTS pour extraire les sociétés qui ont un code inférieur à 'B'.



```
SQL> SELECT CODE_CLIENT, SOCIETE FROM CLIENTS WHERE CODE_CLIENT < 'B';
```

CODE_	SOCIETE
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn

## Différent de

L'opérateur logique **différent de** compare la valeur retournée par l'expression de gauche avec la valeur retournée par l'expression de droite ; si elles sont différentes, il retourne VRAI, sinon FAUX.

Etant donné que certains claviers ne disposent pas du point d'exclamation « ! » ou de l'accent circonflexe « ^ », Oracle prévoit trois formes différentes pour l'opérateur de **différent de** :

```
EXPRESSION1 { != | ^= | <> } EXPRESSION2
```

La requête suivante est une sélection de la table EMPLOYEES pour extraire le nom, le prénom et la fonction des employés qui ne sont pas des représentants.



```
SQL> SELECT NOM, PRENOM, FONCTION FROM EMPLOYEES  
2 WHERE FONCTION <>'Représentant(e)' ;
```

NOM	PRENOM	FONCTION
Devie	Thérèse	Assistante commerciale
Ragon	André	Chef des ventes
Pouetre	Camille-Hélène	Assistante commerciale
Leger	Pierre	Chef des ventes
Ziliox	Francoise	Assistante commerciale
Lampis	Gabrielle	Assistante commerciale
Belin	Chantal	Chef des ventes
...		

# L'opérateur LIKE

## LIKE

L'opérateur « **LIKE** » est très utile pour effectuer des recherches dans des chaînes alphanumériques.

Il utilise deux caractères spéciaux pour signifier le type de correspondance recherchée :

- un signe pourcentage « % », appelé **caractère générique**,
- et un caractère de soulignement « \_ », appelé **marqueur de position**.

Le **caractère générique** placé dans une chaîne remplace une chaîne quelconque de caractères d'une longueur de zéro à n caractères.

Le **marqueur de position** placé dans une chaîne remplace un caractère quelconque mais impose l'existence de ce caractère.

**EXPRESSION LIKE 'Chaîne de caractères avec des caractères spéciaux'**

La requête suivante est une sélection de la table PRODUITS pour extraire les noms du produit et la quantité des produits qui estiment leur quantité en boîtes et en kg.

```
SQL> SELECT NOM_PRODUIT, QUANTITE FROM PRODUITS
2 WHERE QUANTITE LIKE '%boîtes%kg%' ;
```

NOM_PRODUIT	QUANTITE
Konbu	1 boîtes (2 kg)
Alice Mutton	20 boîtes (1 kg)
Filo Mix	16 boîtes (2 kg)
Long Grain Rice	16 boîtes de 2 kg

Dans l'exemple précédent vous pouvez constater que les enregistrements extraits contiennent dans la chaîne de caractère QUANTITE deux chaînes de caractères la première 'boîtes' et la deuxième 'kg'.

## Attention


Les valeurs contenues dans les colonnes sont sensibles à la case (majuscule, minuscule), les informations saisies dans les chaînes de caractères de comparaison doivent l'être aussi.

La requête suivante est une sélection de la table PRODUITS pour extraire les quantités des produits qui dans la colonne QUANTITE ont un '0' en troisième position et en même temps le caractère 'g' en dernière position.

```
SQL> SELECT NOM_PRODUIT, QUANTITE FROM PRODUITS WHERE QUANTITE LIKE '__0%g';
```

NOM_PRODUIT	QUANTITE
Fruit Cocktail	430 g
Pears	430 g
Peaches	430 g
Pineapple	430 g
Cherry Pie Filling	430 g
Green Beans	400 g
Corn	400 g
Peas	400 g
Tuna Fish	140 g
Smoked Salmon	140 g


La requête suivante est une sélection de la table `PRODUITS` pour extraire les noms du produit et la quantité des produits qui dans la colonne `QUANTITE` commencent par trois caractères quelconques et finissant par 'pièces'.



```
SQL> SELECT NOM_PRODUIT, QUANTITE FROM PRODUITS
2 WHERE QUANTITE LIKE '___%pièces' ;
```

NOM_PRODUIT	QUANTITE
Teatime Chocolate Biscuits	10 boîtes x 12 pièces
Sir Rodney's Scones	24 cartons x 4 pièces
Perth Pasties	48 pièces
Escargots de Bourgogne	24 pièces
Wimmers gute Semmelknödel	20 sacs x 4 pièces
Scottish Longbreads	10 sacs x 8 pièces
Chocolate Biscuits Mix	10 boîtes de 12 pièces
Scones	24 paquets de 4 pièces


### Note



Il faut noter que les opérateurs logiques travaillent avec des expressions qui par définition sont des traitements sur des données de même type.

Ainsi, pour les chaînes de caractères, vous pouvez utiliser les opérateurs de concaténation ou toute autre fonction de traitement de chaîne.


Voici une requête qui permet de sélectionner les `CATEGORIES` dont le nom de la catégorie fait partie de la description.



```
SQL> SELECT NOM_CATEGORIE, DESCRIPTION FROM CATEGORIES
2 WHERE DESCRIPTION LIKE '%' || NOM_CATEGORIE || '%' ;
```

NOM_CATEGORIE	DESCRIPTION
Boissons	Boissons, cafés, thés, bières
Desserts	Desserts et friandises
Viandes	Viandes préparées
Viande en conserve	Viande en conserve

### Conseil



Lorsque vous voulez inclure le caractère « % » ou « \_ » parmi les caractères recherchés vous devez définir le caractère d'échappement.

Le caractère d'échappement vous permet de préciser à Oracle que les caractères « % » ou « \_ » ne doivent pas être interprétés comme des caractères spéciaux. Vous pouvez également rechercher le caractère d'échappement lui-même en le répétant.


La syntaxe de déclaration du caractère d'échappement est la suivante :

**EXPRESSION1 LIKE EXPRESSION2 ESCAPE 'caractère'**

**caractère**

Le caractère d'échappement.

La requête suivante est une sélection de la table `PRODUITS` pour extraire les noms du produit et la quantité des produits qui dans la colonne `QUANTITE` ont un caractère '%'.



```
SQL> SELECT NOM_PRODUIT, QUANTITE FROM PRODUITS
2 WHERE QUANTITE LIKE '%\%' ESCAPE '\';
```

NOM_PRODUIT	QUANTITE
-------------	----------

Chartreuse verte	1 bouteille (750 cc) 55%
Laughing Lumberjack Lager	24 bouteilles (12 onces) 3%
Outback Lager	24 bouteilles (355 ml) 5%
Lakkalikööri	1 bouteille (500 ml) 21%
Côte de Blaye	12 bouteilles (75 cl) 13%
Rhönbräu Klosterbier	24 bouteilles (0,5 litre) 7%

La requête suivante est une sélection de la table PRODUITS pour extraire les noms du produit et la quantité des produits qui dans la colonne QUANTITE ont un caractère '%' ainsi que 'ml)' en utilisant le caractère d'échappement ')'.



```
SQL> SELECT NOM_PRODUIT, QUANTITE FROM PRODUITS
2 WHERE QUANTITE LIKE '%ml))%)%' ESCAPE ')';
```

NOM_PRODUIT	QUANTITE
-----	-----
Outback Lager	24 bouteilles (355 ml) 5%
Lakkalikööri	1 bouteille (500 ml) 21%

Rappelez-vous que pour pouvoir rechercher le caractère d'échappement lui-même il faut le répéter.

## La fonction REGEXP\_LIKE

### REGEXP\_LIKE



La fonction « **REGEXP\_LIKE** » est très utile pour effectuer des recherches dans des chaînes alphanumériques correspondant à un certain motif défini par une expression rationnelle (expression régulière).

**REGEXP\_LIKE ( CHAÎNE1, CHAÎNE2 [, PARAMETRE ] ) ;**

<b>CHAÎNE1</b>	La chaîne à traiter.
<b>CHAÎNE2</b>	Une expression rationnelle qui permet de décrire un ensemble de chaînes. Les expressions rationnelles sont construites comme des opérations arithmétiques, en utilisant des opérateurs divers pour combiner des expressions plus petites.
<b>PARAMETRE</b>	Paramètre de correspondance de l'expression rationnelle. Il peut avoir les valeurs suivantes : <ul style="list-style-type: none"> <li><b>i</b> Ignorer les différences majuscules/minuscules dans le motif.</li> <li><b>c</b> Respecter les différences majuscules/minuscules dans le motif.</li> <li><b>x</b> Ignorer les espaces dans la chaîne à traiter.</li> </ul>

Une expression rationnelle correspondant à un caractère unique peut être suivie par l'un des opérateurs de répétition suivants :

<b>^</b>	Un méta-caractères correspondant respectivement à une chaîne vide au début de ligne.
<b>\$</b>	Un méta-caractères correspondant respectivement à une chaîne vide en fin de ligne.
<b>?</b>	L'élément précédent est facultatif et doit être mis en correspondance une fois au maximum.

<b>*</b>	L'élément précédent doit être mis en correspondance zéro ou plusieurs fois.
<b>.</b>	Un méta-caractère correspondant respectivement à un caractère quelconque mais impose l'existence de ce caractère.
<b>+</b>	L'élément précédent doit être mis en correspondance au moins une fois.
<b>{n}</b>	L'élément précédent doit être mis en correspondance exactement n fois.
<b>{n,}</b>	L'élément précédent doit être mis en correspondance n fois ou plus.
<b>{,m}</b>	L'élément précédent est facultatif et doit être mis en correspondance m fois au plus.
<b>{n,m}</b>	L'élément précédent doit être mis en correspondance au moins n fois, mais au plus m fois.
<b>[ ]</b>	Une liste de caractères, encadrée par « [ » et « ] » peut être mise en correspondance avec n'importe quel caractère unique appartenant à la liste.
<b>( )</b>	Une chaîne de caractères, encadrée par « ( » et « ) » peut être mise en correspondance avec la chaîne correspondante et mémorise la correspondance.
<b>\n</b>	La nième occurrence précédente (l'occurrence précédente est définie par 'n') de la chaîne de caractères, regroupé entre parenthèses « (...) », doit être mise en correspondance consécutive deux fois.
<b> </b>	Deux expressions rationnelles peuvent être reliées par l'opérateur «   » ; l'expression résultante correspondra à toute chaîne correspondant à l'une ou l'autre des deux sous-expressions.
<b>\</b>	Les méta-caractères « ? », « + », « { », «   », « ( », et « ) » perdent leurs significations spéciales si vous les préfixez par « \ ».
<b>[^]</b>	Une liste de caractères, encadrée par « [ » et « ] » et commençant par le caractère « ^ » peut être mise en correspondance avec n'importe quel caractère unique sauf les caractères appartenant à la liste.
<b>-</b>	La mise en correspondance des tous les caractères identiques ou compris entre le caractère à gauche et le caractère à droite (dans l'ordre de tri ASCII).

La requête suivante est une sélection de la table **PRODUITS** pour extraire toutes les lignes dont la colonne **QUANTITE** commence par la chaîne '10' et finit par la chaîne 'pièces'.



```
SQL> SELECT QUANTITE FROM PRODUITS
2 WHERE REGEXP_LIKE (QUANTITE, '^(10).*(pièces)$');

QUANTITE
-----
10 boîtes x 12 pièces
10 sacs x 8 pièces
10 boîtes de 12 pièces
```

La requête suivante est une sélection de la table PRODUITS pour extraire toutes les lignes dont la colonne QUANTITE contient n'importe quel caractère ' , ' ou 'x'.



```
SQL> SELECT QUANTITE FROM PRODUITS
2  WHERE REGEXP_LIKE (QUANTITE, '[ ,x]');
```

```
QUANTITE
-----
10 boîtes x 20 sacs
10 boîtes x 12 pièces
24 cartons x 4 pièces
24 boîtes x 2 tartes
50 sacs x 30 saucisses
20 sacs x 4 pièces
10 sacs x 8 pièces
24 bouteilles (0,5 litre) 7%
48 bocaux de 170 g
12 bocaux de 225 g
12 bocaux de 340 g
24 bocaux de 225 g
```

La requête suivante est une sélection de la table PRODUITS pour extraire toutes les lignes dont la colonne QUANTITE contient la chaîne 'sacs' et la chaîne 'pièces'.



```
SQL> SELECT QUANTITE FROM PRODUITS
2  WHERE REGEXP_LIKE (QUANTITE, '(sacs).*(pièces)');
```

```
QUANTITE
-----
20 sacs x 4 pièces
10 sacs x 8 pièces
```

La requête suivante est une sélection de la table PRODUITS pour extraire toutes les lignes dont la colonne QUANTITE contient la chaîne 'carton' ou la chaîne 'pièces' ou la chaîne 'bouteilles' et commence par le caractère '2'.



```
SQL> SELECT QUANTITE FROM PRODUITS
2  WHERE REGEXP_LIKE (QUANTITE, '^2.*(carton|pièces|bouteilles)');
```

```
QUANTITE
-----
24 bouteilles (1 litre)
24 cartons x 4 pièces
24 cartons (500 g)
24 cartons (200 g)
24 bouteilles (70 cl)
24 bouteilles (1 litre)
24 cartons (50 g)
24 cartons (250 g)
24 bouteilles (12 onces) 3%
24 bouteilles (355 ml) 5%
24 cartons (200 g)
24 bouteilles (250 ml)
20 cartons (2 kg)
24 cartons (250 g)
24 pièces
24 bouteilles (500 ml)
20 sacs x 4 pièces
```

24 bouteilles (0,5 litre) 7%  
 24 paquets de 4 pièces  
 24 bouteilles de 35 cl 4

La requête suivante est une sélection de la table PRODUITS pour extraire toutes les lignes dont la colonne QUANTITE contient deux occurrences de la chaîne '100'.



```
SQL> SELECT QUANTITE FROM PRODUITS
      2 WHERE REGEXP_LIKE (QUANTITE, '(100.*){2}');
```

QUANTITE

-----  
 100 pièces (100 g)

La requête suivante est une sélection de la table PRODUITS pour extraire toutes les lignes dont la colonne QUANTITE contient deux occurrences successives du caractère 'l' et deux occurrences successives du numéro '0'.



```
SQL> SELECT QUANTITE FROM PRODUITS
      2 WHERE REGEXP_LIKE (QUANTITE, 'l{2,}.*0{2}');
```

QUANTITE

-----  
 1 bouteille (500 ml) 21%  
 24 bouteilles (500 ml)

La requête suivante est une sélection de la table PRODUITS pour extraire toutes les lignes dont la colonne QUANTITE ne commence pas par un des numéros de la liste.



```
SQL> SELECT QUANTITE FROM PRODUITS WHERE REGEXP_LIKE (QUANTITE, '^[^1-3]');
```

QUANTITE

-----  
 48 pots (6 onces)  
 4 boîtes (250 g)  
 50 cartons (300 g)  
 48 pièces  
 40 cartons (100 g)  
 50 sacs x 30 saucisses  
 48 tartes

La requête suivante est une sélection de la table FOURNISSEURS pour extraire toutes les lignes dont la colonne SOCIETE contient la chaîne 's.r.l.'. En utilisant le paramètre de correspondance de l'expression rationnelle, il est possible d'ignorer les différences majuscules/minuscules dans le motif.



```
SQL> SELECT SOCIETE FROM FOURNISSEURS
      2 WHERE REGEXP_LIKE ( SOCIETE, '(s[.]r[.]l)');
```

SOCIETE

-----  
 Pasta Buttini s.r.l.

```
SQL> SELECT SOCIETE FROM FOURNISSEURS
      2 WHERE REGEXP_LIKE ( SOCIETE, '(s[.]r[.]l)', 'i');
```

SOCIETE

-----  
 Formaggi Fortini S.R.L.  
 Pasta Buttini s.r.l.

L'exemple suivant recherche toutes les sociétés clientes qui ont une voyelle en double dans leur nom.





```
SQL> SELECT SOCIETE FROM CLIENTS
2  WHERE REGEXP_LIKE(SOCIETE, '([AEIOU])\1', 'i');

SOCIETE
-----
Great Lakes Food Market
Queen Cozinha
Split Rail Beer Ale
The Big Cheese
Vins et alcools Chevalier
Blauer See Delikatessen
```

La requête suivante est une sélection de la table PRODUITS pour extraire toutes les lignes dont la colonne QUANTITE contient deux occurrences de la chaîne '100'.



```
SQL> SELECT QUANTITE FROM PRODUITS
2  WHERE REGEXP_LIKE (QUANTITE, '(100).*\1');

QUANTITE
-----
100 pièces (100 g)
```

La requête suivante est une sélection de la table CLIENTS pour extraire toutes les lignes dont la colonne ADRESSE contient deux occurrences consécutives d'un caractère de type alpha numérique.



```
SQL> SELECT ADRESSE FROM CLIENTS
2  WHERE REGEXP_LIKE(ADRESSE, '([0-9a-z])\1', 'i');

ADRESSE
-----
Kirchgasse 6
184, chaussée de Tournai
Romero, 33
Av. Brasil, 442
Carrera 22 con Ave. Carlos Soublette #8-35
1 rue Alsace-Lorraine
1900 Oak St.
12 Orchestra Terrace
Carrera 52 con Ave. Bolívar #65-98 Llano Largo
...
```

La requête suivante recherche la même colonne dans la même table mais pour deux fois deux occurrences consécutives d'un caractère alpha numérique dans la colonne ADRESSE.



```
SQL> SELECT ADRESSE FROM CLIENTS
2  WHERE REGEXP_LIKE '([0-9a-z])\1.*([0-9a-z])\2', 'i');

Carrera 22 con Ave. Carlos Soublette #8-35
Carrera 52 con Ave. Bolívar #65-98 Llano Largo
Heerstr. 22
South House300 Queensbridge
55 Grizzly Peak Rd.
Adenauerallee 900
Avda. de la Constitución 2222
Cerrito 333
Sierras de Granada 9993
```

La requête suivante recherche la même colonne dans la même table mais pour trois fois deux occurrences consécutives d'un caractère alpha numérique dans la colonne ADRESSE.



```
SQL> SELECT ADRESSE FROM CLIENTS WHERE REGEXP_LIKE(ADRESSE,
2 '([0-9a-z])\1.*([0-9a-z])\2.*([0-9a-z])\3','i');
```

↑    ①    ↑
↑    ②    ↑
↑    ③    ↑

ADRESSE

-----

Carrera 22 con Ave. Carlos Soublette #8-35

Adenauerallee 900

Afin de permettre la construction d'expressions encore plus complexes, Oracle supporte les classes de caractères. Grâce à cette classe de caractères, il va être possible d'écrire des expressions régulières extrêmement précises afin de retrouver seulement l'information voulue.

Les classes de caractères sont:

<b>[ :alpha: ]</b>	caractère alphabétique.
<b>[ :alphanum: ]</b>	caractère alpha numérique.
<b>[ :lower: ]</b>	caractère alphabétique en minuscule.
<b>[ :upper: ]</b>	caractère alphabétique en majuscule.
<b>[ :digit: ]</b>	numéro.
<b>[ :xdigit: ]</b>	tous les caractères permis en hexadécimal.
<b>[ :space: ]</b>	espace.
<b>[ :punct: ]</b>	caractère de ponctuation.
<b>[ :cntrl: ]</b>	caractère de contrôle non imprimable.
<b>[ :print: ]</b>	caractère imprimable.
<b>[ :blank: ]</b>	caractère espace ou tabulation.
<b>[ =caractère= ]</b>	caractère qui correspond à tous les caractères d'équivalence dans la même classe de caractères. Par exemple le caractère 'e' est équivalent à : 'é', 'è' et 'e'.
<b>\d</b>	<b>[[:digit:]]</b>
<b>\D</b>	<b>[^[:digit:]]</b>
<b>\w</b>	<b>[[:alphanum:]]_</b>
<b>\W</b>	<b>[^[:alphanum:]]_</b>
<b>\s</b>	<b>[[:space:]]</b>
<b>\S</b>	<b>[^[:space:]]</b>

Ces différentes classes de caractères permettent de couvrir l'ensemble des caractères présents dans la table ASCII.

La requête suivante est une sélection de la table EMPLOYES pour extraire les employés qui ont un prénom qui contient le caractère 'e' suivi d'un ou deux caractères 'l' et suivi du caractère 'e' sans tenir compte des caractères accentués.



```
SQL> SELECT PRENOM FROM EMPLOYES
2 WHERE REGEXP_LIKE( PRENOM, '[[=e=]]l+[[=e=]]');
```

PRENOM

-----

Camille-Hélène

Gabrielle

Axelle

Isabelle

La requête suivante est une sélection de la table `PRODUITS` pour extraire toutes les lignes dont la colonne `QUANTITE` est formatée de la sorte : le début commence par deux ou trois occurrences d'un caractère numérique, ensuite un espace suivi d'un mot de trois à six caractères, un espace ensuite, et une parenthèse suivie de un à trois occurrences d'un caractère numérique, un espace, un caractère facultatif et finalement le caractère 'g' et une parenthèse.



```
SQL> SELECT QUANTITE FROM PRODUITS
2  WHERE REGEXP_LIKE(QUANTITE, '^[[[:digit:]]{2,3}[[[:space:]]
3      [[[:alpha:]]{3,6}[[[:space:]][\(\)]]{1,3}
4      [[[:space:]][[[:alpha:]]?(g[\)])]$', 'x');
```

QUANTITE

```
-----
12 pots (200 g)
20 boîtes (1 kg)
20 verres (450 g)
24 pots (250 g)
16 boîtes (500 g)
16 boîtes (2 kg)
15 unités (300 g)
24 pots (150 g)
32 boîtes (500 g)
100 sacs (250 g)
100 pièces (100 g)
10 verres (200 g)
15 pots (625 g)
```

```
SQL> SELECT QUANTITE FROM PRODUITS
2  WHERE REGEXP_LIKE(QUANTITE,
3  '^\\d{2,3}\\s\\w{3,6}\\s[\\(\\)\\d{1,3}\\s\\w?(g[\\)])]$', 'x');
```

QUANTITE

```
-----
12 pots (200 g)
20 boîtes (1 kg)
20 verres (450 g)
24 pots (250 g)
16 boîtes (500 g)
16 boîtes (2 kg)
15 unités (300 g)
24 pots (150 g)
32 boîtes (500 g)
100 sacs (250 g)
100 pièces (100 g)
10 verres (200 g)
15 pots (625 g)
```

## Le traitement de la case

Les opérateurs logiques de comparaison qui utilisent des expressions de types caractères sont soumis par défaut à tenir compte de majuscules/minuscules et des accents. Il est possible de s'affranchir de la sensibilité majuscules/minuscules ou des accents par la configuration de notre session. Les valeurs combinées des paramètres « `NLS_COMP` » et « `NLS_SORT` » déterminent les règles selon

lesquelles les caractères sont triés et comparés. Pour modifier les paramètres de sa propre session, vous pouvez utiliser la syntaxe suivante :

**ALTER SESSION SET paramètre = valeur ;**

Le paramètre « **NLS\_COMP** » détermine la façon dont « **NLS\_SORT** » est interprété par les instructions **SQL**. Il peut avoir deux valeurs :

<b>BINARY</b>	Tous les tris et les comparaisons sont effectués suivant les valeurs binaires des caractères de la chaîne, quelque soit la valeur fixée pour « <b>NLS_SORT</b> ». C'est le paramètre par défaut.
<b>LINGUISTIC</b>	Tous les tris et les comparaisons sont effectués suivant les règles linguistiques spécifiées par le paramètre « <b>NLS_SORT</b> ».

Le paramètre « **NLS\_SORT** » peut avoir plusieurs valeurs :

<b>BINARY</b>	Tous les tris et les comparaisons sont effectués suivant les valeurs binaires des caractères de la chaîne.
<b>langue</b>	Tous les tris et les comparaisons sont effectués suivant les règles linguistiques spécifiques à chaque langue. Vous pouvez utiliser : « <b>FRENCH</b> », « <b>GERMAN</b> », « <b>SLOVAK</b> », « <b>SPANISH</b> », « <b>SWISS</b> », « <b>UNICODE_BINARY</b> », « <b>WEST_EUROPEAN</b> », etc. Par défaut, le paramètre est initialisé par les options régionales du poste client.

La valeur du paramètre, qu'il s'agisse de « **BINARY** » ou « **langue** », peut comporter deux suffixes :

<b>_CI</b>	Tous les tris et les comparaisons sont effectués sans tenir compte de la sensibilité majuscules/minuscules.
<b>_AI</b>	Tous les tris et les comparaisons sont effectués sans tenir compte de la sensibilité majuscules/minuscules et sans la sensibilité aux accents.

Pour visualiser les deux paramètres « **NLS\_COMP** » et « **NLS\_SORT** », vous pouvez interroger la vue « **V\$NLS\_PARAMETERS** ».



```
SQL> SELECT * FROM V$NLS_PARAMETERS WHERE PARAMETER like 'NLS_O_';
```

PARAMETER	VALUE
NLS_SORT	BINARY_AI
NLS_COMP	LINGUISTIC

La requête suivante est une sélection de la table **EMPLOYES** pour extraire les employés qui ont un prénom qui commence par le caractère 'R' ; les enregistrements sont triés sur le prénom. Vous pouvez remarquer que l'ordre de tri suivant les valeurs binaires des caractères de la chaîne ne correspond pas à la langue française.



```
SQL> ALTER SESSION SET NLS_COMP=BINARY NLS_SORT=BINARY;
```

```
SQL> SELECT PRENOM FROM EMPLOYES WHERE PRENOM LIKE 'R%' ORDER BY 1 ;
```

PRENOM
René
Renée
Robert
Régis

```
SQL> ALTER SESSION SET NLS_SORT=FRENCH;

SQL> SELECT PRENOM FROM EMPLOYES WHERE PRENOM LIKE 'R%' ORDER BY 1 ;

PRENOM
-----
Régis
René
Renée
Robert
```

La requête suivante est une sélection de la table EMPLOYES pour extraire les employés qui ont un prénom qui contient le caractère 'e' et commence par 'r'.



```
SQL> ALTER SESSION SET NLS_COMP=LINGUISTIC NLS_SORT=BINARY;

SQL> SELECT PRENOM FROM EMPLOYES
2  WHERE PRENOM LIKE 'r%e%' ORDER BY PRENOM;

aucune ligne sélectionnée

SQL> ALTER SESSION SET NLS_SORT=BINARY_CI;

SQL> SELECT PRENOM FROM EMPLOYES
2  WHERE PRENOM LIKE 'r%e%' ORDER BY PRENOM;

PRENOM
-----
René
Renée
Robert

SQL> ALTER SESSION SET NLS_SORT=BINARY_AI;

SQL> SELECT PRENOM FROM EMPLOYES
2  WHERE PRENOM LIKE 'r%e%' ORDER BY PRENOM;

PRENOM
-----
Régis
René
Renée
Robert
```

## L'opérateur BETWEEN

---

Il existe également des opérateurs logiques qui permettent d'effectuer des comparaisons avec des listes de valeurs, comme décrit dans la présentation.

### BETWEEN

L'opérateur logique « **BETWEEN** » vérifie si la valeur retournée par **EXPRESSION1** est égale à **EXPRESSION2**, **EXPRESSION3** ou toute valeur comprise entre **EXPRESSION2** et **EXPRESSION3** ; alors retourne **VRAI** sinon **FAUX**.

**EXPRESSION1 BETWEEN EXPRESSION2 AND EXPRESSION3**

La requête suivante est une sélection de la table EMPLOYES pour extraire le nom, le prénom et le salaire des employés qui ont un salaire compris entre 2500 et 3500.



```
SQL> SELECT NOM, PRENOM, SALAIRE FROM EMPLOYES
2 WHERE SALAIRE BETWEEN 1500 AND 2000;
```

NOM	PRENOM	SALAIRE
Devie	Thérèse	1540
Pouetre	Camille-Hélène	2000
Ziliox	Françoise	1900
Poupard	Claudette	1800
Etienne	Brigitte	2000
Grangirard	Patricia	1700
Guerdon	Béatrice	1700

La requête suivante est une sélection de la table EMPLOYES pour extraire le nom, le prénom et la date de naissance des employés qui sont nés en 1985.



```
SQL> SELECT NOM, PRENOM, DATE_NAISSANCE FROM EMPLOYES
2 WHERE DATE_NAISSANCE BETWEEN '01/01/1985' AND '31/12/1985';
```

NOM	PRENOM	DATE_NAISS
Rollet	Philippe	09/03/1985
Coutou	Myriam	31/08/1985
Chaussende	Maurice	31/01/1985
Dupouey	Pierre	13/11/1985

La requête suivante est une sélection de la table CLIENTS pour extraire les sociétés entre 'GALED' et 'HANAR'.



```
SQL> SELECT CODE_CLIENT, SOCIETE, VILLE FROM CLIENTS
2 WHERE CODE_CLIENT BETWEEN 'GALED' AND 'HANAR';
```

CODE_	SOCIETE	VILLE
GALED	Galería del gastrónomo	Barcelona
GODOS	Godos Cocina Típica	Sevilla
GOURL	Gourmet Lanchonetes	Campinas
GREAL	Great Lakes Food Market	Eugene
GROSR	GROSELLA-Restaurante	Caracas
HANAR	Hanari Carnes	Rio de Janeiro

## L'opérateur IN

Il existe également des opérateurs logiques qui permettent d'effectuer des comparaisons avec des listes de valeurs, comme décrit dans la présentation.

### IN

L'opérateur logique « **IN** » vérifie si la valeur retournée par EXPRESSION1 est dans la LISTE\_DE\_VALEURS ; alors il retourne VRAI, sinon FAUX.

**EXPRESSION1 IN (LISTE\_DE\_VALEURS)**

### LISTE\_DE\_VALEURS

La liste des valeurs peut être une liste de constantes ou une liste de valeurs dynamiques (une sous-requête ; le traitement des sous-requêtes est présenté plus loin dans ce module) ; cependant les types de données des différentes constantes doivent être identiques au type retourné par `EXPRESSION1`.

La requête suivante est une sélection de la table `CLIENTS` pour extraire la société et la ville de résidence des clients situés à Paris, Strasbourg et Toulouse.



```
SQL> SELECT SOCIETE, VILLE FROM CLIENTS
2 WHERE VILLE IN ('Paris','Strasbourg','Toulouse') ;
```

SOCIETE	VILLE
La maison d'Asie	Toulouse
Blondel père et fils	Strasbourg
Spécialités du monde	Paris
Paris spécialités	Paris

La requête suivante est une sélection de la table `EMPLOYES` pour extraire le nom, le prénom et la date de naissance des managers pour les employés qui sont nés en 1985. (La requête utilise une sous-requête pour plus d'informations voir le module concernant les sous-requêtes)



```
SQL> SELECT NOM, PRENOM, DATE_NAISSANCE FROM EMPLOYES
2 WHERE NO_EMPLOYE IN (SELECT RECD_COMPTE FROM EMPLOYES
3 WHERE DATE_NAISSANCE BETWEEN '01/01/1985' AND '31/12/1985' );
```

NOM	PRENOM	DATE_NAISS
Buchanan	Steven	26/11/1972
Splingart	Lydia	11/09/1978
Ragon	André	21/05/1989

## Les opérateurs logiques

Les opérateurs logiques présents permettent de comparer des expressions qui retournent une valeur unique. Tous ces opérateurs sont utilisés de façon analogue.

### ANY

L'opérateur logique « **ANY** » compare la valeur retournée par `EXPRESSION1` à chaque donnée de la `LISTE_DE_VALEURS`.

L'opérateur « **=ANY** » équivaut à `IN`. L'opérateur « **<ANY** » signifie « inférieur à au moins une des valeurs » donc « inférieur au maximum ». L'opérateur « **>ANY** » signifie « supérieur à au moins une des valeurs » donc « supérieur au minimum ».



```
SQL> SELECT NO_EMPLOYE, NOM FROM EMPLOYES WHERE NO_EMPLOYE < ANY (2, 4, 5);
```

NO_EMPLOYE	NOM
1	Besse
2	Destenay
3	Letertre
4	Kremser

```
SQL> SELECT NO_EMPLOYE, NOM FROM EMPLOYES WHERE NO_EMPLOYE < 5;
```

```

NO_EMPLOYE NOM
-----
      1 Besse
      2 Destenay
      3 Letertre
      4 Kremser

SQL> SELECT NO_EMPLOYE, NOM FROM EMPLOYES
      2 WHERE NO_EMPLOYE > ANY (90, 98, 107);

```

```

NO_EMPLOYE NOM
-----
     91 Aubert
     92 Thomas
     93 Falatik
     94 Marielle
     95 Leger
     96 Marchand
     97 Seigne
     98 Burst
     99 Damas
    100 Espeche
    101 Charles
    102 Thimoleon
    103 Arrambide
    104 Ziliox
    105 Viry
    106 Cheutin
    107 Lombard
    108 Tourtel
    109 Lampis
    110 Regner
    111 Teixeira

```

## ALL

L'opérateur logique « **ALL** » compare la valeur retournée par EXPRESSION1 à toutes les données de la LISTE\_DE\_VALEURS.

L'opérateur « **<ANY** » signifie « inférieur au minimum » et « **>ALL** » signifie « supérieur au maximum ».



```
SQL> SELECT NO_EMPLOYE, NOM FROM EMPLOYES WHERE NO_EMPLOYE < ALL (2, 4, 5);
```

```

NO_EMPLOYE NOM
-----
      1 Besse

```

```
SQL> SELECT NO_EMPLOYE, NOM FROM EMPLOYES WHERE NO_EMPLOYE > ALL (2,4,107);
```

```

NO_EMPLOYE NOM
-----
    108 Tourtel
    109 Lampis
    110 Regner
    111 Teixeira

```





## Conseil

Les deux opérateurs logiques « **ALL** » et « **ANY** » permettent de comparer des expressions avec des listes de valeurs.

Il est préférable d'utiliser ces deux opérateurs avec des sous-requêtes dynamiques qu'avec les listes de valeurs statiques (voir sous-requêtes).

## Les expressions logiques

Les opérateurs logiques forment des expressions de type logique et ces expressions peuvent être combinées à l'aide des opérateurs logiques « **AND** », « **OR** » ou « **NOT** ».

### AND

L'opérateur logique « **AND** » vérifie si **EXPRESSION1** et **EXPRESSION2** sont **VRAI** en même temps ; alors il retourne **VRAI**, sinon **FAUX**.

#### **EXPRESSION1 AND EXPRESSION2**

La requête suivante est une sélection de la table **PRODUIT** pour extraire les produits qui ont un stock supérieur à 100 et qui sont de type boîte.



```
SQL> SELECT NOM_PRODUIT, QUANTITE, UNITES_STOCK FROM PRODUITS
2 WHERE UNITES_STOCK > 100 AND QUANTITE LIKE '%boîte%' ;
```

NOM_PRODUIT	QUANTITE	UNITES_STOCK
Boston Crab Meat	24 boîtes (4 onces)	123
Pâté chinois	24 boîtes x 2 tartes	115
Crab Meat	24 boîtes de 115 g	120
Green Tea	20 sacs par boîte	125

La requête suivante est une sélection de la table **COMMANDES** pour extraire les commandes du client 'COMMI' passées par les employés '6', '7', '8' et dont la date d'envoi n'a pas été saisie.



```
SQL> SELECT NO_COMMANDE, CODE_CLIENT, NO_EMPLOYE, DATE_ENVOI FROM COMMANDES
2 WHERE CODE_CLIENT = 'COMMI' AND NO_EMPLOYE BETWEEN 6 AND 8
3 AND DATE_ENVOI IS NULL;
```

NO_COMMANDE	CODE_	NO_EMPLOYE	DATE_ENVOI
227851	COMMI	7	
225729	COMMI	7	
224727	COMMI	7	
226905	COMMI	7	
227903	COMMI	7	
227821	COMMI	7	

La requête suivante est une sélection de la table **PRODUITS** pour extraire les produits du fournisseur numéro '1' et du fournisseur numéro '2'.



```
SQL> SELECT NOM_PRODUIT, CODE_CATEGORIE FROM PRODUITS
2 WHERE NO_FOURNISSEUR = 1 AND NO_FOURNISSEUR = 5;
```

aucune ligne sélectionnée



## Attention

Il faut faire très attention aux abus de langage comme dans l'exemple précédent où l'on souhaite afficher les produits du fournisseur numéro '1' ou du fournisseur numéro '2'.

L'ensemble des conditions de la clause « **WHERE** » est exécuté pour valider chaque enregistrement de la table. Par conséquent le numéro fournisseur ne peut pas être '1' et '2' à la fois et pour le même enregistrement.

## OR

L'opérateur logique « **OR** » vérifie si au moins une des deux est VRAI ; alors il retourne VRAI, sinon FAUX.

### EXPRESSION1 OR EXPRESSION2

La requête suivante est une sélection de la table PRODUITS pour extraire les produits qui n'ont pas de quantité ou des unités en stock.



```
SQL> SELECT NOM_PRODUIT, QUANTITE, UNITES_STOCK FROM PRODUITS
3 WHERE QUANTITE IS NULL OR UNITES_STOCK IS NULL;
```

NOM_PRODUIT	QUANTITE	UNITES_STOCK
Mishi Kobe Niku	18 cartons (500 g)	
Alice Mutton	20 boîtes (1 kg)	
Rössle Sauerkraut	25 canettes (825 g)	
Singaporean Hokkien Fried Mee	32 cartons (1 kg)	
Perth Pasties	48 pièces	
Guaraná Fantástica	22 canettes (350 ml)	
Thüringer Rostbratwurst	50 sacs x 30 saucisses	
Granola		100
Potato Chips		200
Hot Cereal		60

La requête suivante est une sélection de la table PRODUITS pour extraire les produits du fournisseur numéro '1' et du fournisseur numéro '5'.



```
SQL> SELECT NOM_PRODUIT, CODE_CATEGORIE FROM PRODUITS
3 WHERE NO_FOURNISSEUR = 1 OR NO_FOURNISSEUR = 5;
```

NOM_PRODUIT	CODE_CATEGORIE
Chai	1
Chang	1
Aniseed Syrup	2
Queso Cabrales	4
Queso Manchego La Pastora	4
Walnuts	7
Green Beans	9
Peas	9

## Conseil

L'opérateur logique « **AND** » est prioritaire par rapport à l'opérateur « **OR** ». Des parenthèses peuvent être utilisées pour imposer une priorité dans l'évaluation de l'expression, ou tout simplement pour rendre l'expression plus claire.



Dans l'exemple suivant on recherche les fournisseurs qui n'ont pas de fax renseigné et qui habitent en Allemagne ou en France.



```
SQL> SELECT SOCIETE, PAYS, FAX FROM FOURNISSEURS
2 WHERE FAX IS NULL AND PAYS = 'Allemagne' OR PAYS = 'France';
```

SOCIETE	PAYS	FAX
Heli Süßwaren GmbH Co. KG	Allemagne	
Plutzer Lebensmittelgroßmärkte AG	Allemagne	
Aux joyeux ecclésiastiques	France	01.03.83.00.62
Escargots Nouveaux	France	
Gai pâturage	France	04.38.76.98.58

```
SQL> SELECT SOCIETE, PAYS, FAX FROM FOURNISSEURS
2 WHERE FAX IS NULL AND ( PAYS = 'Allemagne' OR PAYS = 'France');
```

SOCIETE	PAYS	FAX
Heli Süßwaren GmbH Co. KG	Allemagne	
Plutzer Lebensmittelgroßmärkte AG	Allemagne	
Escargots Nouveaux	France	

### NOT

L'opérateur logique « **NOT** » inverse le sens de EXPRESSION, explicitement si EXPRESSION est FAUX ; alors retourne il VRAI, sinon FAUX.

#### NOT EXPRESSION

La requête suivante est une sélection de la table EMPLOYES pour extraire le nom et la fonction des employés qui ont un salaire supérieur à 2500 et qui ne sont pas des représentants.



```
SQL> SELECT NOM, FONCTION, SALAIRE FROM EMPLOYES
2 WHERE SALAIRE > 2500 AND NOT FONCTION LIKE 'Rep%';
```

NOM	FONCTION	SALAIRE
Ragon	Chef des ventes	13000
Leger	Chef des ventes	19000
Belin	Chef des ventes	10000
Fuller	Vice-Président	96000
Brasseur	Vice-Président	147000
Splingart	Chef des ventes	16000
Buchanan	Chef des ventes	13000
Chambaud	Chef des ventes	12000
Giroux	Président	150000

### NOT IN

L'opérateur logique « **NOT IN** » vérifie si la valeur retournée par EXPRESSION1 n'est pas dans la LISTE\_DE\_VALEURS ; alors il retourne VRAI, sinon FAUX.

#### EXPRESSION1 NOT IN (LISTE\_DE\_VALEURS)

```
SQL> SELECT NOM,PRENOM, TITRE FROM EMPLOYES
2 WHERE TITRE NOT IN ('M.', 'Mme');
```

NOM	PRENOM	TITRE



Davolio	Nancy	Mlle
Callahan	Laura	Mlle
Leverling	Janet	Mlle
Dodsworth	Anne	Mlle


La requête précédente est une sélection de la table EMPLOYES pour extraire le nom, prénom et titre des employés avec une valeur pour la colonne titre autre que 'M.' et 'Mme'.

## NOT BETWEEN

L'opérateur logique « **BETWEEN** » vérifie si la valeur retournée par EXPRESSION1 n'est pas égale à EXPRESSION2, EXPRESSION3 ou toute valeur comprise entre EXPRESSION2 et EXPRESSION3 ; alors retourne VRAI sinon FAUX.

EXPRESSION1 NOT BETWEEN EXPRESSION2 AND EXPRESSION3

La requête suivante est une sélection de la table EMPLOYES pour extraire le nom, le prénom et la date d'embauche des employés qui n'ont pas été recrutés entre '01/01/1990' et '31/07/2003'.



```
SQL> SELECT NOM, PRENOM, DATE_EмбаУCHE FROM EMPLOYES
2 WHERE DATE_EмбаУCHE NOT BETWEEN '01/01/1990' AND '31/07/2003'
3 ORDER BY DATE_EмбаУCHE;
```


NOM	PRENOM	DATE_EмбаУ
-----	-----	-----
Malejac	Yannick	29/11/1987
Alvarez	Marcel	03/08/1988
Burst	Jean-Yves	23/10/1988
Peacock	Margaret	16/04/1989
Maurer	Véronique	27/05/1989
Fuller	Andrew	09/08/2003
Chaussende	Maurice	10/08/2003
Cremel	Brigitte	28/08/2003
Regner	Charles	07/09/2003

## IS NOT NULL

L'opérateur logique « **IS NOT NULL** » vérifie si la valeur retournée par EXPRESSION n'est pas égale à « **NULL** » ; alors retourne il VRAI , sinon FAUX.

EXPRESSION IS NOT NULL

La requête suivante est une sélection de la table EMPLOYES pour extraire les noms et prénoms des employés qui ont une commission renseignée et un salaire supérieur a 10 000.



```
SQL> SELECT NOM, PRENOM, SALAIRE, COMMISSION FROM EMPLOYES
2 WHERE COMMISSION IS NOT NULL AND SALAIRE > 10000;
```

NOM	PRENOM	SALAIRE	COMMISSION
-----	-----	-----	-----
Ragon	André	13000	5980
Leger	Pierre	19000	11150
Splingart	Lydia	16000	16480
Buchanan	Steven	13000	12940
Chambaud	Axelle	12000	11600

## NOT LIKE

L'opérateur logique « **NOT LIKE** » vérifie que la valeur retournée par EXPRESSION1 ne correspond pas au modèle défini dans EXPRESSION2 .

EXPRESSION1 NOT LIKE EXPRESSION2

Voici une requête qui permet de sélectionner les CATEGORIES dont le nom de la catégorie n'est pas une partie de la description.



```
SQL> SELECT NOM_CATEGORIE, DESCRIPTION FROM CATEGORIES  
2 WHERE DESCRIPTION NOT LIKE '%' || NOM_CATEGORIE || '%' ;
```

NOM_CATEGORIE	DESCRIPTION
Condiments	Sauces, assaisonnements et épices
Produits laitiers	Fromages
Pâtes et céréales	Pains, biscuits, pâtes et céréales
Produits secs	Fruits secs, raisins, autres
Poissons et fruits de mer	Poissons, fruits de mer, escargots
Conserves	Fruits, légumes en conserve et confitures

# 4

- *Formatage de chaînes*
- *Recherche*
- *Les zones horaires*
- *TIMESTAMP*
- *INTERVAL*

## Les types de données



### Objectifs

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- Manipuler des chaînes de caractères.
- Manipuler les expressions de type date.
- Extraire une partie d'une chaîne de caractères.
- Rechercher et remplacer des sous chaînes.
- Changer les paramètres de zone horaire pour la session.
- Effectuer des arrondis et manipuler le signe des données.



### Contenu

Types chaîne de caractères	4-2	Fonctions d'arrondis	4-22
Majuscules / Minuscules	4-3	Les zones horaires	4-24
Manipulation de chaînes	4-5	Types date	4-26
Extraire une sous chaîne	4-7	Les conversions	4-29
Recherche dans la chaîne	4-9	Les dates système	4-30
Rechercher et remplacer	4-12	Types intervalle	4-31
Traduction de chaînes	4-14	Manipulation des dates	4-33
Types numériques	4-16	Retrouver une date	4-34
Fonctions de contrôle	4-18	Arrondis des dates	4-35
Fonctions de calcul	4-20		

# Types chaîne de caractères

---

Le module précédent explique la syntaxe du langage **SQL**, plus précisément les modalités d'écriture d'une requête qui extrait à partir d'une table les informations que vous souhaitez.

Une requête extrait de la base une liste de colonnes, d'expressions et/ou de constantes.

Une expression est un ensemble d'une ou plusieurs colonnes, constantes et/ou fonctions combinées au moyen des opérateurs. La présence d'expressions dans les requêtes augmente les possibilités offertes pour les traitements des informations extraites et enrichit celles de conditions restrictives. Dans l'expression, on peut utiliser des fonctions.

Les types SQL sont regroupés dans quatre grandes familles : chaînes de caractères, temporels, numériques et binaires.

Les fonctions SQL sont utilisées pour effectuer les traitements suivants :

- Manipulation des chaînes des caractères
- Calcul arithmétique
- Manipulation de dates
- Conversion et transformation

SQL permet de codifier les chaînes de caractères dans des formats où chaque caractère s'exprime sur 2 octets (ASCII, EBCDIC) ou sur 4 octets (Unicode).

ASCII est un format d'encodage de caractères sur 8 bits, soit 1 octet. Étant donné que certaines langues admettent des caractères diacritiques (accent, cédille, tilde, ligature, etc.) il a fallu inventer un stratagème qui permet de charger un panel de signes différents en fonction de la langue de l'utilisateur.

À la création de la base de données vous spécifiez le jeu de caractères utilisé par la base de données pour stocker les données. Vous spécifiez également le jeu de caractères national utilisé pour stocker les données dans des colonnes définies spécifiquement avec les types « **NCHAR** », « **NCLOB** » ou « **NVARCHAR2** ».

Les types de données chaîne de caractères disponibles sont :

## **VARCHAR2**

Chaîne de caractères de longueur variable comprenant au maximum 4000 bytes.

## **CHAR**

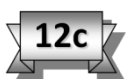
Chaîne de caractères de longueur fixe avec L comprenant au maximum 2000 bytes.

## **NCHAR**

Chaîne de caractères de longueur fixe pour des jeux de caractères multi octets pouvant atteindre 4000 bytes selon le jeu de caractères national.

## **NVARCHAR2**

Chaîne de caractères de longueur variable pour des jeux de caractères multi octets pouvant atteindre 4000 bytes selon le jeu de caractères national.



A partir de la version Oracle 12c, il est possible d'avoir une taille étendue pour les variables de type « **VARCHAR2** », « **NVARCHAR2** », ainsi vous pouvez stocker dans une variable de ce type jusqu'à 32767 bytes.

## **CLOB**

Character Large Object (grand objet caractère), chaîne de caractères avec une longueur maximale d'enregistrement pouvant atteindre ( 4Gb – 1 ) \* (la taille du block du tablespace).

## **NCLOB**

Type de donnée « **CLOB** » pour des jeux de caractères multi octets avec une longueur maximale d'enregistrement pouvant atteindre ( 4Gb – 1 ) \* (la taille du block du tablespace).

## LENGTH

La fonction « **LENGTH** » renvoie la longueur, en nombre des caractères, de la chaîne.



```
SQL> SELECT LENGTH('Chaine') FROM DUAL;
```

```
LENGTH( 'CHAINE' )
-----
                6
```

## VSIZE

La fonction « **VSIZE** » renvoie la longueur, en bytes, de la chaîne.



```
SQL> DESC CLIENTS
```

```
Nom                                NULL ?    Type
-----
CODE_CLIENT                        NOT NULL  CHAR(5)
SOCIETE                            NOT NULL  NVARCHAR2(40)
...
```

```
SQL> SELECT VSIZE(CODE_CLIENT) "TailleCODE_CLIENT",
2         LENGTH(CODE_CLIENT) "LongueurCODE_CLIENT" ,
3         VSIZE(SOCIETE) "TailleSOCIETE",
4         LENGTH(SOCIETE) "LongueurSOCIETE"
5 FROM CLIENTS WHERE CODE_CLIENT < 'BLONP';
```

TailleCODE_CLIENT	LongueurCODE_CLIENT	TailleSOCIETE	LongueurSOCIETE
5	5	38	19
5	5	68	34
5	5	46	23
5	5	30	15
5	5	36	18
5	5	46	23

## Majuscules / Minuscules

Dans Oracle, les fonctions de manipulation de chaînes de caractères opèrent de deux façons. Certaines créent de nouveaux objets à partir d'anciens et produisent comme résultat une modification des données originales, par exemple une conversion en minuscules de caractères majuscules. D'autres permettent d'obtenir des informations relatives à des données, comme le nombre de caractères contenus dans une phrase ou un mot.

## LOWER

La fonction « **LOWER** » permet de convertir les majuscules en minuscules.



```
SQL> SELECT NOM, LOWER(NOM) FROM EMPLOYES WHERE ROWNUM < 5;
```

```
NOM          LOWER(NOM)
-----
Berlitz      berlitz
Nocella      nocella
```



Herve	herve
Mangeard	mangeard

## UPPER

La fonction « **UPPER** » permet de convertir les minuscules en majuscules.



```
SQL> SELECT NOM_PRODUIT, UPPER(NOM_PRODUIT) FROM PRODUITS WHERE ROWNUM < 5;
```

NOM_PRODUIT	UPPER(NOM_PRODUIT)
Chai	CHAI
Chang	CHANG
Aniseed Syrup	ANISEED SYRUP
Chef Anton's Cajun Seasoning	CHEF ANTON'S CAJUN SEASONING

## INITCAP

La fonction « **INITCAP** » permet de convertir en majuscule la première lettre de chaque mot de la chaîne, et toutes les autres lettres en minuscule. Sont considérés comme **séparateurs de mots** tous les caractères qui ne sont pas des lettres.



```
SQL> SELECT INITCAP('LE LANGE_SQL ET pL/sQL pOUR OrACLE')
2 "Fonction INITCAP" FROM DUAL;
```

```
Fonction INITCAP
-----
Le Lange_Sql Et Pl/Sql Pour Oracle
```

```
SQL> SELECT NOM_PRODUIT, INITCAP(QUANTITE) FROM PRODUITS WHERE ROWNUM < 10;
```

NOM_PRODUIT	INITCAP(QUANTITE)
Chai	10 Boîtes X 20 Sacs
Chang	24 Bouteilles (1 Litre)
Aniseed Syrup	22 Bouteilles (550 Ml)
Chef Anton's Cajun Seasoning	48 Pots (6 Onces)
Grandma's Boysenberry Spread	12 Pots (8 Onces)
Uncle Bob's Organic Dried Pears	12 Cartons (1 Kg)
Northwoods Cranberry Sauce	12 Pots (12 Onces)
Mishi Kobe Niku	18 Cartons (500 G)
Ikura	12 Pots (200 G)

## CONCAT

La fonction « **CONCAT** » effectue la concaténation de deux chaînes de caractères.

**CONCAT(EXPRESSION1,EXPRESSION2)= EXPRESSION1||EXPRESSION2**



```
SQL> SELECT CONCAT(CONCAT(NOM, ' '), PRENOM), NOM||' '||PRENOM FROM EMPLOYES;
```

CONCAT(CONCAT(NOM, ' '), PRENOM)	NOM  ' '  PRENOM
Fuller Andrew	Fuller Andrew
Buchanan Steven	Buchanan Steven
Peacock Margaret	Peacock Margaret
Leverling Janet	Leverling Janet
Davolio Nancy	Davolio Nancy
Dodsworth Anne	Dodsworth Anne
...	

# Manipulation de chaînes

## LPAD

La fonction « **LPAD** » complète, ou tronque sur la gauche à une longueur donnée la chaîne de caractères.

**LPAD (CHAÎNE1, LONG[, CHAÎNE2]) = RETOUR**

<b>CHAÎNE1</b>	La chaîne à traiter.
<b>CHAÎNE2</b>	Un ou plusieurs caractères utilisés comme modèle pour le remplissage. Le paramètre est optionnel ; par défaut, la chaîne est complétée par des espaces.
<b>LONG</b>	Le paramètre LONG est la longueur de la chaîne de caractères après le traitement.
<b>RETOUR</b>	La chaîne traitée redimensionne à la longueur LONG. Si LONG est supérieur à la longueur de la chaîne on remplit la chaîne avec le modèle CHAÎNE2, sinon la chaîne est tronquée en éliminant la fin.



```
SQL> SELECT ROWNUM R, NOM, LENGTH(NOM) LN, LPAD(NOM,8,'*') L1,
2 LPAD(NOM,14,'&#') L2, LPAD(PRENOM,14) L3 FROM EMPLOYES;
```

R	NOM	LN	L1	L2	L3
1	Berlioz	7	*Berlioz	&#&#&#&Berlioz	Jacques
2	Nocella	7	*Nocella	&#&#&#&Nocella	Guy
3	Herve	5	***Herve	&#&#&#&#Herve	Didier
4	Mangeard	8	Mangeard	&#&#&#Mangeard	Jocelyne
5	Cazade	6	**Cazade	&#&#&#*Cazade	Anne-Claire
6	Devie	5	***Devie	&#&#&#&#Devie	Thérèse
7	Peacock	7	*Peacock	&#&#&#&Peacock	Margaret
8	Idesheim	8	Idesheim	&#&#&#Idesheim	Annick
9	Rollet	6	**Rollet	&#&#&#*Rollet	Philippe
10	Silberreiss	11	Silberre	&#&#Silberreiss	Albert
11	Weiss	5	***Weiss	&#&#&#&#Weiss	Sylvie
12	Delorgue	8	Delorgue	&#&#&#Delorgue	Jean
13	Zonca	5	***Zonca	&#&#&#&#Zonca	Virginie
14	Twardowski	10	Twardows	&#&#Twardowski	Colette
15	Coutou	6	**Coutou	&#&#&#*Coutou	Myriam
16	King	4	***King	&#&#&#&#King	Robert
17	Ragon	5	***Ragon	&#&#&#&#Ragon	André
18	Dohr	4	***Dohr	&#&#&#&#Dohr	Sylvie
19	Maurousset	10	Maurouss	&#&#Maurousset	James
20	Pouetre	7	*Pouetre	&#&#&#&Pouetre	Camille-Hélène
21	Montesinos	10	Montesin	&#&#Montesinos	Aline
22	Aubert	6	**Aubert	&#&#&#*Aubert	Maria
23	Thomas	6	**Thomas	&#&#&#*Thomas	Patricia
24	Falatik	7	*Falatik	&#&#&#&Falatik	Bernard
...					

Pour les enregistrements 10, 14, 19 et 21 le traitement « **LPAD** » effectué dans la colonne L1 tronque le nom de l'employé à huit caractères.

## RPAD

La fonction « **RPAD** » complète, ou tronque sur la droite à une longueur donnée la chaîne de caractères.

```
RPAD (CHAÎNE1, LONG[, CHAÎNE2]) = RETOUR
```

**CHAÎNE2** Un ou plusieurs caractères utilisés comme modèle pour le remplissage. Le paramètre est optionnel, par défaut, la chaîne est complétée par des espaces.

**LONG** Le paramètre LONG est la longueur de la chaîne de caractères après le traitement.

**RETOUR** La chaîne traitée redimensionne à la longueur `LONG`. Si `LONG` est supérieur à la longueur de la chaîne on remplit la chaîne avec le modèle `CHAÎNE2`, sinon la chaîne est tronquée en éliminant la fin.



```
SQL> SELECT ROWNUM R, NOM, LENGTH(NOM) LN, RPAD(PRENOM, 10) L1,
2      RPAD(NOM, 8, '*') L2, RPAD(NOM, 20, '&#39;') L3 FROM EMPLOYEES;
```

R	NOM	LN	L1	L2	L3
1	Berlioz	7	Jacques	Berlioz*	Berlioz&*&*&*&*&*
2	Nocella	7	Guy	Nocella*	Nocella&*&*&*&*&*
3	Herve	5	Didier	Herve***	Herve&*&*&*&*&*
4	Mangeard	8	Jocelyne	Mangeard	Mangeard&*&*&*&*&*
5	Cazade	6	Anne-Clair	Cazade**	Cazade&*&*&*&*&*
6	Devie	5	Thérèse	Devie***	Devie&*&*&*&*&*
7	Peacock	7	Margaret	Peacock*	Peacock&*&*&*&*&*
8	Idesheim	8	Annick	Idesheim	Idesheim&*&*&*&*&*
9	Rollet	6	Philippe	Rollet**	Rollet&*&*&*&*&*
10	<b>Silberreiss</b>	11	<b>Albert</b>	<b>Silberre</b>	<b>Silberreiss&amp;*&amp;*&amp;*&amp;*&amp;*</b>
11	Weiss	5	Sylvie	Weiss***	Weiss&*&*&*&*&*
12	Delorgue	8	Jean	Delorgue	Delorgue&*&*&*&*&*
13	Zonca	5	Virginie	Zonca***	Zonca&*&*&*&*&*
14	<b>Twardowski</b>	10	<b>Colette</b>	<b>Twardows</b>	<b>Twardowski&amp;*&amp;*&amp;*&amp;*&amp;*</b>
15	Coutou	6	Myriam	Coutou**	Coutou&*&*&*&*&*
16	King	4	Robert	King****	King&*&*&*&*&*&*
17	Ragon	5	André	Ragon***	Ragon&*&*&*&*&*
18	Dohr	4	Sylvie	Dohr****	Dohr&*&*&*&*&*&*
19	<b>Maurousset</b>	10	<b>James</b>	<b>Maurouss</b>	<b>Maurousset&amp;*&amp;*&amp;*&amp;*&amp;*</b>
20	Pouetre	7	Camille-Hé	Pouetre*	Pouetre&*&*&*&*&*
21	<b>Montesinos</b>	10	<b>Aline</b>	<b>Montesin</b>	<b>Montesinos&amp;*&amp;*&amp;*&amp;*&amp;*</b>

## LTRIM

La fonction « **LTRIM** » supprime un ensemble des caractères indésirables à gauche de la chaîne de caractères.

**LTRIM(CHAÎNE1[,CHAÎNE2]) = RETOUR**

**CHAÎNE1** La chaîne à traiter.

**CHAÎNE2** Un caractère ou une liste de caractères indésirables. La requête recherche et efface **une série contiguë** d'un ou plusieurs caractères de la liste, positionnés à gauche de la chaîne de caractères. Le paramètre CHAÎNE2 est optionnel ; par défaut, la chaîne efface tous les espaces.



```
SQL> SELECT LTRIM('          Chaîne') c1,
2  LTRIM('*****Chaîne') c2,
3  LTRIM('A2BD1AC3BCD4-Chaîne', 'ABCD1234') c3 FROM DUAL;
```

C1	C2	C3
Chaîne	*****Chaîne	-Chaîne

## RTRIM

La fonction « **LTRIM** » supprime un ensemble des caractères indésirables à droite de la chaîne de caractères.

**RTRIM(CHAÎNE1[,CHAÎNE2]) = RETOUR**

**CHAÎNE1**

La chaîne à traiter.

**CHAÎNE2**

Un caractère ou une liste de caractères indésirables. La requête recherche et efface **une série contiguë** d'un ou plusieurs caractères de la liste, positionnés à droite de la chaîne de caractères. Le paramètre CHAÎNE2 est optionnel ; par défaut, la chaîne efface tous les blancs.



```
SQL> SELECT RTRIM('Chaîne*****', '*') FROM DUAL ;
```

```
RTRIM(
-----
Chaîne
```

```
SQL> SELECT QUANTITE, RTRIM( LTRIM(QUANTITE, ' 0123456789')
2  , ' 0123456789()kg') RLT FROM PRODUITS
3  WHERE REGEXP_LIKE(QUANTITE, '^([0-9]+.*[\(].*(g[\)])$','x');
```

QUANTITE	RLT
12 cartons (1 kg)	cartons
18 cartons (500 g)	cartons
12 pots (200 g)	pots
1 carton (1 kg)	carton
10 cartons (500 g)	cartons
1 boîtes (2 kg)	boîtes
20 boîtes (1 kg)	boîtes
24 cartons (500 g)	cartons
12 cartons (250 g)	cartons
...	

## Extraire une sous chaîne

### SUBSTR

La fonction « **SUBSTR** » extrait de la chaîne de caractère une sous-chaîne à partir d'une position et longueur donnée.

**SUBSTR(CHAÎNE1, POSITION[, LONGUEUR]) = RETOUR**

**CHAÎNE1**

La chaîne à traiter.

**POSITION**

La position de départ pour la nouvelle chaîne.

**LONGUEUR**

Le paramètre **LONGUEUR** est facultatif ; il détermine le nombre des caractères de la nouvelle chaîne ; par défaut la sous-chaîne va jusqu'à l'extrémité de la chaîne.



```
SQL> SELECT NOM, SUBSTR(NOM,3,5) S1, SUBSTR(NOM,3,25) S2, SUBSTR(NOM,3) S3
      2 FROM EMPLOYES;
```

NOM	S1	S2	S3
Berlioz	rlioz	rlioz	rlioz
Nocella	cella	cella	cella
Herve	rve	rve	rve
Mangeard	ngear	ngear	ngear
Cazade	zade	zade	zade
Devie	vie	vie	vie
Peacock	acock	acock	acock
Idesheim	eshei	esheim	esheim
Rollet	llet	llet	llet
Silberreiss	lberr	lberreiss	lberreiss
Weiss	iss	iss	iss
Delorgue	lorgu	lorgue	lorgue
Zonca	nca	nca	nca
Twardowski	ardow	ardowski	ardowski
...			

**REGEXP\_SUBSTR**

La fonction « **REGEXP\_SUBSTR** » extrait de la chaîne de caractère une sous-chaîne en décrivant la sous-chaîne extraite à l'aide d'une expression régulière. Ceci permet d'extraire la sous-chaîne sans connaître sa position exacte, ni même sa longueur.

**REGEXP\_SUBSTR (CHAÎNE1,CHAÎNE2**

**[, POSITION, OCCURENCE, PARAMETRE]) = NUMERIQUE**

<b>CHAÎNE1</b>	La chaîne à traiter.
<b>CHAÎNE2</b>	Une expression rationnelle qui permet de décrire un ensemble de chaînes. Les expressions rationnelles sont construites comme des opérations arithmétiques, en utilisant des opérateurs divers pour combiner des expressions plus petites. (Voir « <b>REGEXP_LIKE</b> »)
<b>POSITION</b>	La position de départ pour la recherche, paramètre facultatif vaut 1 par défaut.
<b>OCCURENCE</b>	Le paramètre, <b>OCCURENCE</b> permet de rechercher la nième occurrence <b>CHAÎNE2</b> dans la chaîne. Ce paramètre facultatif vaut 1 par défaut.
<b>PARAMETRE</b>	Paramètre de correspondance de l'expression rationnelle. Il peut avoir les valeurs suivantes : <ul style="list-style-type: none"> <li><b>i</b> Ignorer les différences majuscules/minuscules dans le motif.</li> <li><b>c</b> Respecter les différences majuscules/minuscules dans le motif.</li> <li><b>x</b> Ignorer les espaces dans la chaîne à traiter.</li> </ul>
<b>NUMERIQUE</b>	La position trouvée dans la chaîne de caractère. La valeur zéro signifie que la sous-chaîne n'a pas été trouvée.



```
SQL> SELECT QUANTITE, REGEXP_SUBSTR(QUANTITE,
2   '(carton|bouteilles) [^\(]') REGEXP_SUBSTR FROM PRODUITS
4   WHERE REGEXP_LIKE(QUANTITE, '(carton|bouteilles) [^\(]');
```

QUANTITE	REGEXP_SUBSTR
24 bouteilles (1 litre)	bouteilles
22 bouteilles (550 ml)	bouteilles
12 cartons (1 kg)	cartons
18 cartons (500 g)	cartons
...	

```
SQL> SELECT QUANTITE, REGEXP_SUBSTR(QUANTITE,
2   '\([].*[\])]' ) REGEXP_SUBSTR FROM PRODUITS
4   WHERE REGEXP_LIKE(QUANTITE, '\([].*[\])]' );
```

QUANTITE	REGEXP_SUBSTR
24 bouteilles (1 litre)	(1 litre)
22 bouteilles (550 ml)	(550 ml)
48 pots (6 onces)	(6 onces)
12 pots (8 onces)	(8 onces)
12 cartons (1 kg)	(1 kg)
...	

```
SQL> DEFINE CHAINE01='-xyz-Première -xyz-Deuxième -xyz-Troisième'
```

```
SQL> SET VERIFY OFF
```

```
SQL> SELECT REGEXP_SUBSTR( '&CHAINE01', '-xyz-.{1,9}',1,1) RS1,
2   REGEXP_SUBSTR( '&CHAINE01', '-xyz-.{1,9}',1,2) RS2,
3   REGEXP_SUBSTR( '&CHAINE01', '-xyz-.{1,9}',1,3) RS3,
4   REGEXP_SUBSTR( '&CHAINE01', '-xyz-.{1,9}',24,1) RS4
5   FROM DUAL;
```

RS1	RS2	RS3	RS4
-xyz-Première	-xyz-Deuxième	-xyz-Troisième	-xyz-Troisième

## Recherche dans la chaîne

### INSTR

La fonction « **INSTR** » recherche la première occurrence du caractère ou de la chaîne de caractères donnée.

**INSTR(CHAÎNE1, CHAÎNE2, POSITION, OCCURENCE) = NUMERIQUE**

**CHAÎNE1** La chaîne à traiter.

**CHAÎNE2** Une sous-chaîne constituée d'un ou plusieurs caractères recherchés.

**POSITION** La position de départ pour la recherche ; paramètre facultatif valant 1 par défaut. Une valeur négative pour POSITION signifie une recherche à partir de la fin de la chaîne.

## OCCURENCE

Le paramètre OCCURENCE permet de rechercher la n-ième occurrence CHAÎNE2 dans la chaîne. Ce paramètre facultatif vaut 1 par défaut.

## NUMERIQUE

La position trouvée dans la chaîne de caractère. Zéro signifie que la sous-chaîne n'a pas été trouvée.



```
SQL> SELECT QUANTITE, INSTR(QUANTITE,' ') "1", INSTR(QUANTITE,' ',5,2) "2",
2 INSTR(QUANTITE,'kg', -1) "3", INSTR(QUANTITE,' ', -10, 2) "4"
3 FROM PRODUITS ;
```

QUANTITE	1	2	3	4
10 boîtes x 20 sacs	3	12	0	3
24 bouteilles (1 litre)	3	17	0	3
22 bouteilles (550 ml)	3	19	0	0
48 pots (6 onces)	3	11	0	3
12 pots (8 onces)	3	11	0	3
12 cartons (1 kg)	3	14	15	0
12 pots (12 onces)	3	12	0	3
18 cartons (500 g)	3	16	0	0
12 pots (200 g)	3	13	0	0
1 carton (1 kg)	2	12	13	0
10 cartons (500 g)	3	16	0	0
1 boîtes (2 kg)	2	12	13	0
20 boîtes (1 kg)	3	13	14	0
10 boîtes x 12 pièces	3	12	0	10
30 boîtes	3	0	0	0
...				

Dans l'exemple précédent, vous pouvez observer les différentes possibilités d'utilisation de la fonction INSTR.

La requête retourne les valeurs suivantes :

- 1 : la première occurrence, du caractère espace, trouvée dans QUANTITE, recherchée à partir du début.
- 2 : la deuxième occurrence, du caractère espace, trouvée dans QUANTITE, recherchée à partir du cinquième caractère.
- 3 : la première occurrence, de la chaîne de caractère 'kg', trouvée dans QUANTITE, recherchée à partir de la fin.
- 4 : la deuxième occurrence, du caractère espace, trouvée dans QUANTITE, recherchée à partir de 10e caractère à partir de la fin.

## REGEXP\_INSTR



La fonction « **REGEXP\_INSTR** » permet de localiser l'emplacement de départ d'une sous-chaîne à l'intérieur d'une chaîne. L'avantage réside dans le fait qu'il n'est pas nécessaire de citer la sous chaîne, mais qu'il suffit de la décrire à l'aide d'une expression régulière pour la localiser.

**REGEXP\_INSTR**(CHAÎNE1,CHAÎNE2

[, POSITION, OCCURENCE, PARAMETRE] )

= NUMERIQUE

**CHAÎNE1**

La chaîne à traiter.

**CHAÎNE2**

Une expression rationnelle qui permet de décrire un ensemble de chaînes. Les expressions rationnelles sont construites comme des

opérations arithmétiques, en utilisant des opérateurs divers pour combiner des expressions plus petites. (Voir « **REGEXP\_LIKE** »)

**POSITION**

La position de départ pour la recherche, paramètre facultatif vaut 1 par défaut.

**OCCURENCE**

Le paramètre, OCCURENCE permet de rechercher la nième occurrence CHAÎNE2 dans la chaîne. Ce paramètre facultatif vaut 1 par défaut.

**PARAMETRE**

Paramètre de correspondance de l'expression rationnelle. Il peut avoir les valeurs suivantes :

- i** Ignorer les différences majuscules/minuscules dans le motif.
- c** Respecter les différences majuscules/minuscules dans le motif.
- x** Ignorer les espaces dans la chaîne à traiter.

**NUMERIQUE**

La position trouvée dans la chaîne de caractère. La valeur zéro signifie que la sous-chaîne n'a pas été trouvée.



```
SQL> SELECT QUANTITE, REGEXP_INSTR(QUANTITE, '\(.*\)') "REGEXP_INSTR",
2 REGEXP_SUBSTR(QUANTITE, '\(.*\)') "REGEXP_SUBSTR" FROM PRODUITS
3 WHERE REGEXP_LIKE(QUANTITE, '\(.*\)')
```

QUANTITE	REGEXP_INSTR	REGEXP_SUBSTR
24 bouteilles (1 litre)	15	(1 litre)
12 bouteilles (550 ml)	15	(550 ml)
48 pots (6 onces)	9	(6 onces)
12 pots (8 onces)	9	(8 onces)
12 cartons (1 kg)	12	(1 kg)
12 pots (12 onces)	9	(12 onces)
18 cartons (500 g)	12	(500 g)
12 pots (200 g)	9	(200 g)
1 carton (1 kg)	10	(1 kg)
10 cartons (500 g)	12	(500 g)
1 bo`tes (2 kg)	10	(2 kg)
20 bo`tes (1 kg)	11	(1 kg)
24 cartons (500 g)	12	(500 g)
12 cartons (250 g)	12	(250 g)
20 verres (450 g)	11	(450 g)
...		

```
SQL> DEFINE CHAINE01='-xyz-Première -xyz-Deuxième -xyz-Troisième'
```

```
SQL> SET VERIFY OFF
```

```
SQL> SELECT '&CHAINE01' CHAINE FROM DUAL;
```

```
CHAINE
```

```
-----
-xyz-Première -xyz-Deuxième -xyz-Troisième
```

```
SQL> SELECT REGEXP_INSTR( '&CHAINE01', '-xyz-',1,1) RS1,
2 REGEXP_INSTR( '&CHAINE01', '-xyz-',1,2) RS1,
3 REGEXP_INSTR( '&CHAINE01', '-xyz-',1,3) RS3,
4 REGEXP_INSTR( '&CHAINE01', '-xyz-',24,1) RS4
```



5	FROM DUAL;			
	RS1	RS1	RS3	RS4
	-----	-----	-----	-----
	1	16	31	31

## Rechercher et remplacer

### REPLACE

La fonction « **REPLACE** » permet de remplacer dans la chaîne de caractères, toutes les séquences du caractère ou de la chaîne de caractères donnée.

**REPLACE (CHAÎNE1, CHAÎNE2, CHAÎNE3) = RETOUR**

<b>CHAÎNE1</b>	La chaîne à traiter.
<b>CHAÎNE2</b>	Un caractère ou une chaîne de caractères à remplacer.
<b>CHAÎNE3</b>	Un caractère ou une chaîne pour remplacer CHAÎNE2. Si la chaîne est vide, la fonction efface les caractères recherchés.



```
SQL> SELECT REPLACE('JACK et JUE', 'J', '') REPLACE1,
2         REPLACE('JACK et JUE', 'J', 'BL') REPLACE2 FROM DUAL;
```

REPLACE1	REPLACE2
-----	-----
ACK et UE	BLACK et BLUE

### REGEXP\_REPLACE



La fonction « **REGEXP\_REPLACE** » permet de localiser et de remplacer toutes les séquences d'une sous-chaîne à l'intérieur d'une chaîne. L'avantage réside dans le fait qu'il n'est pas nécessaire de citer la sous chaîne, mais qu'il suffit de la décrire à l'aide d'une expression régulière pour la localiser.

**REGEXP\_REPLACE (CHAÎNE1, CHAÎNE2, CHAÎNE3**

**[, POSITION, OCCURRENCE, PARAMETRE]) = RETOUR**

<b>CHAÎNE1</b>	La chaîne à traiter.
<b>CHAÎNE2</b>	Une expression rationnelle qui permet de décrire un ensemble de chaînes. Les expressions rationnelles sont construites comme des opérations arithmétiques, en utilisant des opérateurs divers pour combiner des expressions plus petites. (Voir « <b>REGEXP_LIKE</b> »)
<b>CHAÎNE3</b>	Un caractère ou une chaîne pour remplacer CHAÎNE2. Si la chaîne est vide, la fonction efface les caractères recherchés. Il est possible d'avoir plusieurs occurrences de la chaîne recherchée. Pour chaque occurrence vous pouvez donner une valeur différente de remplacement.
<b>PARAMETRE</b>	Paramètre de correspondance de l'expression rationnelle. Il peut avoir les valeurs suivantes :

- i** Ignorer les différences majuscules/minuscules dans le motif.
- c** Respecter les différences majuscules/minuscules dans le motif.
- x** Ignorer les espaces dans la chaîne à traiter.

**POSITION**

La position de départ pour la recherche ; paramètre facultatif valant 1 par défaut.

**OCCURENCE**

Le paramètre, OCCURENCE, permet de rechercher la nième occurrence CHAÎNE2 dans la chaîne. Ce paramètre facultatif vaut 1 par défaut.



```
SQL> SELECT QUANTITE, REGEXP_REPLACE(QUANTITE, '\(.*\)'.*$',
2      '--XXXXXX--') "REGEXP_REPLACE" FROM PRODUITS
4 WHERE REGEXP_LIKE(QUANTITE, '\(.*\)');
```

QUANTITE	REGEXP_REPLACE
24 bouteilles (1 litre)	24 bouteilles --XXXXXX--
22 bouteilles (550 ml)	22 bouteilles --XXXXXX--
48 pots (6 onces)	48 pots --XXXXXX--
12 pots (8 onces)	12 pots --XXXXXX--
12 cartons (1 kg)	12 cartons --XXXXXX--
12 pots (12 onces)	12 pots --XXXXXX--
18 cartons (500 g)	18 cartons --XXXXXX--
12 pots (200 g)	12 pots --XXXXXX--
1 carton (1 kg)	1 carton --XXXXXX--
10 cartons (500 g)	10 cartons --XXXXXX--
1 boîtes (2 kg)	1 boîtes --XXXXXX--
20 boîtes (1 kg)	20 boîtes --XXXXXX--
24 cartons (500 g)	24 cartons --XXXXXX--
12 cartons (250 g)	12 cartons --XXXXXX--
20 verres (450 g)	20 verres --XXXXXX--
25 canettes (825 g)	25 canettes --XXXXXX--
22 cartons (200 g)	22 cartons --XXXXXX--
24 bouteilles (70 cl)	24 bouteilles --XXXXXX--
24 bouteilles (1 litre)	24 bouteilles --XXXXXX--
24 pots (250 g)	24 pots --XXXXXX--
12 cartons (500 g)	12 cartons --XXXXXX--
1 bouteille (750 cc) 55%	1 bouteille --XXXXXX--
...	

Dans l'exemple précédent on remplace dans la valeur du champ QUALITE tout ce qui est contenu entre les deux parenthèses « ( » et « ) » par « --XXXXXX-- ».

Il est possible d'avoir plusieurs occurrences de la chaîne recherchée. Pour chaque occurrence vous pouvez donner une valeur différente de remplacement.

Dans l'exemple suivant on effectue un traitement différent pour les deux parenthèses ; pour la première colonne, on rajoute le caractère « - » avant et après chaque parenthèse, et pour la deuxième colonne, on remplace « ( » par « [ » et « ) » par « ] », puis on remplace « ( » par « : » et on efface « ) ».



```
SQL> SELECT REGEXP_REPLACE(QUANTITE, '\(.*\)') A1,
2      REGEXP_REPLACE(QUANTITE, '\(.*\)') A2,
3      REGEXP_REPLACE(QUANTITE, '\(.*\)') A3
4 FROM PRODUITS WHERE REGEXP_LIKE(QUANTITE, '(cartons|pots).*\(.*)');
```

A1	A2	A3
48 pots -(-6 onces)-	48 pots [6 onces]	48 pots : 6 onces
12 pots -(-8 onces)-	12 pots [8 onces]	12 pots : 8 onces
12 cartons -(-1 kg)-	12 cartons [1 kg]	12 cartons : 1 kg
12 pots -(-12 onces)-	12 pots [12 onces]	12 pots : 12 onces

18 cartons -(-500 g)-	18 cartons [500 g]	18 cartons : 500 g
12 pots -(-200 g)-	12 pots [200 g]	12 pots : 200 g
10 cartons -(-500 g)-	10 cartons [500 g]	10 cartons : 500 g
24 cartons -(-500 g)-	24 cartons [500 g]	24 cartons : 500 g
12 cartons -(-250 g)-	12 cartons [250 g]	12 cartons : 250 g
22 cartons -(-200 g)-	22 cartons [200 g]	22 cartons : 200 g
24 pots -(-250 g)-	24 pots [250 g]	24 pots : 250 g
12 cartons -(-500 g)-	12 cartons [500 g]	12 cartons : 500 g
32 cartons -(-1 kg)-	32 cartons [1 kg]	32 cartons : 1 kg
24 cartons -(-50 g)-	24 cartons [50 g]	24 cartons : 50 g
50 cartons -(-300 g)-	50 cartons [300 g]	50 cartons : 300 g
24 cartons -(-250 g)-	24 cartons [250 g]	24 cartons : 250 g
24 pots -(-8 onces)-	24 pots [8 onces]	24 pots : 8 onces
24 cartons -(-200 g)-	24 cartons [200 g]	24 cartons : 200 g
24 pots -(-150 g)-	24 pots [150 g]	24 pots : 150 g
40 cartons -(-100 g)-	40 cartons [100 g]	40 cartons : 100 g
12 cartons -(-100 g)-	12 cartons [100 g]	12 cartons : 100 g
20 cartons -(-2 kg)-	20 cartons [2 kg]	20 cartons : 2 kg
24 cartons -(-250 g)-	24 cartons [250 g]	24 cartons : 250 g
15 pots -(-625 g)-	15 pots [625 g]	15 pots : 625 g
10 cartons -(-500 g)-	10 cartons [500 g]	10 cartons : 500 g

Les expressions '\1', '\2' et '\3' sont les chaînes de caractères de l'expression rationnelle, qui doivent être remplacées. Dans notre exemple ce sont : '(\() ', '(.\* )' et '(\)) '. Ainsi chacune de ces chaînes de caractères peut être utilisée ou non dans la nouvelle définition et on peut rajouter d'autres caractères pour formater la chaîne avec une grande flexibilité.



```
SQL> SELECT TELEPHONE, REGEXP_REPLACE(LPAD(REGEXP_REPLACE( TELEPHONE,
2 '([^\0-9])',11),'([[:digit:]]{3})'([[:digit:]]{2})'|'
3 '([[:digit:]]{2})'([[:digit:]]{2})', '(\1) \2.\3.\4.') "Téléphone"
4 FROM CLIENTS;
```

TELEPHONE	Téléphone
(71) 555-2282	( 7) 15.55.22.82
0241-039123	( 02) 41.03.91.23
02.40.67.88.88	( 02) 40.67.88.88
(71) 555-0297	( 7) 15.55.02.97
7675-3425	( ) 76.75.34.25
(11) 555-9857	( 1) 15.55.98.57
(91) 555 94 44	( 9) 15.55.94.44
03.20.16.10.16	( 03) 20.16.10.16
0695-34 67 21	( 06) 95.34.67.21
089-0877310	( 08) 90.87.73.10
02.40.32.21.21	( 02) 40.32.21.21
011-4988260	( 01) 14.98.82.60
...	

## Traduction de chaînes

### TRANSLATE

La fonction « **TRANSLATE** » remplace dans une chaîne de caractère chaque caractère présent dans une liste source par son correspondant, caractère ayant la même position, dans une liste cible.

**TRANSLATE (CHAÎNE1,CHAÎNE2,CHAÎNE3) = RETOUR**

**CHAÎNE1** La chaîne à traiter.

**CHAÎNE2** Une chaîne de caractères considérée comme une liste de caractères source, devant être remplacée par les caractères de la liste cible.

**CHAÎNE3** Une chaîne de caractères considérée comme une liste de caractères cible. Si la liste de caractères cible est plus courte que la liste des caractères source, les caractères de la liste source qui n'ont pas de correspondant sont supprimés.



```
SQL> SELECT NOM,TRANSLATE(NOM,'ABCabc','12345') FROM EMPLOYES;
```

NOM	TRANSLATE(NOM,'ABCABC','12345')
Berlioz	2erlioz
Nocella	Noell4
Herve	Herve
Mangeard	M4nge4rd
Cazade	34z4de
Devie	Devie
Peacock	Pe4ok
Idesheim	Idesheim
Rollet	Rollet
Silberreiss	Sil5erreiss
Weiss	Weiss
Delorgue	Delorgue
Zonca	Zon4
...	

## SOUNDEX

La fonction « **SOUNDEX** » permet de trouver des mots qui "sonnent" comme ceux spécifiés, quelle que soit leur orthographe. « **SOUNDEX** » est presque toujours utilisée dans une clause « **WHERE** » et il est particulièrement utile pour trouver des mots dont vous n'êtes pas sûr de bien connaître l'orthographe.

**SOUNDEX (CHAÎNE1) = SOUNDEX (CHAÎNE2)**



```
SQL> SELECT SOCIETE,ADRESSE FROM CLIENTS
2 WHERE SOUNDEX(SOCIETE) = SOUNDEX('bllaumter');
```

SOCIETE	ADRESSE
Blondel père et fils	24, place Kléber...

## ASCII

La fonction « **ASCII** » retourne le code ASCII du caractère.

**ASCII ('CARACTER') = NUMERIQUE**



```
SQL> SELECT ASCII('A') FROM DUAL;
```

ASCII('A')
65

## CHR, NCHR

La fonction « **CHR** » ou « **NCHR** » retourne le caractère de la valeur ASCII.

**CHR(NUMERIQUE) = 'CARACTER'**



```
SQL> SELECT CHR(65),NCHR(65),VSIZE(CHR(65)),VSIZE(NCHR(65)) FROM DUAL;
```

```
C N VSIZE(CHR(65)) VSIZE(NCHR(65))
--
A A                1                2
```

## Types numériques

### NUMBER (P,S)

Champ de longueur variable acceptant la valeur zéro ainsi que des nombres négatifs et positifs. La précision maximum de « **NUMBER** », est de 38 chiffres de  $10^{-130} \div 10^{126}$ . Lors de la déclaration, il est possible de définir la précision P chiffres significatifs stockés et un arrondi à droite de la marque décimale à S chiffres entre  $-84 \div 127$ .

Chaque colonne de type « **NUMBER** » nécessite de 1 à 22 bytes pour le stockage.

<i>Valeur d'affectation</i>	<i>Déclaration</i>	<i>Valeur stocke dans la table</i>
7456123.89	NUMBER	7456123.89
7456123.89	NUMBER(9)	7456124
7456123.89	NUMBER(9,2)	7456123.89
7456123.89	NUMBER(9,1)	7456123.9
7456123.89	NUMBER(6)	précision trop élevée
7456123.89	NUMBER(7,-2)	7456100
7456123.89	NUMBER(7,2)	précision trop élevée
.01234	NUMBER(4,5)	.01234
.00012	NUMBER(4,5)	.00012
.000127	NUMBER(4,5)	.00013
.0000012	NUMBER(2,7)	.0000012
.00000123	NUMBER(2,7)	.0000012

### BINARY\_FLOAT

Nombre réel à virgule flottante encodé sur 32 bits.

Chaque colonne de type « **BINARY\_FLOAT** » nécessite 5 bytes pour le stockage.

### BINARY\_DOUBLE

Nombre réel à virgule flottante encodé sur 64 bits.

Chaque colonne de type « **BINARY\_DOUBLE** » nécessite 9 bytes pour le stockage.

	<i>BINARY_FLOAT</i>	<i>BINARY_DOUBLE</i>
L'entier maximum	1.79e308	3.4e38
L'entier minimum	-1.79e308	-3.4e38
La plus petite valeur positive	2.3e-308	1.2e-38
La plus petite valeur négative	-2.3e-308	-1.2e-38



### Attention

Les constantes numériques doivent comporter un suffixe '**F**' ou '**f**' pour les constantes réel à virgule flottante encodé sur 32 bits et '**D**' ou '**d**' pour les constantes réel à virgule flottante encodé sur 64 bits.

Ainsi Oracle converti directement les chaînes de caractères en « **BINARY\_FLOAT** » ou « **BINARY\_DOUBLE** ».



```
SQL> SELECT 2.5, 2.5f, 2.5D FROM DUAL;
```

```
2.5      2.5F      2.5D
-----
```

```
2,5      2,5E+000    2,5E+000
```

```
SQL> SELECT 2.5/0, 2.5f, 2.5D FROM DUAL;
```

```
SELECT 2.5/0, 2.5f, 2.5D FROM DUAL
```

```
*
```

ERREUR à la ligne 1 :

ORA-01476: le diviseur est égal à zéro

```
SQL> SELECT 2.5, 2.5f/0, 2.5D/0 FROM DUAL;
```

```
2.5      2.5F/0      2.5D/0
-----
```

```
2,5      Inf      Inf
```

### Note

Les nombres réels à virgule flottante supportent la division par zéro sans produire une erreur ; ce n'est pas le cas des valeurs numériques classiques. Evidemment, la valeur qui résulte de cette opération est infinie.



```
SQL> DESC VIRGULE_FLOTTANTE
```

```
Nom                                NULL ?    Type
-----
ID                                NOT NULL  NUMBER(2)
VF_FLOAT                          BINARY_FLOAT
VF_DOUBLE                         BINARY_DOUBLE
DESCRIPT                          VARCHAR2(100)
```

```
SQL> SELECT VF_FLOAT, VF_DOUBLE FROM VIRGULE_FLOTTANTE;
```

```
VF_FLOAT  VF_DOUBLE
```

```
-----
          Nan          Nan
          Inf          Inf
3,403E+038 1,798E+308
1,175E-038 2,225E-308
1,175E-038 2,225E-308
1,401E-045 4,941E-324
```

Pour traiter les valeurs numériques à virgule flottante, Oracle fournit un ensemble de constantes.

<i>Constante</i>	<i>Description</i>
BINARY_FLOAT_NAN	Pas un numérique
BINARY_FLOAT_INFINITY	Infini
BINARY_FLOAT_MAX_NORMAL	3.40282347e+38
BINARY_FLOAT_MIN_NORMAL	1.17549435e-038
BINARY_FLOAT_MAX_SUBNORMAL	1.17549421e-038
BINARY_FLOAT_MIN_SUBNORMAL	1.40129846e-045
BINARY_DOUBLE_NAN	Pas un numérique
BINARY_DOUBLE_INFINITY	Infini
BINARY_DOUBLE_MAX_NORMAL	1.7976931348623157E+308
BINARY_DOUBLE_MIN_NORMAL	2.2250738585072014E-308
BINARY_DOUBLE_MAX_SUBNORMAL	2.2250738585072009E-308
BINARY_DOUBLE_MIN_SUBNORMAL	4.9406564584124654E-324

## Fonctions de contrôle

### NANVL

La fonction « **NANVL** » permet de remplacer une expression si elle n'est pas une valeur numérique ou si elle n'a pas de valeur, par une valeur significative. Cette fonction est essentiellement utilisée pour les nombres réels à virgule flottante.

**NANVL (EXPRESSION1, EXPRESSION2) = VALEUR\_DE\_RETOUR**

**EXPRESSION1** Une expression qui peut retourner la valeur NULL.

**EXPRESSION2** La valeur de remplacement dans le cas où EXPRESSION1 est égale à NULL ou n'est pas une valeur numérique.

**VALEUR\_DE\_RETOUR** Est égale à EXPRESSION2 si EXPRESSION1 est égale à NULL, sinon à EXPRESSION1.



```
SQL> DESC VIRGULE_FLOTTANTE
```

Nom	NULL ?	Type
ID	NOT NULL	NUMBER(2)
VF_FLOAT		BINARY_FLOAT
VF_DOUBLE		BINARY_DOUBLE
DESCRIPT		VARCHAR2(100)

```
SQL> SELECT VF_FLOAT, NANVL( VF_FLOAT,0), VF_DOUBLE, NANVL( VF_DOUBLE,0)
2 FROM VIRGULE_FLOTTANTE;
```

VF_FLOAT	NANVL(VF_FLOAT,0)	VF_DOUBLE	NANVL(VF_DOUBLE,0)
Nan	0	Nan	0
Inf	Inf	Inf	Inf
3,403E+038	3,403E+038	1,798E+308	1,798E+308
1,175E-038	1,175E-038	2,225E-308	2,225E-308
1,175E-038	1,175E-038	2,225E-308	2,225E-308
1,401E-045	1,401E-045	4,941E-324	4,941E-324

## IS NAN

L'opérateur logique « **IS NAN** » vérifie si la valeur retournée par EXPRESSION n'est une valeur numérique; alors retourne VRAI sinon FAUX.

### EXPRESSION IS NAN



```
SQL> SELECT VF_FLOAT FROM VIRGULE_FLOTTANTE WHERE VF_FLOAT IS NAN ;
```

VF_FLOAT
Nan

```
SQL> SELECT VF_DOUBLE FROM VIRGULE_FLOTTANTE WHERE VF_DOUBLE IS NOT NAN ;
```

VF_DOUBLE
Inf
1,798E+308
2,225E-308
2,225E-308
4,941E-324

## IS INFINITE

L'opérateur logique « **IS INFINITE** » vérifie si la valeur retournée par EXPRESSION n'est une valeur numérique infinie; alors retourne VRAI sinon FAUX.

### EXPRESSION IS INFINITE



```
SQL> SELECT VF_FLOAT, VF_DOUBLE FROM VIRGULE_FLOTTANTE
2 WHERE VF_FLOAT IS INFINITE AND VF_DOUBLE IS INFINITE;
```

VF_FLOAT	VF_DOUBLE
Inf	Inf

```
SQL> SELECT VF_FLOAT, VF_DOUBLE FROM VIRGULE_FLOTTANTE
2 WHERE VF_FLOAT IS NOT INFINITE AND VF_DOUBLE IS NOT INFINITE;
```



VF_FLOAT	VF_DOUBLE
Nan	Nan
3,403E+038	1,798E+308
1,175E-038	2,225E-308
1,175E-038	2,225E-308
1,401E-045	4,941E-324

Il est possible d'utiliser les opérateurs logiques pour traiter les valeurs numériques à virgule flottante, avec les constantes suivantes :

- BINARY\_FLOAT\_NAN
- BINARY\_FLOAT\_INFINITY
- BINARY\_DOUBLE\_NAN
- BINARY\_DOUBLE\_INFINITY



```
SQL> SELECT VF_FLOAT FROM VIRGULE_FLOTTANTE
2 WHERE VF_FLOAT = BINARY_FLOAT_NAN;
```

VF_FLOAT
Nan

```
SQL> SELECT VF_FLOAT FROM VIRGULE_FLOTTANTE
2 WHERE VF_FLOAT <> BINARY_FLOAT_NAN;
```

VF_FLOAT
Inf
3,403E+038
1,175E-038
1,175E-038
1,401E-045

```
SQL> SELECT VF_FLOAT, VF_DOUBLE FROM VIRGULE_FLOTTANTE
2 WHERE VF_FLOAT <> BINARY_FLOAT_INFINITY AND
3 VF_DOUBLE <> BINARY_DOUBLE_INFINITY;
```

VF_FLOAT	VF_DOUBLE
Nan	Nan
3,403E+038	1,798E+308
1,175E-038	2,225E-308
1,175E-038	2,225E-308
1,401E-045	4,941E-324

## Fonctions de calcul

Une expression arithmétique est une combinaison de noms de colonnes, de constantes et de fonctions arithmétiques combinées au moyen des **opérateurs** arithmétiques addition « + », soustraction « - », multiplication « \* » ou division « / ».

Les constantes et opérateurs arithmétiques ont été présentés précédemment ; les principales fonctions arithmétiques sont exposées ci-après.

## MOD

La fonction « **MOD** » permet de calculer le reste de la division du premier argument par le deuxième.

**MOD (ARGUMENT1, ARGUMENT2) = RESTE**



```
SQL> SELECT MOD(7,2) FROM DUAL;

MOD(7,2)
-----
        1
```

## POWER

La fonction « **POWER** » permet d'élever un nombre à une puissance.

**POWER (ARGUMENT1, ARGUMENT2) = ARGUMENT1<sup>ARGUMENT2</sup>**



```
SQL> SELECT POWER(3,2) FROM DUAL;

POWER(3,2)
-----
        9
```

## SQRT

La fonction « **SQRT** » permet de calculer une racine carrée. ARGUMENT doit être > 0.

**SQRT (ARGUMENT) = POWER (ARGUMENT, 0.5) = ARGUMENT<sup>0.5</sup>**



```
SQL> SELECT SQRT(9) FROM DUAL;

SQRT(9)
-----
        3

SQL> SELECT SQRT(-9) FROM DUAL
SELECT SQRT(-9) FROM DUAL ;
*
ERREUR à la ligne 1 :
ORA-01428: argument '-9' hors limites
```

## EXP

La fonction « **EXP** » permet de calculer une puissance de e (2,71828183...).

**EXP (ARGUMENT) = POWER(e, ARGUMENT) = e<sup>ARGUMENT</sup>**

## LOG

La fonction « **LOG** » permet de calculer un logarithme à base 10.

**LOG (ARGUMENT1, ARGUMENT2)**

**ARGUMENT1 > 0 ; ARGUMENT1 ≠ 1 ; ARGUMENT2 > 0**

## LN

La fonction « **LN** » permet de calculer un logarithme népérien.

**LN (ARGUMENT) = LOG (e, ARGUMENT)**

**ARGUMENT > 0**

```
SQL> SELECT POWER( 2.71828183, 10) "Power", EXP(10) "Exp",
2      LN(22026.4658) "Ln", LN(10) "Ln", LOG(2.71828183,10) "Log" FROM DUAL
```

Power	Exp	Ln	Ln	Log
22026,4659	22026,4658	10	2,30258509	2,30258509

**SIN, COS, TAN**

La fonction « **SIN** », « **COS** », « **TAN** » permet de calculer le sinus, cosinus et la tangente et renvoie la valeur trigonométrique standard d'un angle exprimée en radians (degrés multipliés par  $\pi$  et divisés par 180).

**ASIN, ACOS, ATAN**

La fonction « **ASIN** », « **ACOS** », « **ATAN** » permet de calculer l'arc sinus, cosinus et tangente d'un angle exprimé en radians (degrés multipliés par  $\pi$  et divisés par 180).



```
SQL> SELECT 30*3.141593/180 "Angle(30*pi/180)",
2      SIN(30*3.141593/180) "SIN(Angle)",
3      ASIN(SIN(30*3.141593/180)) "ASIN(sin(Angle))" FROM DUAL;
```

Angle(30*pi/180)	SIN(Angle)	ASIN(sin(Angle))
,523598833	,50000005	,523598833

**SINH, COSH, TANH**

La fonction « **SINH** », « **COSH** », « **TANH** » permet de calculer le sinus, cosinus et tangente hyperbolique.



```
SQL> SELECT (EXP(10)-EXP(-10))/2, SINH(10), (EXP(10)+EXP(-10))/2, COSH(10)
2      FROM DUAL;
```

(EXP(10)-EXP(-10))/2	SINH(10)	(EXP(10)+EXP(-10))/2	COSH(10)
11013,2329	11013,2329	11013,2329	11013,2329

## Fonctions d'arrondis

**ABS**

La fonction « **ABS** » permet de calculer la valeur absolue de l'argument.



```
SQL> SELECT ABS(-10),ABS(0),ABS(1) FROM DUAL;
```

ABS(-10)	ABS(0)	ABS(1)
10	0	1

**SIGN**

La fonction « **SIGN** » permet de calculer la signe de l'argument.

**SIGN(ARGUMENT)**

La valeur de retour est : 1 si ARGUMENT > 0, -1 si ARGUMENT < 0 et 0 si ARGUMENT = 0.



```
SQL> SELECT SIGN(-10),SIGN(0),SIGN(20) FROM DUAL;
```

SIGN(-10)	SIGN(0)	SIGN(20)
-1	0	1

## CEIL

La fonction « **CEIL** » permet de calculer le plus petit entier supérieur ou égal à l'argument.



```
SQL> SELECT CEIL(-10.23),CEIL(0),CEIL(20.23) FROM DUAL;
```

CEIL(-10.23)	CEIL(0)	CEIL(20.23)
-10	0	21

## FLOOR

La fonction « **FLOOR** » permet de calculer le plus grand entier inférieur à l'argument.



```
SQL> SELECT FLOOR(-10.23),FLOOR(0),FLOOR(20.23) FROM DUAL;
```

FLOOR(-10.23)	FLOOR(0)	FLOOR(20.23)
-11	0	20

## ROUND

La fonction « **ROUND** » permet de calculer une valeur arrondie avec une précision donnée.

**ROUND (ARGUMENT , PRECISION)**

### PRECISION

La précision peut prendre trois types de valeurs :

1. positive ; alors elle détermine le nombre de décimales à conserver,
2. zéro, est la valeur par défaut ; on ne conserve pas de décimales,
3. négative ; alors l'arrondi se fait sur les valeurs entières.

La fonction arrondi à la valeur supérieure si la décimale est supérieure ou égale à 5.



```
SQL> SELECT ROUND(-10.2326,2),ROUND(10.2366,3),ROUND(-10.2326) FROM DUAL;
```

ROUND(-10.2326,2)	ROUND(10.2366,3)	ROUND(-10.2326)
-10,23	10,237	-10

```
SQL> SELECT ROUND(102356,-2),ROUND(102326,-3),ROUND(102326) FROM DUAL;
```

ROUND(102356,-2)	ROUND(102326,-3)	ROUND(102326)
102400	102000	102326

```
SQL> SELECT ROUND(-10.2326,2f),ROUND(10.2366,3f),ROUND(-10.2326f)
2 FROM DUAL;
```

ROUND(-10.2326,2F)	ROUND(10.2366,3F)	ROUND(-10.2326F)
-10,23	10,237	-1,0E+001

```
SQL> SELECT ROUND(-10.2326,2d),ROUND(10.2366,3d),ROUND(-10.2326d)
```

```

2 FROM DUAL;

ROUND(-10.2326,2D) ROUND(10.2366,3D) ROUND(-10.2326D)
-----
-10,23          10,237          -1,0E+001

```

## TRUNC

La fonction « **TRUNC** » permet de calculer une valeur tronquée à la précision indiquée.

**TRUNC (ARGUMENT, PRECISION)**



```

SQL> SELECT TRUNC(10.2326,2),TRUNC(10.2376,2),
2          TRUNC(102826,-3),TRUNC(10.2326) FROM DUAL;

TRUNC(10.2326,2) TRUNC(10.2376,2) TRUNC(102826,-3) TRUNC(10.2326)
-----
10,23          10,23          102000          10

SQL> SELECT TRUNC(10.2326f,2),TRUNC(10.2376f,2),TRUNC(102826f,-3)
2 FROM DUAL;

TRUNC(10.2326F,2) TRUNC(10.2376F,2) TRUNC(102826F,-3)
-----
10,23          10,23          102000

SQL> SELECT TRUNC(10.2326d,2),TRUNC(10.2376d,2),TRUNC(102826d,-3)
2 FROM DUAL;

TRUNC(10.2326D,2) TRUNC(10.2376D,2) TRUNC(102826D,-3)
-----
10,23          10,23          102000

```

### Note



Les fonctions d'arrondis acceptent comme arguments des valeurs numériques mais également des nombres réels à virgule flottante.

Ils effectuent l'arrondi en respectant les mêmes règles que pour les types numériques classiques.

## Les zones horaires

SQL\*Plus et SQL reconnaissent les colonnes de type DATE, et comprennent les instructions qui permettent d'effectuer des calculs sur des valeurs de ce type.

Toutefois, étant donné que les dates Oracle peuvent inclure des heures, des minutes et des secondes, ces calculs particuliers peuvent se révéler complexes ;

Après les fonctions de manipulation des chaînes de caractères et les fonctions arithmétiques on va découvrir à présent les fonctions de manipulation de dates.

L'architecture de la base de données et l'interaction avec l'infrastructure réseau vous permettent d'avoir des clients qui se connectent de n'importe où dans le monde. C'est assez pratique quand on veut développer une application internationale, notamment pour les sites web où les internautes viennent de toute la planète.

Chaque session peut donc se connecter avec une synchronisation de fuseau horaire différente. Le serveur de base de données lui-même à une référence par rapport à son fuseau horaire.

## DBTIMEZONE

Une pseudo-colonne qui indique le fuseau horaire du serveur, sous la forme d'un décalage par rapport à l'heure universelle **UTC** (Universal Time Coordinated), anciennement **GMT** (Greenwich Meridian Time).

Le format de « **DBTIMEZONE** » est :

« {+|-}HH:MI »

C'est un paramètre de base de données spécifié à la création de la base de données et il peut être modifié par



```
SQL> SELECT DBTIMEZONE FROM DUAL;
```

```
DBTIME
```

```
-----
```

```
+02:00
```

## SESSIONTIMEZONE

Une pseudo-colonne qui indique le fuseau horaire de la session, sous la forme d'un décalage par rapport à l'heure universelle **UTC** (Universal Time Coordinated).

Le format de « **SESSIONTIMEZONE** » est :

« {+|-}HH:MI »

C'est un paramètre de la session qui peut être modifié à l'aide de la commande :

```
ALTER SESSION SET TIME_ZONE = PARAMETRE ;
```

**PARAMETRE**

La zone horaire peut être exprimée de deux manières :

- Un décalage par rapport à l'heure universelle **UTC**. La plage de valeurs valides pour « {+|-}HH:MI » s'étend de « -12:00 » à « +14:00 ».
- Un nom d'une zone horaire.

Pour obtenir une liste des noms de zones disponibles ou les abréviations de ces zones horaires vous devez interroger la vue « **V\$TIMEZONE\_NAMES** ».



```
SQL> SELECT TZNAME, TZABBREV FROM V$TIMEZONE_NAMES
2 WHERE TZNAME LIKE 'Europe/%';
```

```
TZNAME
```

```
-----
```

```
Europe/Andorra
```

```
Europe/Bratislava
```

```
Europe/Dublin
```

```
Europe/Jersey
```

```
Europe/Bucharest
```

```
Europe/Samara
```

```
Europe/San_Marino
```

```
Europe/Luxembourg
```

```
Europe/Paris
```

```
Europe/Tallinn
```

```
...
```

```
SQL> SELECT SESSIONTIMEZONE, DBTIMEZONE FROM DUAL;
```

```
SESSIONTIMEZONE
```

```
DBTIMEZONE
```

```
-----
```

```
+01:00          +01:00

SQL> ALTER SESSION SET TIME_ZONE = 'Europe/Paris';

Session modifiée.

SQL> SELECT SESSIONTIMEZONE, DBTIMEZONE FROM DUAL;

SESSIONTIMEZONE          DBTIMEZONE
-----
Europe/Paris              +01:00

SQL> ALTER SESSION SET TIME_ZONE = 'America/New_York';

Session modifiée.

SQL> SELECT SESSIONTIMEZONE, DBTIMEZONE FROM DUAL;

SESSIONTIMEZONE          DBTIMEZONE
-----
America/New_York          +01:00

SQL> ALTER SESSION SET TIME_ZONE = '+10:00';

Session modifiée.

SQL> SELECT SESSIONTIMEZONE, DBTIMEZONE FROM DUAL;

SESSIONTIMEZONE          DBTIMEZONE
-----
+10:00                    +01:00
```

## Types date

---

### **DATE**

Champ de longueur fixe de 7 octets utilisé pour stocker n'importe quelle date, incluant l'heure. La valeur d'une date est comprise entre 01/01/4712 avant JC et 31/12/9999 après JC.

### **TIMESTAMP [ (P) ]**

Champ de type date, incluant des fractions de seconde, et se fondant sur la valeur d'horloge du système d'exploitation. Une valeur de précision **P** un entier de 0 à 9 (6 étant la précision par défaut) - permet de choisir le nombre de chiffres voulus dans la partie décimale des secondes.

### **TIMESTAMP [ (P) ] WITH TIME ZONE**

Champ de type « **TIMESTAMP** » avec un paramètre de zone horaire associé. La zone horaire peut être exprimée sous la forme d'un décalage par rapport à l'heure universelle **UTC** (Universal Coordinated Time) sous la forme « **{+|-}HH :MI** », tel que "-5:0", ou d'un nom de zone, tel que "US/Pacific".

### **TIMESTAMP [ (P) ] WITH LOCAL TIME ZONE**

Champ de type « **TIMESTAMP WITH TIME ZONE** » sauf que la date est ajustée par rapport à la zone horaire de la base de données lorsqu'elle est stockée, puis adaptée à celle du client lorsqu'elle est extraite.



```
SQL> SELECT SESSIONTIMEZONE, DBTIMEZONE FROM DUAL;
```

```
SESSIONTIMEZONE  DBTIMEZONE
-----
+01:00           +01:00
```

```
SQL> DESC DATE_HEURE
```

Nom	NULL ?	Type
ID		NUMBER(2)
C_DATE		DATE
C_TS		TIMESTAMP(6)
C_TS_WTZ		TIMESTAMP(6) WITH TIME ZONE
C_TS_WLTZ		TIMESTAMP(6) WITH LOCAL TIME ZONE
DESCRIPT		VARCHAR2(100)

```
SQL> INSERT INTO DATE_HEURE VALUES
```

```
2 ( 1, SYSDATE, SYSDATE, SYSDATE, SYSDATE, SESSIONTIMEZONE);
```

1 ligne créée.

```
SQL> SELECT * FROM DATE_HEURE
```

ID	C_DATE	C_TS	C_TS_WTZ	C_TS_WLTZ	DESCRIPT
1	14/02/2011	14/02/2011 10:44:20,000000	14/02/2011 10:44:20,000000 +01:00	14/02/2011 10:44:20,000000	+01:00

Dans l'exemple précédent, vous pouvez observer la création d'un enregistrement avec la même information, la pseudo-colonne « **SYSDATE** ». Vous remarquerez que les trois types « **TIMESTAMP** » ont la même valeur et que la colonne de type « **TIMESTAMP WITH TIME ZONE** » fournit la zone horaire du client qui l'a saisie.

### Note

Il faut noter que la pseudo-colonne « **SYSDATE** » retourne automatiquement la date et l'heure du système d'exploitation dans lequel le serveur de base de données est installé.



```
SQL> ALTER SESSION SET TIME_ZONE = 'America/New_York';
```

Session modifiée.

```
SQL> INSERT INTO DATE_HEURE VALUES
```

```
2 ( 2, SYSDATE, SYSDATE, SYSDATE, SYSDATE, SESSIONTIMEZONE);
```





```

1 ligne créée.

SQL> SELECT * FROM DATE_HEURE

      ID C_DATE
-----
C_TS
-----
C_TS_WTZ
-----
C_TS_WLTZ
-----
DESCRIPT
-----
      1 14/02/2011
14/02/2011 10:44:20,000000
14/02/2011 10:44:20,000000 +01:00
14/02/2011 04:44:20,000000
+01:00

      2 14/02/2011
14/02/2011 10:45:56,000000
14/02/2011 10:45:56,000000 AMERICA/NEW_YORK
14/02/2011 10:45:56,000000
America/New_York

```

Après le changement de la zone horaire du client par la commande SQL « **ALTER SESSION** » et la création d'un nouvel enregistrement vous pouvez voir les valeurs insérées dans la table.

Le premier enregistrement est identique à première interrogation sauf pour la colonne de type « **TIMESTAMP WITH LOCAL TIME ZONE** » qui convertit l'heure suivant la zone horaire du client.

Le deuxième enregistrement consigne, pour la colonne de type « **TIMESTAMP WITH TIME ZONE** », la zone horaire du client.



```

SQL> ALTER SESSION SET TIME_ZONE = 'Australia/Sydney';

Session modifiée.

SQL> SELECT C_DATE, C_TS, C_TS_WTZ, C_TS_WLTZ, DESCRIPT FROM DATE_HEURE;

      ID C_DATE
-----
C_TS
-----
C_TS_WTZ
-----
C_TS_WLTZ
-----
DESCRIPT
-----
      1 14/02/2011
14/02/2011 10:44:20,000000
14/02/2011 10:44:20,000000 +01:00
14/02/2011 20:44:20,000000
+01:00

```

```

2 14/02/2011
14/02/2011 10:45:56,000000
14/02/2011 10:45:56,000000 AMERICA/NEW_YORK
15/02/2011 02:45:56,000000
America/New_York

```

## Les conversions

### SYS\_EXTRACT\_UTC

La fonction « **SYS\_EXTRACT\_UTC** » permet de convertir une date et heure d'une zone horaire spécifique en date et heure au format UTC c'est-à-dire à l'heure de Greenwich.

**ARGUMENT**

La valeur d'une date avec les informations concernant la zone horaire spécifique.



```

SQL> SELECT C_TS, SYS_EXTRACT_UTC(C_TS) SE_C_TS,
2 SYS_EXTRACT_UTC(C_TS_WTZ) SE_C_TS_WTZ,
3 SYS_EXTRACT_UTC(C_TS_WLTZ) SE_C_TS_WLTZ FROM DATE_HEURE;

```

C\_TS

SE\_C\_TS

SE\_C\_TS\_WTZ

SE\_C\_TS\_WLTZ

```

14/02/2011 10:44:20,000000
14/02/2011 09:44:20,000000
14/02/2011 09:44:20,000000
14/02/2011 09:44:20,000000

```

```

14/02/2011 10:45:56,000000
14/02/2011 09:45:56,000000
14/02/2011 15:45:56,000000
14/02/2011 15:45:56,000000

```

### TZ\_OFFSET

La fonction « **TZ\_OFFSET** » permet de connaître l'écart de fuseau horaire entre la zone passée en paramètre et UTC c'est-à-dire à l'heure de Greenwich.

**ARGUMENT**

La valeur d'une date avec les informations concernant la zone horaire spécifique.



```

SQL> SELECT TZ_OFFSET('America/Guadeloupe') FROM DUAL;

```

TZ\_OFFS

-04:00

```

SQL> SELECT TZ_OFFSET('Australia/Melbourne') FROM DUAL;

```

TZ\_OFFS

# Les dates système

## CURRENT\_DATE

La fonction « **CURRENT\_DATE** » permet de connaître la date et l'heure actuelle en prenant en compte le paramétrage de la zone horaire « **TIME\_ZONE** » de la session. La pseudo-colonne « **SYSDATE** » retourne automatiquement la date et l'heure du système d'exploitation dans lequel le serveur de base de données est installé.



```
SQL> SELECT SESSIONTIMEZONE, DBTIMEZONE, CURRENT_DATE, SYSDATE
2 FROM DUAL;
```

SESSIONTIMEZONE	DBTIMEZONE	CURRENT_DATE	SYSDATE
Europe/Paris	+01:00	14/02/2011	14/02/2011

```
SQL> SELECT TZ_OFFSET('Pacific/Kiritimati') FROM DUAL;
```

```
TZ_OFFSET
-----
+14:00
```

```
SQL> ALTER SESSION SET TIME_ZONE='Pacific/Kiritimati';
```

Session modifiée.

```
SQL> SELECT SESSIONTIMEZONE, DBTIMEZONE, CURRENT_DATE, SYSDATE
2 FROM DUAL;
```

SESSIONTIMEZONE	DBTIMEZONE	CURRENT_DATE	SYSDATE
Pacific/Kiritimati	+01:00	15/02/2011	14/02/2011

## CURRENT\_TIMESTAMP

La fonction « **CURRENT\_TIMESTAMP** » permet de connaître la date et l'heure relative à la plage horaire de la session.

### ARGUMENT

La valeur de la précision, un entier de 0 à 9 (6 étant la précision par défaut).

### VALEUR

La valeur de retour est de type  
« **TIMESTAMP WITH TIME ZONE** »



```
SQL> ALTER SESSION SET TIME_ZONE='Europe/Paris';
```

Session modifiée.

```
SQL> SELECT SESSIONTIMEZONE, DBTIMEZONE,
2 CURRENT_TIMESTAMP(2), SYSDATE FROM DUAL;
```

SESSIONTIMEZONE	DBTIMEZONE	CURRENT_TIMESTAMP(2)	SYSDATE
Europe/Paris	+01:00	14/02/2011 11:33:35,28	EUROPE/PARIS 14/02/2011

## LOCALTIMESTAMP

La fonction « **LOCALTIMESTAMP** » est identique à la fonction « **CURRENT\_TIMESTAMP** », sauf par sa valeur de retour qui est un type « **TIMESTAMP** ».

## SYSTIMESTAMP

La fonction « **LOCALTIMESTAMP** » permet de connaître la date et l'heure, y compris les fractions de secondes, en s'appuyant sur la zone horaire configurée sur le serveur de base de données.



```
SQL> ALTER SESSION SET TIME_ZONE='Europe/Paris';
```

Session modifiée.

```
SQL> SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP, SYSTIMESTAMP FROM DUAL;
```

```
CURRENT_TIMESTAMP
```

```
-----
```

```
LOCALTIMESTAMP
```

```
-----
```

```
SYSTIMESTAMP
```

```
-----
```

```
14/02/2011 11:36:21,027000 EUROPE/PARIS
```

```
14/02/2011 11:36:21,027000
```

```
14/02/2011 11:36:21,027000 +01:00
```

```
SQL> ALTER SESSION SET TIME_ZONE='America/Los_Angeles';
```

Session modifiée.

```
SQL> SELECT LOCALTIMESTAMP, SYSTIMESTAMP FROM DUAL;
```

```
CURRENT_TIMESTAMP
```

```
-----
```

```
LOCALTIMESTAMP
```

```
-----
```

```
SYSTIMESTAMP
```

```
-----
```

```
14/02/2011 02:37:03,183000 AMERICA/LOS_ANGELES
```

```
14/02/2011 02:37:03,183000
```

```
14/02/2011 11:37:03,183000 +01:00
```

## Types intervalle

Un intervalle spécifie une période de temps que l'on peut additionner ou soustraire à une date. Vous pouvez spécifier ces périodes en termes d'années et de mois, ou en termes de jours, heures, minutes et secondes. Oracle prend en charge deux types d'intervalles : « **YEAR TO MONTH** » et « **DAY TO SECOND** ».

### INTERVAL YEAR [(P)] TO MONTH

Il représente un intervalle de temps exprimé en années et en mois. C'est une valeur relative qui peut être utilisée pour incrémenter ou décrémenter une valeur absolue d'un type date.

**P**

Un littéral entier entre 0 et 9 devant être utilisé pour spécifier le nombre de chiffres acceptés pour représenter les années (2 étant la valeur par défaut).

Vous pouvez saisir un intervalle de type « **YEAR TO MONTH** » dans une requête à l'aide de la syntaxe suivante :

**INTERVAL** ' [+|-] [a] - [m] ' [YEAR [(P)]] [TO MONTH]

**a**

Un entier représentant les années de l'intervalle ; il doit être inférieur à la précision.

**m**

Un entier représentant les mois de l'intervalle.



```
SQL> DESC I_YTOM
```

Nom	NULL ?	Type
C_I_YTOM		INTERVAL YEAR(9) TO MONTH

```
SQL> INSERT INTO I_YTOM VALUES( INTERVAL '2' YEAR);
SQL> INSERT INTO I_YTOM VALUES( INTERVAL '6' MONTH);
SQL> INSERT INTO I_YTOM VALUES( INTERVAL '16' MONTH);
SQL> INSERT INTO I_YTOM VALUES( INTERVAL '1-4' YEAR TO MONTH);
SQL> INSERT INTO I_YTOM VALUES( INTERVAL '0-3' YEAR TO MONTH);
SQL> INSERT INTO I_YTOM VALUES( INTERVAL '-1-6' YEAR(2) TO MONTH);
SQL> INSERT INTO I_YTOM VALUES( '1-4');
SQL> INSERT INTO I_YTOM VALUES( '-5-4');
SQL> SELECT SYSDATE, C_I_YTOM, SYSDATE + C_I_YTOM, SYSDATE - C_I_YTOM
2 FROM I_YTOM;
```

SYSDATE	C_I_YTOM	SYSDATE+C_I_YTOM	SYSDATE-C_I_YTOM
14/02/2011 14:21	+0000000002-00	14/02/2013 14:21	14/02/2009 14:21
14/02/2011 14:21	+0000000000-06	14/08/2011 14:21	14/08/2010 14:21
14/02/2011 14:21	+0000000001-04	14/06/2012 14:21	14/10/2009 14:21
14/02/2011 14:21	+0000000001-04	14/06/2012 14:21	14/10/2009 14:21
14/02/2011 14:21	+0000000000-03	14/05/2011 14:21	14/11/2010 14:21
14/02/2011 14:21	-0000000001-06	14/08/2009 14:21	14/08/2012 14:21
14/02/2011 14:21	+0000000001-04	14/06/2012 14:21	14/10/2009 14:21
14/02/2011 14:21	-0000000005-04	14/10/2005 14:21	14/06/2016 14:21

L'exemple ci-dessus vous montre l'insertion de plusieurs enregistrements dans la table DATE\_INTERVAL. Vous pouvez également observer l'utilisation des valeurs de la colonne « **C\_I\_YTOM** » pour incrémenter et décrémenter la date du jour.

**INTERVAL DAY [(P)] TO SECOND [(P)]**

Il représente un intervalle de temps exprimé en jours, heures, minutes et secondes. C'est une valeur relative qui peut être utilisée pour incrémenter ou décrémenter une valeur absolue d'un type date.

**P**

Un littéral entier entre 0 et 9 devant être utilisé pour spécifier le nombre de chiffres acceptés pour représenter les jours et les fractions de secondes (2 et 6 étant respectivement les valeurs par défaut).

Vous pouvez saisir un intervalle de type « **DAY TO SECOND** » dans une requête à l'aide de la syntaxe suivante :

**INTERVAL** ' [+|-] [d] [h[:m[:s]]]' [DAY [(P)]]  
[TO HOUR | MINUTE | SECOND[(P)]]

<b>d</b>	Un entier représentant les jours ; il doit être inférieur à la précision.
<b>h</b>	Un entier représentant les heures ; si vous voulez saisir cette option, vous devez inclure « <b>TO HOUR</b> » dans la description.
<b>m</b>	Un entier représentant les minutes ; si vous voulez saisir cette option, vous devez inclure « <b>TO MINUTE</b> » dans la description.
<b>s</b>	Une valeur représentant les secondes ; si vous voulez saisir cette option vous devez inclure « <b>TO SECOND</b> » dans la description.



```
SQL> DESC DATE_INTERVAL
```

Nom	NULL ?	Type
C_I_DTOS		INTERVAL DAY(9) TO SECOND(9)

```
SQL> INSERT INTO I_DTOS VALUES( INTERVAL '6' DAY);
SQL> INSERT INTO I_DTOS VALUES( INTERVAL '6' HOUR);
SQL> INSERT INTO I_DTOS VALUES( INTERVAL '6' MINUTE);
SQL> INSERT INTO I_DTOS VALUES( INTERVAL '6' SECOND);
SQL> INSERT INTO I_DTOS VALUES( INTERVAL '1 2' DAY TO HOUR);
SQL> INSERT INTO I_DTOS VALUES( INTERVAL '1 2:30' DAY TO MINUTE);
SQL> INSERT INTO I_DTOS VALUES( INTERVAL '1 2:30:30' DAY TO SECOND);
SQL> INSERT INTO I_DTOS VALUES( '-8 0:30:45');
SQL> SELECT SYSDATE, C_I_DTOS, SYSDATE + C_I_DTOS FROM I_DTOS;
```

SYSDATE	C_I_DTOS	SYSDATE+C_I_DTOS
14/02/2011 15:53	+0000000006 00:00:00.000000000	20/02/2011 15:53
14/02/2011 15:53	+0000000000 06:00:00.000000000	14/02/2011 21:53
14/02/2011 15:53	+0000000000 00:06:00.000000000	14/02/2011 15:59
14/02/2011 15:53	+0000000000 00:00:06.000000000	14/02/2011 15:53
14/02/2011 15:53	+0000000001 02:00:00.000000000	15/02/2011 17:53
14/02/2011 15:53	+0000000001 02:30:00.000000000	15/02/2011 18:23
14/02/2011 15:53	+0000000001 02:30:30.000000000	15/02/2011 18:23
14/02/2011 15:53	-0000000008 00:30:45.000000000	06/02/2011 15:22

Dans l'exemple précédent vous pouvez remarquer les deux types de syntaxe utilisés pour l'insertion des enregistrements de type « **INTERVAL DAY TO SECOND** ». Vous pouvez observer également l'utilisation des valeurs de la colonne « **C\_I\_DTOS** » pour incrémenter et décrémenter la date du jour.

## Manipulation des dates

### ADD\_MONTHS

La fonction « **ADD\_MONTHS** » permet d'ajouter ou soustraire un nombre de mois à une date.

**ADD\_MONTHS (DATE, ARGUMENT) = VALEUR**

<b>ARGUMENT</b>	Le nombre des mois à ajouter ou soustraire.
<b>VALEUR</b>	La valeur de retour est de type date.



```
SQL> SELECT SYSDATE, ADD_MONTHS(SYSDATE,6) FROM DUAL;
```

```
SYSDATE      ADD_MONTHS
-----
14/02/2011  14/08/2011
```

```
SQL> SELECT ADD_MONTHS('31/01/2011',1) FROM DUAL;
```

```
ADD_MONTHS
-----
28/02/2011
```

### Attention



Comme vous pouvez le constater dans l'exemple précédent, la fonction « **ADD\_MONTHS** » effectue un arrondi sur le dernier jour du mois si le mois cible ne comporte pas la date demandée. En conséquence, il convient de faire attention quand on utilise cette fonction pour le dernier jour du mois.

## MONTHS\_BETWEEN

La fonction « **MONTHS\_BETWEEN** » permet de trouver le nombre de mois qui séparent deux dates.

**MONTHS\_BETWEEN**(DATE1,DATE2) = VALEUR

**VALEUR**

La différence entre DATE1 et DATE2 exprimé en nombre de mois; le résultat peut être un nombre décimal. La partie fractionnaire du résultat est calculée en considérant chaque jour comme égal à 1/31 de mois.



```
SQL> SELECT SYSDATE + 35, SYSDATE, 35/31,
2          MONTHS_BETWEEN(SYSDATE + 35,SYSDATE) MB1,
3          MONTHS_BETWEEN('28/08/2011','28/07/2011') MB2
4 FROM DUAL;
```

```
SYSDATE+35  SYSDATE      35/31      MB1      MB2
-----
21/03/2011  14/02/2011  1,12903226  1,22580645  1
```

## Retrouver une date

### LAST\_DAY

La fonction « **LAST\_DAY** » permet de trouver la date du dernier jour du mois qui contient celle qui est passée en argument.



```
SQL> SELECT SYSDATE,
2          LAST_DAY(SYSDATE) ,
3          LAST_DAY(ADD_MONTHS(SYSDATE,1))
4 FROM DUAL;
```

```
SYSDATE      LAST_DAY(S LAST_DAY(A
-----
21/04/2006  30/04/2006  31/05/2006
```

### NEXT\_DAY

La fonction « **NEXT\_DAY** » permet de trouver la date du prochain jour de la semaine spécifié.

**NEXT\_DAY (DATE, JOUR\_SEMAINE) = VALEUR**

**JOUR\_SEMAINE**

Une chaîne de caractère qui indique le jour de la semaine ('Lundi', 'Mardi', etc.). La valeur du jour de la semaine doit être saisie dans la langue de la session courante.

**VALEUR**

La valeur de retour est de type date.



```
SQL> SELECT SYSDATE,
2         NEXT_DAY (SYSDATE, 'Lundi') "Lundi",
3         NEXT_DAY (SYSDATE, 'Mardi') "Mardi",
4         NEXT_DAY (SYSDATE, 'Mercredi') "Mercredi",
5         NEXT_DAY (SYSDATE, 'Jeudi') "Jeudi",
6         NEXT_DAY (SYSDATE, 'Vendredi') "Vendredi",
7         NEXT_DAY (SYSDATE, 'Samedi') "Samedi",
8         NEXT_DAY (SYSDATE, 'Dimanche') "Dimanche"
9 FROM DUAL;
```

SYSDATE	Lundi	Mardi	Mercredi	Jeudi	Vendredi
Samedi	Dimanche				
14/02/2011	21/02/2011	15/02/2011	16/02/2011	17/02/2011	18/02/2011
19/02/2011	20/02/2011				

## Arrondis des dates

### ROUND

La fonction « **ROUND** » permet de calculer l'arrondi d'une date selon une précision spécifiée.

**ROUND (ARGUMENT, PRECISION)**

**PRECISION**

La précision est indiquée en utilisant un des masques de mise en forme de la date. On peut ainsi arrondir une date à l'année, au mois, à la minute, etc. Par défaut, la précision est le jour.

<i>Format</i>	<i>Précision pour ROUND et TRUNC</i>
CC, SCC	Le siècle
YYYY, YYYY, YEAR, SYEAR, YYY, YY, Y	Année
IYYY, IY, IY, I	Année ISO
Q	Le numéro du trimestre
MONTH, MON, MM, RM	Le mois
WW	Numéro de la semaine dans l'année
IW	Semaine de l'année selon le standard ISO
W	Numéro de la semaine dans le mois
DDD	Numéro de jour dans l'année, de 1 à 366



DD	Numéro de jour dans le mois, de 1 à 31
J	Numéro de jour de la semaine, de 1 à 7
DAY, DY, D	Le jour de la semaine
HH, HH12, HH24	Heure
MI	Minute

Sans l'argument format, cette fonction arrondit la valeur de date à 12 A.M. (minuit, le début du jour concerné) si la date est située avant midi ; sinon, la fonction arrondit la date au jour suivant. L'emploi d'un format est étudié plus loin dans ce module.



```
SQL> SELECT NOM, SYSDATE-DATE_NAISSANCE "Date heure",
2 ROUND(SYSDATE)-DATE_NAISSANCE "Date" FROM EMPLOYES;
```

NOM	Date heure	Date
Berlioz	15831,7076	15832
Nocella	12505,7076	12506
Herve	11471,7076	11472
Mangeard	11372,7076	11373
Cazade	11471,7076	11472
Devie	15665,7076	15666
Peacock	15306,7076	15307
Idesheim	10365,7076	10366
Rollet	9473,70759	9474
Silberreiss	11868,7076	11869
Weiss	12090,7076	12091
Delorgue	10810,7076	10811
...		

```
SQL> SELECT SYSDATE, ROUND(SYSDATE,'YEAR') "Année",
2 ROUND(SYSDATE,'MONTH') "Mois", ROUND(SYSDATE,'Q') "Trimestre",
3 SYSDATE+200, ROUND(SYSDATE+200,'YEAR') "Année",
4 ROUND(SYSDATE+200,'MONTH') "Mois", ROUND(SYSDATE+200,'Q') "Trimestre"
5 FROM DUAL ;
```

SYSDATE	Année	Mois	Trimestre	SYSDATE+20	Année
14/02/2011	01/01/2011	01/02/2011	01/01/2011	02/09/2011	01/01/2012
01/09/2011	01/10/2011				

## TRUNC

La fonction « **TRUNC** » permet de calculer une valeur tronquée d'une date selon une précision spécifiée.

**TRUNC (ARGUMENT , PRECISION)**



```
SQL> SELECT SYSDATE, TRUNC(SYSDATE,'YEAR') "Année",
2 TRUNC(SYSDATE,'MONTH') "Mois", TRUNC(SYSDATE,'Q') "Trimestre",
3 SYSDATE+200, TRUNC(SYSDATE+200,'YEAR') "Année",
7 TRUNC(SYSDATE+200,'MONTH') "Mois", TRUNC(SYSDATE+200,'Q') "Trimestre"
8 FROM DUAL ;
```

SYSDATE	Année	Mois	Trimestre	SYSDATE+20	Année
-----	-----	-----	-----	-----	-----
Mois	Trimestre				
-----	-----				
14/02/2011	01/01/2011	01/02/2011	01/01/2011	02/09/2011	01/01/2011
01/09/2011	01/07/2011				

# 5

- *Manipulation de chaînes*
- *Manipulation de dates*
- *Conversions*
- *DECODE*
- *CASE*

## Les conversions SQL



### Objectifs

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- Manipuler les données et effectuer les conversions implicites.
- Effectuer des conversions entre les différents types de données.
- Manipuler les expressions de type date.
- Manipuler les expressions de type chaîne de caractères.



### Contenu

Opérateurs	5-2	Extraire des informations	5-9
Les conversions implicites	5-3	Chaîne vers autres types	5-10
Les fonctions de conversion	5-3	Numérique vers interval	5-13
Numérique vers chaîne	5-5	Les fonctions générales	5-14
Date vers chaîne	5-7		

# Opérateurs

Une expression de type date est une combinaison de noms de colonnes, de constantes et de fonctions de manipulation de date combinés au moyen des **opérateurs** addition « + », soustraction « - », multiplication « \* » ou division « / ».

Opérande / Opérande		DATE	TIMESTAMP	INTERVAL	Number
	Opérateur				
DATE	+	—	—	DATE	DATE
	-	DATE	DATE	DATE	DATE
TIMESTAMP	+	—	—	TIMESTAMP	—
	-	INTERVAL	INTERVAL	TIMESTAMP	TIMESTAMP
INTERVAL	+	DATE	TIMESTAMP	INTERVAL	—
	-	—	—	INTERVAL	—
	*	—	—	—	INTERVAL
	/	—	—	—	INTERVAL
Number	+	DATE	DATE	—	NA
	-	—	—	—	NA
	*	—	—	INTERVAL	NA
	/	—	—	—	NA

Dans le tableau précédent vous pouvez voir une présentation des opérations possibles entre toutes les données de type date.

L'opération doit être lue de la manière suivante :

« OPERANDE » « OPERATEUR » « OPERANDE » = « RESULTAT »

Dans l'exemple ci-après, vous pouvez remarquer l'utilisation de l'opérateur de multiplication avec un type de donnée « **INTERVAL DAY TO SECOND** ».



```
SQL> SELECT C_I_YTOM, C_I_YTOM * 4 FROM I_YTOM;
```

```
C_I_YTOM          C_I_YTOM*4
-----
+0000000002-00 +0000000008-00
+0000000000-06 +0000000002-00
+0000000001-04 +0000000005-04
+0000000001-04 +0000000005-04
...
```

```
SQL> SELECT C_I_DTOS, C_I_DTOS / 2 FROM I_DTOS;
```

```
C_I_DTOS          C_I_DTOS/2
-----
+0000000006 00:00:00.000000000 +0000000003 00:00:00.000000000
+0000000000 06:00:00.000000000 +0000000000 03:00:00.000000000
+0000000000 00:06:00.000000000 +0000000000 00:03:00.000000000
+0000000000 00:00:06.000000000 +0000000000 00:00:03.000000000
+0000000001 02:00:00.000000000 +0000000000 13:00:00.000000000
...
```

# Les conversions implicites

Le langage SQL détecte la nécessité d'une conversion ; il essaie de changer les valeurs de manière à pouvoir effectuer l'opération requise. Le tableau de l'image présente les types de conversions implicites effectuées par le langage SQL.

De / Vers	CHAR	VARCHAR2	NCHAR	NVARCHAR2	DATE	DATETIME / INTERVAL	NUMBER	BINARY_FLOAT	BINARY_DOUBLE
CHAR	✗	✓	✓	✓	✓	✓	✓	✓	✓
VARCHAR2	✓	✗	✓	✓	✓	✓	✓	✓	✓
NCHAR	✓	✓	✗	✓	✓	✓	✓	✓	✓
NVARCHAR2	✓	✓	✓	✗	✓	✓	✓	✓	✓
DATE	✓	✓	✓	✓	✗	✗	✗	✗	✗
DATETIME/INTERVAL	✓	✓	✓	✓	✗	✗	✗	✗	✗
NUMBER	✓	✓	✓	✓	✗	✗	✗	✓	✓
BINARY_FLOAT	✓	✓	✓	✓	✗	✗	✓	✗	✓
BINARY_DOUBLE	✓	✓	✓	✓	✗	✗	✓	✓	✗

Avec les conversions implicites vous pouvez spécifier des valeurs littérales à la place de données au format interne adéquat ; le langage SQL les transformera si nécessaire.



```
SQL> SELECT RPAD(NOM,15) || ' ' || DATE_NAISSANCE || ' ' || SALAIRE * '1,1'
2 "Employe" FROM EMPLOYES;
```

Employe

```
-----
Berlioz      12/10/1967 10340
Nocella      19/11/1976 8360
Herve        19/09/1979 7370
Mangeard     27/12/1979 9020
Cazade       19/09/1979 7920
Devie        26/03/1968 1694
Peacock      20/03/1969 6710
Idesheim     29/09/1982 8030
Rollet       09/03/1985 5500
Silberreiss  18/08/1978 10120
...
```

## Conseil



Bien que le moteur du SGBDR qui exécute chaque ordre SQL sache prendre en compte l'évaluation de certaines expressions qui utilisent des données de types différents, il est toujours préférable de programmer des expressions homogènes, dans lesquelles les conversions de types sont clairement indiquées par utilisation de fonctions de conversion.

# Les fonctions de conversion

Le langage SQL propose de nombreuses fonctions de conversion automatique entre les types de données. Bien que le moteur du SGBDR qui exécute chaque ordre SQL sache prendre en compte

l'évaluation de certaines expressions qui utilisent des données de types différents, il est toujours préférable de programmer des expressions homogènes, dans lesquelles les conversions de types sont clairement indiquées par utilisation de fonctions de conversion.

De / Vers	CHAR VARCHAR2 NCHAR NVARCHAR2	NUMBER	DATETIME INTERVAL	BINARY_FLOAT BINARY_DOUBLE
CHAR VARCHAR2 NCHAR NVARCHAR2	TO_CHAR TO_NCHAR	TO_NUMBER	TO_DATE TO_TIMESTAMP TO_TIMESTAMP_TZ TO_YMINTERVAL TO_DSINTERVAL	TO_BINARY_FLOAT TO_BINARY_DOUBLE
NUMBER	TO_CHAR TO_NCHAR	✗	TO_DATE NUMTOYMINTERVAL NUMTODSINTERVAL	TO_BINARY_FLOAT TO_BINARY_DOUBLE
DATETIME INTERVAL	TO_CHAR TO_NCHAR	✗	✗	✗
BINARY_FLOAT	TO_CHAR TO_NCHAR	TO_NUMBER	✗	TO_BINARY_FLOAT TO_BINARY_DOUBLE
BINARY_DOUBLE	TO_CHAR TO_NCHAR	TO_NUMBER	✗	TO_BINARY_FLOAT TO_BINARY_DOUBLE

Le tableau de l'image présente les fonctions de conversion entre les différents types de données que l'on va détailler.

## CAST

La fonction « **CAST** », est un mécanisme de conversion d'un type de donnée en un autre type de donnée extrêmement souple et pratique. Cette fonction est connue des développeurs utilisant les langages orientés objet dans lesquels il est souvent nécessaire de transtyper un objet d'une classe en un objet d'une autre classe.

**CAST( EXPRESSION AS NOM\_TYPE )**

**EXPRESSION** Une expression qui doit être convertie.

**NOM\_TYPE** Le type de donnée cible.



```
SQL> SELECT CAST( DATE_NAISSANCE AS TIMESTAMP WITH TIME ZONE )
2 FROM EMPLOYES;

CAST( DATE_NAISSANCE AS TIMESTAMP WITH TIME ZONE )
-----
09/01/58 00:00:00,000000 +02:00
04/03/55 00:00:00,000000 +02:00
19/09/58 00:00:00,000000 +02:00
...
```

## CONVERT

La fonction permet de convertir une chaîne de caractères d'une page de code vers une autre. Ce type de conversion est utilisé essentiellement lorsque-vous transférez des informations d'une base de données vers une autre.

**CONVERT( chaîne, destination , source )**

**chaîne** Une expression de type chaîne de caractères qui doit être convertie.

**destination** La page de code dans laquelle doivent être converties les chaînes de caractères.

**source** La page de code source de la chaîne de caractères.

Vous pouvez interroger la vue « **V\$NLS\_PARAMETERS** » pour déterminer la page des codes pour les caractères par défaut, utilisée pour les champs de type « **CHAR** », « **VARCHAR2** » et « **CLOB** », ainsi que la page des codes pour tous les caractères nationaux, utilisée pour les champs de type « **CHAR** », « **VARCHAR2** » et « **CLOB** ».

Vous pouvez également interroger la vue « **V\$NLS\_VALID\_VALUES** » pour déterminer les pages de codes valides pour votre base de données.



```
SQL> SELECT * FROM V$NLS_PARAMETERS WHERE PARAMETER LIKE '%CHARACTERSET';
```

PARAMETER	VALUE
NLS_CHARACTERSET	WE8MSWIN1252
NLS_NCHAR_CHARACTERSET	AL16UTF16

```
SQL> SELECT VALUE FROM V$NLS_VALID_VALUES WHERE PARAMETER = 'CHARACTERSET'
2 AND ISDEPRECATED = 'FALSE' AND VALUE IN ( 'US7ASCII', 'WE8MSWIN1252',
3 'AL16UTF16','WE8ISO8859P1', 'WE8ISO8859P15');
```

VALUE
US7ASCII
WE8ISO8859P1
WE8ISO8859P15
WE8MSWIN1252
AL16UTF16

```
SQL> SELECT SOCIETE, CONVERT( SOCIETE,'US7ASCII', 'WE8MSWIN1252')
2 FROM FOURNISSEURS WHERE PAYS IN ('France','Suède');
```

SOCIETE	CONVERT(SOCIETE,'US7ASCII',
PB Knäckebröd AB	PB Knackebrod AB
Svensk Sjöföda AB	Svensk Sjofoda AB
Aux joyeux ecclésiastiques	Aux joyeux ecclesiastiques
Escargots Nouveaux	Escargots Nouveaux
Gai pâturage	Gai paturage

## Numérique vers chaîne

### TO\_CHAR, TO\_NCHAR

La fonction « **TO\_CHAR** » ou « **TO\_NCHAR** » permet de convertir un numérique, avec un certain format, en chaîne de caractères.

**TO\_CHAR (NUMBER, FORMAT)**

**NUMBER**

L'argument **NUMBER** est une expression de type :  
**NUMBER**,  
**BINARY\_FLOAT**,  
**BINARY\_DOUBLE**.

**FORMAT**

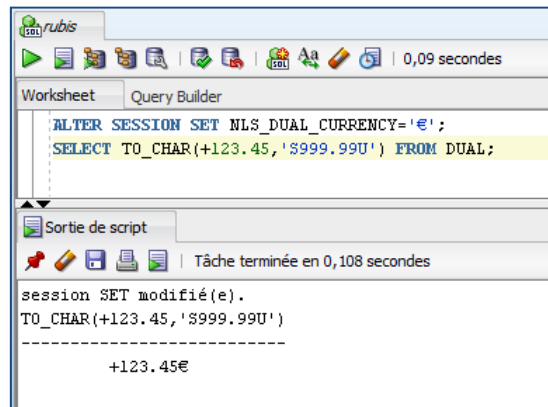
Le format (masque) pour afficher la valeur numérique. Le tableau suivant présente les options des formats pour les types numériques.

Format	Description
,	Retourne une virgule, utilisée dans certains formats comme séparateur de milliers.
.	Retourne un point comme séparateur de décimale.
\$	Retourne le symbole monétaire \$, il précédera le premier chiffre significatif.
0	Retourne un chiffre, présent même si non significatif (zéro).
9	Retourne un chiffre, non représenté dans le cas d'un zéro non significatif.
B	Le nombre sera représenté par des blancs s'il vaut zéro.
C	Retourne le symbole monétaire ISO de votre environnement de travail.
D	Retourne le symbole de séparateur de décimales de votre système. Par défaut c'est « . ».
EEEE	Retourne un nombre représenté avec un exposant (le spécifier avant MI ou PR).
FM	Retourne une valeur sans espaces à gauche ou à droite.
G	Retourne le symbole de séparateur de milliers de votre système.
L	Retourne le symbole monétaire local de votre environnement de travail.
MI	Retourne le signe négatif à droite du masque.
PR	Retourne les nombres négatifs affichés entre « < > »
RN	Retourne une valeur numérique en chiffres romains. La valeur doit être un entier compris entre 1 et 3999.
S	Retourne une valeur précédée par le signe « - ».
U	Retourne le symbole monétaire spécifié dans le paramètre de votre session « <b>NLS_DUAL_CURRENCY</b> ».
V	Retourne une valeur multipliée par 10 <sup>n</sup> , ou la valeur n égal au nombre des 9 après le caractère « <b>V</b> ».
X	Retourne une valeur en hexadécimal, si la valeur n'est pas un entier Oracle l'arrondi.

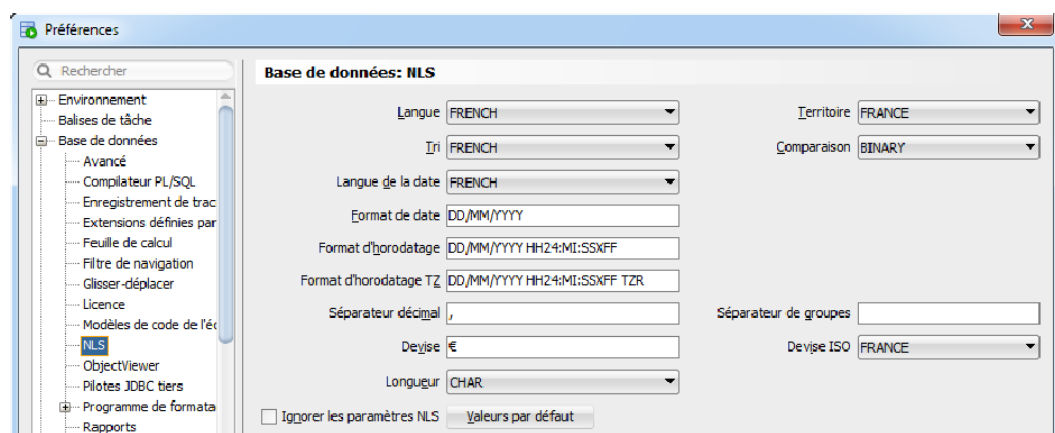
Nombre	Format	Résultat
-1234567890	9999999999S	'1234567890-'
0	99.99	' ,00 '
+0.1	99.99	' ,10 '
-0.2	99.99	' -,20 '
0	90.99	' 0,00 '
+0.1	90.99	' 0,10 '
-0.2	90.99	' -0,20 '
0	9999	' 0 '
1	9999	' 1 '
0	B9999	' '
1	B9999	' 1 '
0	B90.99	' '
+123.456	999.999	' 123,456 '
-123.456	999.999	' -123,456 '
+123.456	FM999D009	'123,456 '
+123.456	9D9EEEE	' 1,2E+02 '
+1E+123	9D9EEEE	' 1,0E+123 '
+123.456	FM9D9EEEE	'1,2E+02 '
+123.45	FM999D009	'123,45 '
+123.0	FM999.009	'123,00 '
+123.45	L999.99	' \$123,45 '
+123.45	FML999D99	'\$123,45 '
+1234567890	9G999G999G999S	'1 234 567 890+ '
+123.45	S999D99U	' +123,45€ '

Pour visualiser le paramètre « **NLS\_DUAL\_CURRENCY** », vous pouvez interroger la vue « **V\$NLS\_PARAMETERS** ». Il est également possible de modifier ce paramètre à l'aide de la commande « **ALTER SESSION SET** », mais pour **SQL\*Plus** en ligne de commande, il n'y a pas d'impact ; en revanche, pour **SQL\*Developer** le résultat est celui attendu.





Vous pouvez configurer SQL\*Developer pour que toutes les sessions ultérieures soient configurées à l'identique par choix dans le menu **Outils/Préférences** et ensuite configurer les paramètres comme suit :



## Date vers chaîne

### TO\_CHAR, TO\_NCHAR

La fonction « **TO\_CHAR** » ou « **TO\_NCHAR** » permet également de convertir une date, avec un certain format, en chaîne de caractères.

#### TO\_CHAR (DATE, FORMAT)

##### DATE

L'argument **DATE** est une expression de type :

- DATE,
- TIMESTAMP,
- TIMESTAMP WITH TIME ZONE,
- TIMESTAMP WITH LOCAL TIME ZONE.

##### FORMAT

Le format (masque) pour afficher la date numérique. Le tableau suivant présente les options des formats pour les types dates.

<i>Format</i>	<i>Description</i>
MM	Numéro du mois dans l'année
RM	Numéro du mois dans l'année en chiffres romains
MON	Le nom du mois abrégé sur trois lettres
MONTH	Le nom du mois écrit en entier
DDD	Numéro du jour dans l'année, de 1 à 366
DD	Numéro du jour dans le mois, de 1 à 31

D	Numéro du jour dans la semaine, de 1 à 7
DY	Le nom de la journée abrégé sur trois lettres
DAY	Le nom de la journée écrit en entier
YYYY	Année complète sur quatre chiffres
YYY	Les trois derniers chiffres de l'année
RR	Deux derniers chiffres de l'année de la date courante
CC	Le siècle
YEAR	Année écrite en lettres: TWO THOUSAND (option apparemment non francisée)
Q	Le numéro du trimestre
WW	Numéro de la semaine dans l'année
IW	Semaine de l'année selon le standard ISO
W	Numéro de la semaine dans le mois
J	Calendrier Julien -jours écoulés depuis le 31 décembre 4713 av. J.-C
HH	Heure du jour, toujours de format 1-12
HH24	Heure du jour, sur 24 heures
MI	Minutes écoulées dans l'heure
SS	Secondes écoulées dans une minute
SSSSS	Secondes écoulées depuis minuit, toujours 0-86399
AM, PM	Affiche AM ou PM selon qu'il s'agit du matin ou de l'après-midi
FM	Les valeurs sont renvoyées sans les caractères blanc avant ou après
TZH	Affiche l'heure du fuseau horaire
TZM	Affiche les minutes du fuseau horaire
TZR	Affiche le fuseau horaire complet

Vous pouvez insérer une chaîne de caractères dans le format pour l'afficher la constante chaîne de caractères est délimitée par le caractère « " ».



```
SQL> SELECT TO_CHAR( SYSDATE, 'D DD DDD DAY Day') FROM DUAL ;
```

```
TO_CHAR( SYSDATE, 'DDDDDDDAY
```

```
-----
5 20 140 VENDREDI Vendredi
```

```
SQL> SELECT TO_CHAR( SYSDATE, 'MM MON Mon MONTH Month') FROM DUAL ;
```

```
TO_CHAR( SYSDATE, 'MMMONMONMONTHMONT
```

```
-----
05 MAI Mai MAI Mai
```

```
SQL> SELECT TO_CHAR( SYSDATE, 'FMMM MON Mon MONTH Month') FROM DUAL ;
```

```
TO_CHAR( SYSDATE, 'FMMMONMONMONTHMONTH' )
```

```
-----
5 MAI Mai MAI Mai
```

```
SQL> SELECT TO_CHAR( SYSDATE,
2 'DD/MM/YYYY Year Q WW iW W HH:MM:SS SSSSS') FROM DUAL ;
```

```
TO_CHAR( SYSDATE, 'DD/MM/YYYYYEARQWWIWWHH:MM:SSSSSSS' )
```

```
-----
20/05/2011 Twenty Eleven 2 20 20 3 12:05:36 46056
```

```
SQL> SELECT TO_CHAR(SYSTIMESTAMP, 'DD/MM/YYYY HH24:MM:SS SSSSS') FROM DUAL ;
```

```

TO_CHAR(SYSTIMESTAMP, 'DD/
-----
20/05/2011 12:05:05 46085

SQL> SELECT TO_CHAR(SYSTIMESTAMP+1/24, 'HH:MM:SS SSSSS AM'),
2          TO_CHAR(SYSTIMESTAMP+1/24, 'HH24:MM:SS SSSSS') FROM DUAL ;

TO_CHAR(SYSTIMESTAMP, 'TO_CHAR(SYSTIMESTAMP, '
-----
01:05:24 50004 PM 13:05:24 50004

SQL> SELECT TO_CHAR(SYSTIMESTAMP,
2          '"Nous sommes le : "FMdd month yyyy') FROM DUAL ;

TO_CHAR(SYSTIMESTAMP, '"NOUSSOMMESL
-----
Nous sommes le : 20 mai 2011

```

### Note

« **SYSTIMESTAMP** » est une pseudocolonne que l'on peut utiliser dans une expression de type « **TIMESTAMP** ».

## Extraire des informations

### EXTRACT

La fonction « **EXTRACT** » permet d'extraire un élément (jour, mois, année, heure, minute, seconde...) depuis un élément de type date ou bien un intervalle de temps.

**EXTRACT (FORMAT FROM EXPRESSION)**

#### FORMAT

Le format peut être une des valeurs suivantes :

- « **YEAR** », « **MONTH** », « **DAY** »,
- « **HOURL** », « **MINUTE** », « **SECOND** »,
- « **TIMEZONE\_HOUR** », « **TIMEZONE\_MINUTE** »,
- « **TIMEZONE\_REGION** », « **TIMEZONE\_ABBR** ».

```

SQL> SELECT NOM, EXTRACT (YEAR FROM DATE_EMBAUICHE) "Année",
2          EXTRACT (MONTH FROM DATE_EMBAUICHE) "Mois",
3          EXTRACT (DAY FROM DATE_EMBAUICHE) "Jour"
4 FROM EMPLOYEES ORDER BY 2,3,4;

```

NOM	Année	Mois	Jour
Malejac	1987	11	29
Alvarez	1988	8	3
Burst	1988	10	23
Peacock	1989	4	16
Maurer	1989	5	27
Tourtrel	1990	3	12
...			

```

SQL> SELECT EXTRACT (TIMEZONE_HOUR FROM CURRENT_TIMESTAMP) "Heure",
2          EXTRACT (TIMEZONE_MINUTE FROM CURRENT_TIMESTAMP) "Minutes",

```

```

3  EXTRACT(TIMEZONE_REGION FROM CURRENT_TIMESTAMP) "Région",
4  EXTRACT(TIMEZONE_ABBR FROM CURRENT_TIMESTAMP)"Abréviatio" FROM DUAL;

```

Heure	Minutes	Région	Abréviatio
01	00	UNKNOWN	UNK

```

SQL> ALTER SESSION SET TIME_ZONE = 'Europe/Paris';
SQL> SELECT EXTRACT(TIMEZONE_HOUR FROM CURRENT_TIMESTAMP) "Heure",
2  EXTRACT(TIMEZONE_MINUTE FROM CURRENT_TIMESTAMP) "Minutes",
3  EXTRACT(TIMEZONE_REGION FROM CURRENT_TIMESTAMP) "Région",
4  EXTRACT(TIMEZONE_ABBR FROM CURRENT_TIMESTAMP)"Abréviatio" FROM DUAL;

```

Heure	Minutes	Région	Abréviatio
01	00	Europe/Paris	CET

## Chaîne vers autres types

### TO\_NUMBER

La fonction « **TO\_NUMBER** » permet de convertir une chaîne de caractères, avec un certain format, en nombre.

**TO\_NUMBER(CHAINÉ,FORMAT)**

**FORMAT**

Le format (masque) que doit avoir la chaîne de caractères ; il est rarement utilisé. La définition du format est traitée plus loin dans la fonction « **TO\_CHAR** ».



```

SQL> SELECT TO_NUMBER('-1234567890','S9999999999') "Colonne 1",
2  TO_NUMBER('+1234567890','S9999999999') "Colonne 2",
3  TO_NUMBER('+123.456','S999.999') "Colonne 3",
4  TO_NUMBER('-123.456','999.999') "Colonne 4",
5  TO_NUMBER('+1E+123','S9.9EEEE') "Colonne 5",
6  TO_NUMBER('-1E+123','9.9EEEE') "Colonne 6" FROM DUAL ;

```

Colonne 1	Colonne 2	Colonne 3	Colonne 4	Colonne 5	Colonne 6
-1,235E+09	1234567890	123,456	-123,456	1,000E+123	-1,00E+123

### TO\_BINARY\_FLOAT

La fonction « **TO\_BINARY\_FLOAT** » permet de convertir un nombre en nombre réel à virgule flottante encodé sur 32 bits.



```

SQL> SELECT 123.69,TO_BINARY_FLOAT(123.69),TO_BINARY_FLOAT(123.69/0F)
2  FROM DUAL;

```

123.69	TO_BINARY_FLOAT(123.69)	TO_BINARY_FLOAT(123.69/0F)
123,69	1,237E+002	Inf

## TO\_BINARY\_DOUBLE

La fonction « **TO\_BINARY\_FLOAT** » permet de convertir un nombre en nombre réel à virgule flottante encodé sur 64 bits résultat.



```
SQL> SELECT SALAIRE, COMMISSION, TO_BINARY_DOUBLE(SALAIRE /
2 TO_BINARY_DOUBLE(COMMISSION)) "Résultat" FROM EMPLOYES;
```

SALAIRE	COMMISSION	Résultat
8000		
2856	250	1,142E+001
3500	1000	3,5E+000
3135	1500	2,09E+000
2180	0	Inf
2356	800	2,945E+000
...		

## TO\_DATE

La fonction « **TO\_DATE** » permet de convertir une chaîne de caractères, avec un certain format, en date.

**TO\_DATE(CHAÎNE,FORMAT)**

```
SELECT TO_DATE( '10/04/2006','DD/MM/YYYY'), TO_DATE( '12/10/2006 20:12:00',
2 'DD/MM/YYYY HH24:MI:SS') "TO_DATE" FROM DUAL ;
```

```
TO_DATE( TO_DATE
-----
10/04/06 12/10/06
```

## TO\_TIMESTAMP

La fonction « **TO\_TIMESTAMP** » permet de convertir une chaîne de caractères, avec un certain format, en valeur de type « **TIMESTAMP** ».

**TO\_TIMESTAMP(CHAÎNE,FORMAT)**

```
SQL> SELECT TO_TIMESTAMP ('10-septembre-2006 14:10:10,123000',
2 'DD-Month-YYYY HH24:MI:SSXFF') "TO_TIMESTAMP" FROM DUAL;
```

```
TO_TIMESTAMP
-----
10/09/06 14:10:10,123000000
```

### Note

Notez les fractions de seconde « **,123000** » et l'utilisation de « **XFF** » dans le masque de format. L'élément de format « **X** » indique l'emplacement du caractère décimal, dans ce cas une virgule « **,** », qui sépare les secondes entières des fractions de seconde. Dans l'exemple suivant on peut observer l'utilisation bien un point « **.FF** » pour obtenir le même résultat. La différence est que lorsque « **X** » est spécifié, Oracle détermine le bon caractère décimal à partir de la valeur du paramètre national.



```
SQL> SELECT TO_TIMESTAMP ('10-septembre-2006 14:10:10.123000',
2 'DD-Month-YYYY HH24:MI:SS.FF') "TO_TIMESTAMP" FROM DUAL;
```

```
TO_TIMESTAMP
-----
10/09/06 14:10:10,123000000
```



## TO\_TIMESTAMP\_TZ

La fonction « **TO\_TIMESTAMP\_TZ** » permet de convertir une chaîne de caractères, avec un certain format, en valeur de type « **TIMESTAMP WITH TIME ZONE** ».

**TO\_TIMESTAMP\_TZ(CHAÎNE,FORMAT)**

**CHAÎNE** L'argument CHAÎNE est une expression de type chaîne de caractères.

**FORMAT** Le format (masque) permet de lire la chaîne de caractère pour construire la date numérique.



```
SQL> SELECT TO_TIMESTAMP_TZ ('10092006 14:10:10.123000 -5:00',
2 'DDMMYYYY HH24:MI:SS.FF TZh:TzM') TO_TIMESTAMP_TZ",
3 TO_CHAR( SYSTIMESTAMP, 'DDMMYYYY HH24:MI:SS.FF TZh:TzM') "TO_CHAR"
4 FROM DUAL;
```

TO_TIMESTAMP_TZ	TO_CHAR
10/09/06 14:10:10,123000000 -05:00	14052006 09:37:23.218000 +02:00

```
SQL> SELECT TO_TIMESTAMP_TZ ('10092006 14:10:10.123000 AMERICA/NEW_YORK',
2 'DDMMYYYY HH24:MI:SS.FF TZR') "TO_TIMESTAMP_TZ" FROM DUAL;
```

TO_TIMESTAMP_TZ
10/09/06 14:10:10,123000000 AMERICA/NEW_YORK

### Note



Notez que le fuseau horaire est représenté utilisant un décalage en heures et minutes par rapport à la zone UTC. L'utilisation des éléments de format « **TZh** » et « **TzM** » permet d'indiquer l'emplacement des heures et des minutes dans la chaîne en entrée.

Le deuxième exemple montre un fuseau horaire spécifié en utilisant le nom de la région, la zone « **AMERICA/NEW\_YORK** » ; il faut utiliser l'élément « **TZR** » dans le masque de format afin d'indiquer à quel endroit le nom de la région apparaît dans la chaîne en entrée.

## TO\_YMINTERVAL

La fonction « **TO\_YMINTERVAL** » permet de convertir une chaîne de caractères, en valeur de type « **INTERVAL YEAR TO MONTH** ».



```
SQL> SELECT SYSDATE, SYSDATE + TO_YMINTERVAL('01-02') "14 mois" FROM DUAL;
```

SYSDATE	14 mois
14/05/06	14/07/07

## TO\_DSINTERVAL

La fonction « **TO\_DSINTERVAL** » permet de convertir une chaîne de caractères, en valeur de type « **INTERVAL DAY TO SECOND** ».



```
SQL> SELECT SYSDATE, TO_DSINTERVAL ('10 10:00:0000.1'),
2 SYSDATE + TO_DSINTERVAL ('10 10:00:0000.1') FROM DUAL;
```

SYSDATE	TO_DSINTERVAL('1010:00:0000.1')	SYSDATE+
14/05/06	+0000000010 10:00:00.100000000	24/05/06

# Numérique vers interval

## NUMTOYMINTERVAL

La fonction « **NUMTOYMINTERVAL** » permet de convertir une valeur numérique, avec un certain format, en valeur de type « **INTERVAL YEAR TO MONTH** ».

**NUMTOYMINTERVAL (NUMERIQUE, FORMAT)**

**NUMERIQUE** L'argument **NUMERIQUE** est une expression de type numérique.

**FORMAT** Le format (masque) permet de lire la chaîne de caractère pour construire la date numérique.

<i>Format</i>	<i>Description</i>
YEAR	Nombre d'années, de 1 à 999.999.999.
MONTH	Nombre de mois, de 1 à 11.
DAY	Nombre de jours, de 0 à 999.999.999.
HOURL	Nombre d'heures, de 0 à 23.
MINUTE	Nombre de minutes, de 0 à 59.
SECOND	Nombre de secondes, de 0 à 59.999999999.



```
SQL> SELECT NUMTOYMINTERVAL( 8.5, 'YEAR' ) FROM DUAL;
```

```
NUMTOYMINTERVAL( 8.5, 'YEAR' )
```

```
-----
+0000000008-06
```

```
SQL> SELECT NUMTOYMINTERVAL( 8, 'MONTH' ) FROM DUAL;
```

```
NUMTOYMINTERVAL( 8, 'MONTH' )
```

```
-----
+0000000000-08
```

```
SQL> SELECT NUMTOYMINTERVAL( 8.5 ) FROM DUAL;
```

```
SELECT NUMTOYMINTERVAL( 8.5 )
```

```
*
```

```
ERREUR à la ligne 1 :
```

```
ORA-00909: nombre d'arguments non valide
```

### Attention

Le format, comme vous pouvez voir dans l'exemple précédent, n'est pas optionnel ; il faut, pour chaque valeur numérique que vous voulez convertir, expliquer sa signification par le format.

Cette règle est également valable pour la fonction SQL « **NUMTODSINTERVAL** ».



## NUMTODSINTERVAL

La fonction « **NUMTODSINTERVAL** » permet de convertir une valeur numérique, avec un certain format, en valeur de type « **INTERVAL DAY TO SECOND** ».

**NUMTODSINTERVAL (NUMERIQUE, FORMAT)**

## NUMERIQUE

L'argument NUMERIQUE est une expression de type numérique.

## FORMAT

Le format (masque) permet de lire la chaîne de caractère pour construire la date numérique.



```
SQL> SELECT NUMTODSINTERVAL( 2056, 'MINUTE'),
2          NUMTODSINTERVAL( 200.56001, 'SECOND') FROM DUAL;

NUMTODSINTERVAL(2056, 'MINUTE')  NUMTODSINTERVAL(200.56001, 'SECOND')
-----
+0000000001 10:16:00.000000000    +0000000000 00:03:20.560010000

SQL> SELECT NUMTODSINTERVAL( 2056, 'DAY') + NUMTODSINTERVAL( 20, 'MINUTE') +
2 NUMTODSINTERVAL( 20.568, 'SECOND') "Somme" FROM DUAL;

Somme
-----
+000002056 00:20:20.568000000
```

# Les fonctions générales

Le langage SQL propose également des fonctions générales qui travaillent avec tous les types de données.

## GREATEST

La fonction « **GREATEST** » permet de trouver la plus grande valeur dans une liste de valeurs.

**GREATEST**(EXPRESSION1, EXPRESSION2[, EXPRESSION3...])

### EXPRESSION

Les arguments EXPRESSION peuvent être de type numérique, chaîne ou date. Le type de donnée du premier argument détermine le type de retour de la fonction. Les arguments suivants sont convertis automatiquement au type du premier.



```
SQL> SELECT GREATEST ('HARRY', 'Harry', 'HARRIOT', 'HAROLD')
2          "Greatest", ASCII('A'), ASCII('a') FROM DUAL;

Great ASCII('A') ASCII('a')
-----
Harry          65          97

SQL> SELECT GREATEST ( TO_DATE('01/10/2011'), TO_DATE('22/05/2011'),
2 TO_DATE('21/08/2011')) "Date", GREATEST ('01/10/2002',
3 TO_DATE('22/05/2011'), TO_DATE('21/08/2011')) "Chaîne" FROM DUAL ;

Date          Chaîne
-----
01/10/2011 00:00:00 22/05/2011 00:00:00
```

## LEAST

La fonction « **LEAST** » permet de trouver la plus petite valeur dans une liste de valeurs.

**LEAST**(EXPRESSION1, EXPRESSION2[, EXPRESSION3...])



**EXPRESSION**

Les arguments **EXPRESSION** peuvent être de type numérique, chaîne ou date. Le type de donnée du premier argument détermine le type de retour de la fonction. Les arguments suivants sont convertis automatiquement au type du premier.



```
SQL> SELECT LEAST ( TO_NUMBER('033'),'22','21') "Numérique",
2          LEAST ('033','22','21') "Chaîne" FROM DUAL ;
```

```
Numérique Cha
-----
21 033
```

**DECODE**

La fonction « **DECODE** » permet de choisir une valeur parmi une liste d'expressions, en fonction de la valeur prise par une expression servant de critère de sélection.

**DECODE(EXPRESSION,VALEUR1,RESULTAT1[,VALEUR2,RESULTAT2...][,DEFAULT])**

**EXPRESSION**

L'argument **EXPRESSION** peut être de type numérique, chaîne ou date et retourne la valeur qui doit être évaluée.

**VALEUR1...N**

L'argument **VALEUR1** est de même type que **EXPRESSION**. Si **EXPRESSION** retourne une valeur égale à **VALEUR1** alors « **DECODE** » retourne **RESULTAT1**.

**DEFAULT**

L'argument **DEFAULT** est la valeur de retour pour « **DECODE** » si **EXPRESSION** n'a pas une valeur dans la liste **VALEUR1, ... VALEURN**.



```
SQL> SELECT NOM||' '||PRENOM "Employé",
2      ROUND(( DATE_EмбаУCHE - DATE_NAISSANCE)/365,-1) "Ancienneté",
3      DECODE( ROUND(( DATE_EмбаУCHE - DATE_NAISSANCE)/365,-1),
4            10,'Nouveau', 20,'Ancien','Senior') FROM EMPLOYES ;
```

```
Employé          Ancienneté DECODE(
-----
Berlioz Jacques      30 Senior
Nocella Guy          20 Ancien
Herve Didier         20 Ancien
Mangeard Jocelyne    20 Ancien
Cazade Anne-Claire   20 Ancien
Devie Thérèse        30 Senior
Peacock Margaret     20 Ancien
Idesheim Annick      20 Ancien
Rollet Philippe      20 Ancien
Silberreiss Albert   20 Ancien
Weiss Sylvie         20 Ancien
Delorgue Jean        20 Ancien
Zonca Virginie       20 Ancien
Twardowski Colette   20 Ancien
Coutou Myriam        20 Ancien
King Robert          30 Senior
Ragon André          10 Nouveau
...
```

```
SQL> SELECT NO_FOURNISSEUR "N°", DECODE(MOD(ROWNUM,5),0,ROWNUM) "Ligne",
2      SOCIETE FROM FOURNISSEURS;
```

```
N° Ligne SOCIETE
```

```

-----
1      Exotic Liquids
2      Nouvelle-Orléans Cajun Delights
3      Grandma Kelly's Homestead
4      Tokyo Traders
5      5 Cooperativa de Quesos 'Las Cabras'
6      Mayumi's
7      Pavlova, Ltd.
8      Specialty Biscuits, Ltd.
9      PB Knäckebröd AB
10     10 Refrescos Americanas LTDA
11     Heli Süßwaren GmbH Co. KG
...

SQL> SELECT DECODE(PAYS,'France',NO_FOURNISSEUR) "France",
2         DECODE(PAYS,'Allemagne',NO_FOURNISSEUR) "Allemagne",
3         DECODE(PAYS,'Royaume-Uni',NO_FOURNISSEUR) "Royaume-Uni",
4         DECODE(PAYS,'France','','Allemagne','','Royaume-Uni','','
5         NO_FOURNISSEUR) "Autres" FROM FOURNISSEURS ;

      France  Allemagne  Royaume-Uni  Autres
-----
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
1
2
3
4
5
6
7
8
9
10

```

## CASE

L'instruction « **CASE** » permet de mettre en place une condition d'instruction conditionnelle « **IF..THEN..ELSE** » directement dans une requête. Le fonctionnement est similaire à la fonction « **DECODE** » avec plus de flexibilité. La première syntaxe de cette fonction est :

### CASE EXPRESSION

```
WHEN VALEUR1 THEN RESULTAT1
[WHEN VALEUR2 THEN RESULTAT2,...]
[ELSE RESULTAT]
```

END ;

#### EXPRESSION

L'argument EXPRESSION peut être de type numérique, chaîne, ou date, et retourne la valeur qui doit être évaluée.

#### VALEUR1...N

L'argument VALEUR1 est de même type que EXPRESSION.



```
SQL> SELECT NOM||' '||PRENOM "Employé", FONCTION,
2          CASE FONCTION
3            WHEN 'Président' THEN
4              SALAIRE*1.05
5            WHEN 'Vice-Président' THEN
6              SALAIRE*1.1
7            WHEN 'Chef des ventes' THEN
8              SALAIRE*1.2
9            WHEN 'Représentant(e)' THEN
10             SALAIRE*1.1 + COMMISSION
11           ELSE
12             SALAIRE
13           END "Salaire"
14 FROM EMPLOYES;
```

Employé	FONCTION	Salaire
...		
Brasseur Hervé	Vice-Président	161700
Pagani Hector	Représentant(e)	8480
Gerard Sylvie	Représentant(e)	10820
Poupard Claudette	Assistante commerciale	1800
Piroddi Nathalie	Représentant(e)	7560
Splingart Lydia	Chef des ventes	19200
Chambaud Axelle	Chef des ventes	14400
Giroux Jean-Claude	Président	157500
...		

La deuxième syntaxe de cette fonction est :

### CASE

```
WHEN CONDITION1 THEN RESULTAT1
[WHEN CONDITION2 THEN RESULTAT2,...]
[ELSE RESULTAT]
```

END ;

#### CONDITION

L'argument CONDITION est une expression logique.



```
SQL> SELECT NOM, FONCTION, SALAIRE,
2          CASE
3            WHEN FONCTION = 'Assistante commerciale'
4              THEN '10%'
5            WHEN FONCTION = 'Représentant(e)' AND
6              SALAIRE < 2600
```

```

7      THEN '30%'
8      WHEN FONCTION = 'Représentant(e)' AND
9          SALAIRE < 3200
10     THEN '20%'
11     ELSE
12         'Pas d'augmentation'
13     END "Salaire" FROM EMPLOYES ;

```

NOM	FONCTION	SALAIRE	Salaire
Berlioz	Représentant(e)	9400	Pas d'augmentation
Nocella	Représentant(e)	7600	20%
Herve	Représentant(e)	6700	20%
Mangeard	Représentant(e)	8200	Pas d'augmentation
Cazade	Représentant(e)	7200	20%
Devie	Assistante commerciale	1540	10%
Peacock	Représentant(e)	6100	20%
Idesheim	Représentant(e)	7300	20%
Rollet	Représentant(e)	5000	30%
Silberreiss	Représentant(e)	9200	Pas d'augmentation
Weiss	Représentant(e)	6600	20%
Delorgue	Représentant(e)	5900	30%
Zonca	Représentant(e)	6700	20%
Twardowski	Représentant(e)	9400	Pas d'augmentation

## NULLIF

L'instruction « **NULLIF** » permet de comparer **EXPRESSION1** et **EXPRESSION2** ; si les deux expressions sont égales alors la valeur « **NULL** » est retournée, sinon **EXPRESSION1**.

La syntaxe de « **NULLIF** » est : **NULLIF ( EXPRESSION1, EXPRESSION2 ) ;**



```

SQL> SELECT COMMISSION, NULLIF(COMMISSION, 1000) RETOUR1,
2  NOM||'|'|PRENOM "Employé", NULLIF(LENGTH(NOM),LENGTH(PRENOM)) RETOUR2
3  FROM EMPLOYES;

```

COMMISSION	RETOUR1	Employé	RETOUR2
980	980	Berlioz Jacques	
910	910	Nocella Guy	7
1170	1170	Herve Didier	5
190	190	Mangeard Jocelyne	
550	550	Cazade Anne-Claire	6
		Devie Thérèse	5
930	930	Peacock Margaret	7
600	600	Idesheim Annick	8
570	570	Rollet Philippe	6
1100	1100	Silberreiss Albert	11
1110	1110	Weiss Sylvie	5
510	510	Delorgue Jean	8
470	470	Zonca Virginie	5
1540	1540	Twardowski Colette	10
...			

## COALESCE

L'instruction « **COALESCE** » permet de retourner la première expression « **NOT NULL** » de la liste des paramètres.



**COALESCE ( EXPRESSION1, EXPRESSION2 [,...] ) ;**

```
SQL> SELECT NOM_PRODUIT,
2          CASE
3              WHEN UNITES_COMMANDEES IS NOT NULL
4              THEN 'Unités commandées = '
5              WHEN UNITES_STOCK IS NOT NULL
6              THEN 'Unités en stock = '
7              ELSE
8                  'Produit indisponible = '
9          END ||
10         COALESCE(UNITES_COMMANDEES, UNITES_STOCK, INDISPONIBLE)
11         "Stock des produits"
12 FROM PRODUITS;
```

NOM_PRODUIT	Stock des produits
Chai	Unités en stock = 39
Chang	Unités commandées = 40
Aniseed Syrup	Unités commandées = 70
Chef Anton's Cajun Seasoning	Unités en stock = 53
Grandma's Boysenberry Spread	Unités en stock = 120
Uncle Bob's Organic Dried Pears	Unités en stock = 15
Northwoods Cranberry Sauce	Unités en stock = 6
Mishi Kobe Niku	Produit indisponible = 0
Ikura	Unités en stock = 31
Queso Cabrales	Unités commandées = 30
Queso Manchego La Pastora	Unités en stock = 86
Konbu	Unités en stock = 24
Alice Mutton	Produit indisponible = 0
Teatime Chocolate Biscuits	Unités en stock = 25
Sir Rodney's Marmalade	Unités en stock = 40
Sir Rodney's Scones	Unités commandées = 40
Gustaf's Knäckebröd	Unités en stock = 104
Tunnbröd	Unités en stock = 61
NuNuCa Nuß-Nougat-Creme	Unités en stock = 76
Rössle Sauerkraut	Produit indisponible = 0
Mascarpone Fabioli	Unités commandées = 40
Sasquatch Ale	Unités en stock = 111
Steeleye Stout	Unités en stock = 20
Inlagd Sill	Unités en stock = 112
Gravad lax	Unités commandées = 50
Chartreuse verte	Unités en stock = 69
Boston Crab Meat	Unités en stock = 123
Singaporean Hokkien Fried Mee	Produit indisponible = 0
Ipoh Coffee	Unités commandées = 10
...	

- Fonctions « verticales »
- Groupe
- Sélection du groupe
- CUBE
- ROLLUP

# 6

## Le groupement des données



### Objectifs

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- Utiliser les fonctions "verticales".
- Effectuer des regroupements dans le cadre des requêtes.
- Sélectionner les lignes du groupe.
- Effectuer des regroupements à deux niveaux.
- Utiliser les fonctions d'agrégation multidimensionnelles.
- Effectuer des regroupements et calculer les résultats intermédiaires.



### Contenu

Fonctions	6-2	Le groupe à deux niveaux	6-9
Fonctions d'agrégat	6-2	Groupe ROLLUP	6-9
Fonctions d'agrégat statistiques	6-4	Groupe CUBE	6-13
Le groupe	6-5	GROUPING SETS	6-14
La sélection de groupe	6-7	Fonctions de groupes	6-17

# Fonctions

---

## Fonctions « Horizontales »

Le module précédent explique l'utilisation des fonctions pour enrichir les requêtes de base et permettre de manipuler les données stockées dans la base.

Les fonctions étudiées sont des fonctions "horizontales" qui manipulent des données d'une seule ligne (enregistrement).

Les fonctions "horizontales" fournissent un résultat et utilisent comme arguments les valeurs des colonnes, pour chaque ligne de la requête. Il est impossible, avec les fonctions "horizontales", de calculer des expressions entre plusieurs lignes.

Dans la pratique on a besoin d'effectuer des calculs qui portent sur les valeurs d'une ligne, mais aussi des calculs de synthèse, par exemple connaître le stock des produits, le cumul des salaires ou les produits vendus par client. SQL fournit une série de fonctions "verticales" pour les regroupements et le calcul cumulatif.

## Fonctions « Verticales »

Les fonctions "verticales" sont des fonctions qui opèrent sur des groupes de lignes. Les fonctions "verticales" ou les fonctions d'agrégat, sont utilisées pour des calculs cumulatifs de valeurs définies par requête. Ce sont essentiellement des fonctions de calcul qui assemblent des données de même type.

Le langage SQL offre un mécanisme permettant de travailler sur des valeurs obtenues par regroupement des lignes résultats de l'exécution d'une requête. Soit la requête :

```
SQL> SELECT NOM_PRODUIT, NO_FOURNISSEUR, CODE_CATEGORIE,  
2          PRIX_UNITAIRE, UNITES_STOCK, UNITES_COMMANDEES  
3 FROM PRODUITS;
```

Elle permet d'afficher pour chaque produit de chaque catégorie, son nom, numéro fournisseur, prix unitaire, unités en stock et unités commandées, qui sont les données brutes de la base de données.

Il est aussi possible de connaître en une seule requête des informations complexes construites à partir des données enregistrées, telles que la quantité totale des unités en stock par catégorie, la quantité totale des unités commandées ou la catégorie qui cumule le plus d'unités commandées.

L'étude des fonctions "verticales" commence avec le calcul de synthèse sur l'ensemble des lignes retournées par la requête.

# Fonctions d'agrégat

---

Les fonctions "verticales" ou les fonctions d'agrégat, sont utilisées pour le calcul cumulatif des valeurs par rapport à un regroupement ou pour l'ensemble des lignes de la requête. La notion du groupe fait l'objet d'une présentation ultérieure, pour l'instant les fonctions d'agrégat sont utilisées pour l'ensemble des lignes de la requête.

Les fonctions "verticales" traitent les valeurs « **NULL** » différemment des fonctions "horizontales" dans ce sens qu'elles n'en tiennent pas compte et calculent le résultat malgré leur présence.

## SUM

La fonction « **SUM** » calcule la somme des expressions arguments pour l'ensemble des lignes correspondantes.



```
SQL> SELECT SUM(SALAIRE),SUM(COMMISSION) FROM EMPLOYES ;
```

SUM(SALAIRE)	SUM(COMMISSION)
1185440	157690

## AVG

La fonction « **AVG** » calcule la moyenne des expressions arguments pour l'ensemble des lignes correspondantes. La fonction « **AVG** » est influencée par les valeurs « **NULL** », la somme est calculée pour l'ensemble des lignes mais le nombre des lignes pris en compte est seulement celui pour la quelle la valeur EXPRESSION est « **NOT NULL** ».



```
SQL> SELECT AVG(COMMISSION),AVG(NVL(COMMISSION,0)) FROM EMPLOYES ;
```

AVG(COMMISSION)	AVG(NVL(COMMISSION,0))
1609,08163	1420,63063

## MIN

La fonction « **MIN** » calcule la plus petite des valeurs pour les expressions arguments pour l'ensemble des lignes correspondantes.



```
SQL> SELECT MIN(UNITES_STOCK) MIN_STOCK, MIN(UNITES_COMMANDEES) MIN_COMM,
2 MIN(NOM_PRODUIT) MIN_NOM FROM PRODUITS;
```

MIN_STOCK	MIN_COMM	MIN_NOM
0	5	Alice Mutton

## MAX

La fonction « **MAX** » calcule la plus grande des valeurs pour les expressions arguments pour l'ensemble des lignes correspondantes.



```
SQL> SELECT MAX(SALAIRE), MAX(COMMISSION), MAX(DATE_NAISSANCE) MAX_DATE,
3 MAX(NOM) FROM EMPLOYES;
```

MAX(SALAIRE)	MAX(COMMISSION)	MAX_DATE	MAX(NOM)
150000	16480	31/07/1991 00:00:00	Zonca

## COUNT

La fonction « **COUNT** » calcule le nombre des valeurs non NULL des expressions arguments pour l'ensemble des lignes correspondantes.

**COUNT([ALL|DISTINCT] EXPRESSION) = RETOUR**



```
SQL> SELECT COUNT(*), COUNT(FONCTION), COUNT(DISTINCT FONCTION),
3 COUNT(COMMISSION) FROM EMPLOYES;
```

COUNT(*)	COUNT(FONCTION)	COUNT(DISTINCTFONCTION)	COUNT(COMMISSION)
111	111	5	98

Dans l'exemple, vous pouvez distinguer quatre utilisations de la fonction COUNT pour le calcul du nombre :

- des lignes distinctes de la table EMPLOYES,



- des valeurs non « **NULL** » de la colonne **FONCTION**, sans tenir compte des doublons,
- des valeurs non « **NULL** » et distinctes de la colonne **FONCTION**.
- des valeurs non « **NULL** » de la colonne **COMMISSION**.



### Conseil

L'argument « **DISTINCT** » est utilisé pour calculer les valeurs de l'expression distinctes et non « **NULL** ». Il peut être utilisé dans toutes les fonctions "verticales" pour éliminer les doublons ; cependant il faut faire attention car l'élimination des doublons impacte sur le résultat.

## Fonctions d'agrégat statistiques

### STDDEV et STDDEV\_SAMP

La fonction calcule l'écart type des valeurs pour l'ensemble des lignes correspondantes dans l'hypothèse où les arguments ne représentent qu'un échantillon de la population. L'écart type est une mesure de la dispersion des valeurs par rapport à la valeur moyenne. S'il n'y a qu'une seule ligne comme argument, la fonction « **STDDEV** » retourne la valeur « **0** » et la fonction « **STDDEV\_SAMP** » retourne la valeur « **NULL** ».

### STDDEV\_POP

La fonction calcule l'écart type des valeurs pour l'ensemble des lignes correspondantes dans l'hypothèse où les arguments représentent l'ensemble de la population.

### VARIANCE et VAR\_SAMP

La fonction calcule la variance pour l'ensemble des lignes correspondantes dans l'hypothèse où les arguments ne représentent qu'un échantillon de la population. S'il n'y a qu'une seule ligne comme argument la fonction « **VARIANCE** » retourne la valeur « **0** » et la fonction « **VARIANCE\_SAMP** » retourne la valeur « **NULL** ».

Le calcul est effectué à l'aide de la formule :

$$(\text{SUM}(\text{expr}^2) - \text{SUM}(\text{expr})^2 / \text{COUNT}(\text{expr})) / (\text{COUNT}(\text{expr}) - 1)$$

### VAR\_POP

La fonction calcule la variance pour l'ensemble des lignes correspondantes dans l'hypothèse où les arguments représentent l'ensemble de la population. Le calcul est effectué à l'aide de la formule :

$$(\text{SUM}(\text{expr}^2) - \text{SUM}(\text{expr})^2 / \text{COUNT}(\text{expr})) / \text{COUNT}(\text{expr})$$



```
SQL> SELECT AVG(SALAIRE) MOYENNE, STDDEV_POP(SALAIRE) STDDEV_POP,
2 STDDEV_SAMP(SALAIRE) STDDEV_SAMP, VAR_POP(SALAIRE) VAR_POP,
3 VAR_SAMP(SALAIRE) VAR_SAMP, VARIANCE(SALAIRE) VARIANCE FROM EMPLOYES;
```

MOYENNE	STDDEV_POP	STDDEV_SAMP	VAR_POP	VAR_SAMP	VARIANCE
10679,6396	20639,5775	20733,1815	425992158	429864814	429864814

### COVAR\_SAMP

La fonction calcule la covariance, moyenne des produits des écarts pour chaque série d'observations, pour l'ensemble des lignes correspondantes dans l'hypothèse où les arguments ne représentent qu'un échantillon de la population. Utilisez la covariance pour déterminer la relation entre deux ensembles de données.

**COVAR\_SAMP ( EXPRESSION1, EXPRESSION2) = RETOUR**

Le calcul est effectué à l'aide de la formule :

$$(\text{SUM}(\text{expr1} * \text{expr2}) - \text{SUM}(\text{expr2}) * \text{SUM}(\text{expr1}) / n) / (n-1)$$

## COVAR\_POP

La fonction calcule la covariance, moyenne des produits des écarts pour chaque série d'observations, pour l'ensemble des lignes correspondantes dans l'hypothèse où les arguments représentent l'ensemble de la population.

**COVAR\_POP ( EXPRESSION1, EXPRESSION2) = RETOUR**

Le calcul est effectué à l'aide de la formule :

$$(\text{SUM}(\text{expr1} * \text{expr2}) - \text{SUM}(\text{expr2}) * \text{SUM}(\text{expr1}) / n) / n$$


```
SQL> SELECT COVAR_SAMP(SYSDATE-DATE_EмбаУCHE, SALAIRE) COVAR_SAMP,
2 COVAR_POP(SYSDATE-DATE_EмбаУCHE, SALAIRE) COVAR_POP,
3 (VAR_POP((SYSDATE-DATE_EмбаУCHE) + SALAIRE) -
4 (VAR_POP(SYSDATE-DATE_EмбаУCHE) + VAR_POP(SALAIRE))) / 2
5 COVAR_POP_CALCUL FROM EMPLOYES;
```

COVAR_SAMP	COVAR_POP	COVAR_POP_CALCUL
-3181762,1	-3153097,6	-3153097,6

## Le groupe

Les fonctions "verticales" ou les fonctions d'agrégat peuvent être utilisées pour le calcul cumulatif des valeurs par rapport à un regroupement ou pour l'ensemble des lignes de la requête.

Le groupe offre un mécanisme permettant de travailler sur un ou plusieurs regroupements de lignes dans l'ensemble des enregistrements de la requête. Un regroupement est formé d'un ensemble d'enregistrements ayant une ou plusieurs caractéristiques communes.

La définition du groupe se fait par l'intermédiaire de la clause « **GROUP BY** ».

Dans l'exemple on calcule la somme des salaires pour chaque élément du groupe FONCTION, le résultat étant le cumul des salaires pour chaque fonction.

La syntaxe de l'instruction « **SELECT** » :

**SELECT [ALL | DISTINCT]{\*,[EXPRESSION1 [AS] ALIAS1[,...]]}**

**FROM NOM\_TABLE WHERE PREDICAT**

**GROUP BY [NOM\_COLONNE1|EXPRESSION1][,...]**

**ORDER BY [NOM\_COLONNE1|POSITION1] [ASC|DESC][,...] ;**

```
SQL> SELECT FONCTION, SUM(SALAIRE) FROM EMPLOYES GROUP BY FONCTION ;
```

FONCTION	SUM(SALAIRE)
Assistante commerciale	16540
Chef des ventes	83000
Président	150000
Représentant(e)	692900
Vice-Président	243000



## Note

Le regroupement se fait d'abord selon le premier critère spécifié dans la clause « **GROUP BY** », puis les lignes ayant le même groupe sont regroupées selon le deuxième critère de la clause « **GROUP BY** », etc. L'ensemble des critères définit le groupe ; les fonctions "verticales" sont exécutées chaque fois que la valeur du groupe change.

```
SQL> SELECT NO_FOURNISSEUR, CODE_CATEGORIE, SUM(UNITES_STOCK) SUM_STOCK,
2         COUNT(CODE_CATEGORIE) NB_CATEG FROM PRODUITS
3 GROUP BY NO_FOURNISSEUR, CODE_CATEGORIE
4 ORDER BY NO_FOURNISSEUR, CODE_CATEGORIE;
```

NO_FOURNISSEUR	CODE_CATEGORIE	SUM_STOCK	NB_CATEG
1	1	56	2
1	2	13	1
2	2	133	4
2	3	20	1
2	7	20	1
2	9	40	1
3	2	126	2
3	7	15	1
4	2	60	1
4	3	20	1
4	6		1
4	7	104	2
...			

Dans l'exemple précédent, vous pouvez remarquer que le groupe est formé par les deux critères précisés dans la clause « **GROUP BY** », le numéro de fournisseur (NO\_FOURNISSEUR) et le code catégorie (CODE\_CATEGORIE). Le groupe détermine le niveau de détail, l'ensemble des lignes pour lesquelles on exécute le calcul de la somme.

Pour les requêtes qui n'utilisent pas des fonctions "verticales" et groupes, le niveau de détail est défini par les enregistrements des tables. Pour les requêtes qui utilisent les groupes et fonctions "verticales", le niveau de détail est déterminé par le groupe.

```
SQL> SELECT SUM(UNITES_STOCK*PRIX_UNITAIRE),
2         SUM(UNITES_COMMANDEES*PRIX_UNITAIRE)
3 FROM PRODUITS WHERE NO_FOURNISSEUR IN (1,2,5);
```

SUM(UNITES_STOCK*PRIX_UNITAIRE)	SUM(UNITES_COMMANDEES*PRIX_UNITAIRE)
38575	18950

Dans l'exemple précédent, vous pouvez voir que la requête ne retourne qu'une seule ligne qui rassemble l'ensemble des lignes de la table qui respecteront les conditions de la clause « **WHERE** ».

## Attention

Toute requête qui utilise des fonctions "verticales" sur un groupe défini, doit afficher, dans les expressions qui ne sont pas des arguments des fonctions "verticales" seulement les colonnes contenues dans la clause « **GROUP BY** ». Les colonnes affichables, en dehors des fonctions "verticales", sont celles qui ont une valeur unique dans le groupe.

```
SQL> SELECT NOM,FONCTION, SUM(SALAIRE+NVL(COMMISSION,0)) FROM EMPLOYES;
SELECT NOM,FONCTION,
*
ERREUR à la ligne 1 :
ORA-00937: la fonction de groupe ne porte pas sur un groupe simple
```



```
SQL> SELECT CODE_CLIENT, DATE_ENVOI, SUM(PORT) FROM COMMANDES
2      GROUP BY CODE_CLIENT, TO_CHAR(DATE_ENVOI, 'YYYY');
      DATE_ENVOI,
      *
ERREUR à la ligne 2 :
ORA-00979: N'est pas une expression GROUP BY
```

Dans l'exemple précédent, vous pouvez constater que DATE\_ENVOI, qui se trouve dans la clause « **GROUP BY** » dans la composition d'une expression, ne peut pas être affiché parce que sa valeur n'est pas unique dans le groupe.

Une colonne composant d'une expression critère d'un groupe doit, pour pouvoir être utilisée dans les expressions destinées à l'affichage, être employée avec la même expression de la clause « **GROUP BY** ».



```
SQL> SELECT NO_EMPLOYE,
2      'Année ' || TO_CHAR(DATE_COMMANDE, ' YYYY ' ) "Année",
3      TO_CHAR(SUM(PORT), '99G999D99') "Port"
4  FROM COMMANDES GROUP BY NO_EMPLOYE, TO_CHAR(DATE_COMMANDE, ' YYYY ' );
```

NO_EMPLOYE	Année	Port
31	Année 2010	3 749,80
101	Année 2010	11 629,10
108	Année 2010	1 296,00
53	Année 2010	5 068,00
62	Année 2010	9 687,40
17	Année 2010	3 609,30
59	Année 2010	9 828,10
81	Année 2010	3 162,00
19	Année 2010	1 236,90
65	Année 2011	2 855,40
111	Année 2011	12 632,50
9	Année 2011	3 029,30
...		

### Note

Il faut noter que les enregistrements retournés par les requêtes ne sont pas triés dans l'ordre des colonnes décrites dans la clause « **GROUP BY** ». Pour effectuer le tri des enregistrements, il faut préciser explicitement les critères de tri dans la clause « **ORDER BY** ».

## La sélection de groupe

Les sélections dans une requête sans groupe sont effectuées dans la clause « **WHERE** ». Dans cette clause le prédicat (l'ensemble des critères de sélection) est exécuté pour chaque enregistrement de la table, le niveau de détail, le résultat de la requête étant formé par les lignes qui vérifient le prédicat.

Les requêtes groupées peuvent être sélectionnées à l'aide de la clause « **HAVING** », pour spécifier le prédicat sur groupe.

La syntaxe de l'instruction « **SELECT** » :

```
SELECT [ALL | DISTINCT]{*, [EXPRESSION1 [AS] ALIAS1[, ...]]}
FROM NOM_TABLE WHERE PREDICAT
GROUP BY [NOM_COLONNE1 | EXPRESSION1][, ...] HAVING PREDICAT
```



```
ORDER BY [NOM_COLONNE1|EXPRESSION1] [ASC|DESC][, ...] ;
SQL> SELECT NO_EMPLOYE, TO_CHAR(DATE_COMMANDE, ' YYYY ') "Année",
2 TO_CHAR(SUM(PORT), '99G999D99') "Port" FROM COMMANDES
3 WHERE DATE_COMMANDE > '01/01/2011' AND NO_EMPLOYE <= 30
4 GROUP BY NO_EMPLOYE, TO_CHAR(DATE_COMMANDE, ' YYYY ')
5 HAVING SUM(PORT) > 5000 ORDER BY SUM(PORT) DESC;
```

NO_EMPLOYE	Année	Port
29	2011	13 068,30
7	2011	11 680,30
1	2011	6 976,90
4	2011	5 805,30
22	2011	5 464,40
3	2011	5 051,30

Dans l'exemple précédent, vous pouvez remarquer que le groupe est formé par les deux critères précisés dans la clause « **GROUP BY** », le numéro d'employé (NO\_EMPLOYE) et l'année de la commande. Oracle exécute les clauses dans un ordre bien défini :

1. Sélectionne les lignes conformément à la clause « **WHERE** ».
2. Groupe les lignes conformément à la clause « **GROUP BY** ».
3. Calcule les résultats des fonctions d'agrégat pour chaque groupe.
4. Élimine les groupes conformément à la clause « **HAVING** ».
5. Ordonne les groupes conformément à la clause « **ORDER BY** ».

L'ordre d'exécution est important, car il affecte directement les performances des requêtes. En général, plus le nombre d'enregistrements éliminés par une clause « **WHERE** » est grand, plus l'exécution de la requête est rapide. Ce gain en performances provient de la réduction du nombre de lignes devant être traitées durant l'opération « **GROUP BY** ».

Lorsqu'une requête inclut une clause « **HAVING** », il est préférable de la remplacer par une clause « **WHERE** ». Toutefois, cette substitution est généralement possible seulement lorsque la clause « **HAVING** » est utilisée pour éliminer des groupes basés sur la colonne de groupement. Prenez l'exemple précédent : NO\_EMPLOYE peut être utilisé aussi bien dans la clause « **WHERE** » que dans la clause « **HAVING** » cependant la requête s'exécute plus vite s'il est utilisé dans la clause « **WHERE** » étant donné que le nombre des lignes à regrouper est moins important.

### Attention

Les expressions utilisées dans la clause « **HAVING** » peuvent contenir seulement des colonnes et expressions contenues dans la clause « **GROUP BY** » ou des fonctions "verticales" qui respectent la même syntaxe que les expressions de l'affichage.

```
SQL> SELECT NO_EMPLOYE, TO_CHAR(DATE_COMMANDE, ' YYYY ') "Année",
2 TO_CHAR(SUM(PORT), '99G999D99U') FROM COMMANDES
3 GROUP BY NO_EMPLOYE, TO_CHAR(DATE_COMMANDE, ' YYYY ')
4 HAVING SUM(PORT) > 18000 AND DATE_COMMANDE > '01/01/1997'
9 ORDER BY SUM(PORT) DESC ;
DATE_COMMANDE > '01/01/1997'
*
```

ERREUR à la ligne 8 :

ORA-00979: N'est pas une expression GROUP BY

Une requête peut contenir à la fois une clause « **WHERE** » et une clause « **HAVING** ». Dans ce cas, la clause « **WHERE** » doit précéder la clause « **GROUP BY** » et la clause « **HAVING** » doit lui succéder. Sachez que vous pouvez utiliser un alias de colonne dans une clause « **ORDER BY** », mais pas dans une autre clause « **WHERE** », « **GROUP BY** » ou « **HAVING** ».



## Le groupe à deux niveaux

Il est possible d'appliquer au résultat d'un select qui utilise le partitionnement de groupe un second niveau de fonction de groupe.

Pour comprendre l'exemple précédent il faut savoir que Oracle exécute la requête en deux pas :

1. Sélectionne et groupe les lignes conformément à la clause « **GROUP BY** ».



```
SQL> SELECT FONCTION, SUM(SALAIRE) FROM EMPLOYES GROUP BY PAYS;
```

PAYS	SUM( SALAIRE )
	-----
	492540
Norvège	35300
Allemagne	51200
Suède	31300
Espagne	36700
Mexique	28900
États-Unis	23100
Argentine	38900
Finlande	29800
Venezuela	37800
Irlande	36400
France	32200
Autriche	25600
Danemark	27500
Canada	35500
Pologne	32400
Suisse	36700
Belgique	27000
Brésil	23100
Royaume-Uni	42900
Italie	29000
Portugal	31600

2. Exécute les fonctions "verticales" sur l'ensemble des lignes obtenues dans le passage précédent comme si c'était des enregistrements provenant d'une table.



```
SQL> SELECT MAX(SUM(SALAIRE)), SUM(SUM(SALAIRE)) FROM EMPLOYES
2 GROUP BY PAYS;
```

MAX( SUM( SALAIRE ) )	SUM( SUM( SALAIRE ) )
-----	-----
492540	1185440

## Groupage ROLLUP

Un calcul d'agrégats porte toujours sur le regroupement indiqué par la clause « **GROUP BY** », mais il est parfois nécessaire d'effectuer un regroupement plus large afin de connaître d'autres valeurs.

La norme SQL 1999 introduit des nouvelles façons de réaliser des regroupements de données à l'aide des extensions à la clause « **GROUP BY** » il s'agit des deux fonctions de groupage « **ROLLUP** » et « **CUBE** » et une expression plus générale « **GROUPING SETS** ».

## ROLLUP

La fonction « **ROLLUP** » permet de générer des sous-totaux pour les attributs spécifiés, plus une ligne supplémentaire représentant le total global.

```
SELECT [ALL | DISTINCT]{*, [EXPRESSION1 [AS] ALIAS1[, ...]]}
```

```
FROM NOM_TABLE WHERE PREDICAT
```

```
GROUP BY ROLLUP ([NOM_COLONNE1 | EXPRESSION1][, ...]) ;
```

```
SQL> SELECT FONCTION, SUM(SALAIRE) FROM EMPLOYES GROUP BY ROLLUP(FONCTION);
```

FONCTION	SUM(SALAIRE)
Assistante commerciale	2000
Chef des ventes	8000
Représentant(e)	16561
Vice-Président	10000
	<b>36561</b>

Dans l'exemple, vous pouvez observer la fonction « **ROLLUP** » permettant de calculer le total pour la somme des salaires des employés.

```
SQL> SELECT FONCTION, PAYS, SUM(SALAIRE) FROM EMPLOYES  
2 GROUP BY ROLLUP(FONCTION, PAYS);
```

FONCTION	PAYS	SUM(SALAIRE)
Président		150000
<b>Président</b>		<b>150000</b>
Vice-Président		243000
<b>Vice-Président</b>		<b>243000</b>
Chef des ventes		83000
<b>Chef des ventes</b>		<b>83000</b>
Représentant(e)	Suède	31300
Représentant(e)	Brésil	23100
Représentant(e)	Canada	35500
Représentant(e)	France	32200
Représentant(e)	Italie	29000
Représentant(e)	Suisse	36700
Représentant(e)	Espagne	36700
Représentant(e)	Irlande	36400
Représentant(e)	Mexique	28900
Représentant(e)	Norvège	35300
Représentant(e)	Pologne	32400
Représentant(e)	Autriche	25600
Représentant(e)	Belgique	27000
Représentant(e)	Danemark	27500
Représentant(e)	Finlande	29800
Représentant(e)	Portugal	31600
Représentant(e)	Allemagne	51200
Représentant(e)	Argentine	38900
Représentant(e)	Venezuela	37800
Représentant(e)	États-Unis	23100
Représentant(e)	Royaume-Uni	42900
<b>Représentant(e)</b>		<b>692900</b>
Assistante commerciale		16540
<b>Assistante commerciale</b>		<b>16540</b>
		<b>1185440</b>



Dans l'exemple précédent, vous pouvez remarquer que le groupe est formé par les deux critères précisés dans la clause « **GROUP BY** » ; les résultats de la fonction « **SUM** » sont retournés aux niveaux suivants :

- par le pays **PAYS** et la fonction **FONCTION**;
- sous-totaux par la fonction de l'employé **FONCTION**;
- total global.



```
SQL> SELECT CODE_CLIENT, ANNEE, TRIMESTRE, SUM(PORT) PORT FROM COMMANDES
2 WHERE CODE_CLIENT IN ( 'ALFKI', 'ANTON' )
3 GROUP BY ROLLUP( CODE_CLIENT, ANNEE, TRIMESTRE ) ;
```

CODE_	ANNEE	TRIMESTRE	PORT
-----	-----	-----	-----
ALFKI	2010	1	2444,3
ALFKI	2010	2	2028,9
ALFKI	2010	3	1261,7
ALFKI	2010	4	1370,9
<b>ALFKI</b>	<b>2010</b>		<b>7105,8</b>
ALFKI	2011	1	1738,6
ALFKI	2011	2	1772,7
<b>ALFKI</b>	<b>2011</b>		<b>3511,3</b>
<b>ALFKI</b>			<b>10617,1</b>
ANTON	2010	1	877,6
ANTON	2010	2	2111,1
ANTON	2010	3	1650,4
ANTON	2010	4	2328,8
<b>ANTON</b>	<b>2010</b>		<b>6967,9</b>
ANTON	2011	1	2248,1
ANTON	2011	2	1526,1
ANTON	2011		3774,2
<b>ANTON</b>			<b>10742,1</b>
			<b>21359,2</b>

Dans l'exemple précédent, vous pouvez remarquer la fonction « **ROLLUP** » permettant d'obtenir les valeurs intermédiaires des regroupements pour toutes les combinaisons existantes et le total global.

### Conseil



Il est particulièrement utile d'avoir des sous-totaux dans une dimension hiérarchique de type temporelle ou géographique. Par exemple, une requête spécifiant « **ROLLUP** » (année, mois, jour) ou « **ROLLUP** » (pays, canton, ville).

Il est possible d'utiliser la fonction « **ROLLUP** » uniquement sur une partie des champs précisés dans la clause « **GROUP BY** ». Ainsi les regroupements sont effectués uniquement sur les champs passés comme arguments à la fonction « **ROLLUP** ».



```
SQL> SELECT NO_EMPLOYE, CODE_CLIENT, ANNEE, TRIMESTRE, SUM(PORT) PORT
2 FROM COMMANDES
3 WHERE CODE_CLIENT = 'ALFKI' AND NO_EMPLOYE IN (40,59)
4 GROUP BY NO_EMPLOYE, CODE_CLIENT, ROLLUP( ANNEE, TRIMESTRE );
```

NO_EMPLOYE	CODE_	ANNEE	TRIMESTRE	PORT
-----	-----	-----	-----	-----
40	ALFKI	2010	1	314,4
40	ALFKI	2010	2	302,6
40	ALFKI	2010	3	201,4
40	ALFKI	2010	4	228,2



40	ALFKI	2010		1046,6
40	ALFKI	2011	1	79,1
40	ALFKI	2011	2	590,4
40	ALFKI	2011		669,5
40	ALFKI			1716,1
59	ALFKI	2010	1	268,3
59	ALFKI	2010	2	189,2
59	ALFKI	2010	3	168,7
59	ALFKI	2010	4	64,6
59	ALFKI	2010		690,8
59	ALFKI	2011	1	436
59	ALFKI	2011	2	224,7
59	ALFKI	2011		660,7
59	ALFKI			1351,5

### Note

Il faut remarquer que la fonction « **ROLLUP** » effectue automatiquement un ordre de tri suivant l'ordre des champs précisés dans la clause « **GROUP BY** ».

Il faut noter également que la fonction « **ROLLUP** », comme toutes les fonctions de groupage, permet de hiérarchiser les éléments du groupage en utilisant les parenthèses.

Dans l'exemple suivant vous pouvez voir la requête qui affiche le client, l'année, le trimestre et le mois mais les regroupement sont calculés uniquement pour le client et l'année.

```
SQL> SELECT CODE_CLIENT, ANNEE, TRIMESTRE, MOIS, SUM(PORT) PORT
2 FROM COMMANDES
3 WHERE CODE_CLIENT BETWEEN 'ALFKI' AND 'ANATR'
4 GROUP BY CODE_CLIENT, ROLLUP( ANNEE,(TRIMESTRE,MOIS )) ;
```

CODE_	ANNEE	TRIMESTRE	MOIS	PORT
ALFKI	2010	1	1	926,7
ALFKI	2010	1	2	793
ALFKI	2010	1	3	724,6
ALFKI	2010	2	4	549,7
ALFKI	2010	2	5	900,7
ALFKI	2010	2	6	578,5
ALFKI	2010	3	7	687,5
ALFKI	2010	3	8	238,1
ALFKI	2010	3	9	336,1
ALFKI	2010	4	10	748,9
ALFKI	2010	4	11	622
ALFKI	2010			7105,8
ALFKI	2011	1	2	1287,9
ALFKI	2011	1	3	450,7
ALFKI	2011	2	4	977,8
ALFKI	2011	2	5	64,4
ALFKI	2011	2	6	730,5
ALFKI	2011			3511,3
ALFKI				10617,1
ANATR	2010	1	1	637
ANATR	2010	1	2	703,7
ANATR	2010	1	3	427,6
ANATR	2010	2	4	471,4
ANATR	2010	2	5	329,3
ANATR	2010	2	6	336,3

ANATR	2010	3	7	634,5
ANATR	2010	3	8	623,7
ANATR	2010	3	9	370,5
ANATR	2010	4	10	179,4
ANATR	2010	4	11	373,2
ANATR	2010	4	12	1034
<b>ANATR</b>	<b>2010</b>			<b>6120,6</b>
ANATR	2011	1	1	186,3
ANATR	2011	1	2	930,7
ANATR	2011	1	3	942,2
ANATR	2011	2	4	911,6
ANATR	2011	2	5	988,4
ANATR	2011	2	6	573,3
<b>ANATR</b>	<b>2011</b>			<b>4532,5</b>
<b>ANATR</b>				<b>10653,1</b>


## Groupage CUBE

### CUBE

La fonction « **CUBE** » va plus loin que « **ROLLUP** » ; elle permet de générer des sous-totaux pour toute combinaison d'attributs possibles parmi les argument de la fonctions, des totaux par attribut, et un total global.

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
FROM NOM_TABLE WHERE PREDICAT
GROUP BY CUBE ([NOM_COLONNE1|EXPRESSION1][,...]) ;
```

```
SQL> SELECT CODE_CLIENT, ANNEE, TRIMESTRE, SUM(PORT) PORT FROM COMMANDES
2 WHERE CODE_CLIENT = 'ALFKI'
3 GROUP BY ROLLUP( CODE_CLIENT, ANNEE, TRIMESTRE ) ;
```



CODE_	ANNEE	TRIMESTRE	PORT
-----	-----	-----	-----
ALFKI	2010	1	2444,3
ALFKI	2010	2	2028,9
ALFKI	2010	3	1261,7
ALFKI	2010	4	1370,9
<b>ALFKI</b>	<b>2010</b>		<b>7105,8</b>
ALFKI	2011	1	1738,6
ALFKI	2011	2	1772,7
<b>ALFKI</b>	<b>2011</b>		<b>3511,3</b>
<b>ALFKI</b>			<b>10617,1</b>
			<b>10617,1</b>

```
SQL> SELECT CODE_CLIENT, ANNEE, TRIMESTRE, SUM(PORT) PORT FROM COMMANDES
2 WHERE CODE_CLIENT = 'ALFKI'
3 GROUP BY CUBE( CODE_CLIENT, ANNEE, TRIMESTRE ) ;
```

CODE_	ANNEE	TRIMESTRE	PORT
-----	-----	-----	-----
			10617,1
		1	4182,9
		2	3801,6

		3	1261,7
		4	1370,9
	2010		7105,8
	2010	1	2444,3
	2010	2	2028,9
	2010	3	1261,7
	2010	4	1370,9
	2011		3511,3
	2011	1	1738,6
	2011	2	1772,7
ALFKI			10617,1
ALFKI		1	4182,9
ALFKI		2	3801,6
ALFKI		3	1261,7
ALFKI		4	1370,9
ALFKI	2010		7105,8
ALFKI	2010	1	2444,3
ALFKI	2010	2	2028,9
ALFKI	2010	3	1261,7
ALFKI	2010	4	1370,9
ALFKI	2011		3511,3
ALFKI	2011	1	1738,6
ALFKI	2011	2	1772,7

Dans l'exemple précédent, vous pouvez observer la comparaison entre la fonction « **CUBE** » et la fonction « **ROLLUP** ». Le groupe est formé par les trois critères le client, l'année et le mois précisés dans la clause « **GROUP BY** » ; les résultats de la fonction « **SUM** » sont retournés dans le cas de la fonction « **CUBE** » aux niveaux suivants :

- total global
- sous-totaux par Trimestre
- sous-totaux par Année
- sous-totaux par Trimestre pour l'Année
- sous-totaux par Client
- sous-totaux par Trimestre pour le Client
- sous-totaux par Année pour le Client
- sous-totaux par Trimestre pour l'Année et par Client



### Attention


Dans les versions Oracle 10g et suivantes les lignes de calculs supplémentaires pour la fonction « **CUBE** » sont affichées au début ; par contre pour les versions antérieures ces lignes sont affichées à la fin. Pour ne pas avoir des affichages différents suivant la version ou si vous souhaitez un autre ordre d'affichage, vous pouvez utiliser la clause « **ORDER BY** ».

## GROUPING SETS

Dans une clause « **GROUP BY** » avec la fonction « **GROUPING SETS** », on peut spécifier le jeu de groupes que l'on désire créer. Ceci permet des spécifications précises au travers de dimensions multiples sans devoir calculer le « **CUBE** » en entier.

La fonction « **GROUPING SETS** » permet de définir plusieurs groupes dans la même requête.

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
FROM NOM_TABLE WHERE PREDICAT
GROUP BY GROUPING SETS ([NOM_COLONNE1|EXPRESSION1[,...]]);
SQL> SELECT CODE_CLIENT, ANNEE, MOIS, SUM(PORT) PORT FROM COMMANDES
2 WHERE CODE_CLIENT BETWEEN 'ALFKI' AND 'BLONP'
3 GROUP BY GROUPING SETS( CODE_CLIENT, ANNEE, MOIS)
4 ORDER BY CODE_CLIENT, ANNEE, MOIS;
```



CODE_	ANNEE	MOIS	PORT
ALFKI			10617,1
ANATR			10653,1
ANTON			10742,1
AROUT			12944,5
BERGS			11969,2
BLAUS			9724,5
BLONP			10413,4
	2010		44890,1
	2011		32173,8
		1	5979,3
		2	10228
		3	9514,7
		4	10183,1
		5	9474
		6	8575,9
		7	3654,5
		8	3296,1
		9	2642,4
		10	5282,2
		11	4086
		12	4147,7

Comme on peut le constater dans cet exemple, les résultats ne sont que ceux des groupages, le premier sur Client, le deuxième sur Année, le troisième sur le Mois et le quatrième sur l'ensemble global. En fait, cette écriture synthétique correspond à la concaténation des résultats des trois requêtes suivantes :

```
SELECT CODE_CLIENT "Client", SUM(PORT) "Port"
FROM COMMANDES GROUP BY CODE_CLIENT;
SELECT ANNEE "Année", SUM(PORT) "Port"
FROM COMMANDES GROUP BY ANNEE;
SELECT MOIS "Mois", SUM(PORT) "Port"
FROM COMMANDES GROUP BY MOIS;
```

### Conseil

La fonction « **GROUPING SETS** » permet de hiérarchiser les éléments du groupage en utilisant les parenthèses.

On peut ainsi imbriquer des fonctions « **ROLLUP** » ou des fonctions « **CUBE** ».

L'exemple suivant montre l'utilisation des arguments composés à l'aide des parenthèses. Chaque parenthèse est un argument pour la fonction « **GRUPING SETS** » et la troisième parenthèse constitue le cumul total pour l'ensemble des enregistrements.



```
SQL> SELECT CODE_CLIENT, ANNEE, TRIMESTRE, SUM(PORT) PORT FROM COMMANDES
2 WHERE CODE_CLIENT BETWEEN 'ALFKI' AND 'AROUT'
3 GROUP BY GROUPING SETS( (CODE_CLIENT,ANNEE), (ANNEE,TRIMESTRE), ())
4 ORDER BY CODE_CLIENT, ANNEE, TRIMESTRE;
```

CODE_	ANNEE	TRIMESTRE	PORT
ALFKI	2010		7105,8
ALFKI	2011		3511,3
ANATR	2010		6120,6
ANATR	2011		4532,5
ANTON	2010		6967,9
ANTON	2011		3774,2
AROUT	2010		7440,5
AROUT	2011		5504
	2010	1	6873,9
	2010	2	7011,6
	2010	3	6232,7
	2010	4	7516,6
	2011	1	8045,8
	2011	2	9276,2
			<b>44956,8</b>

```
SQL> SELECT CODE_CLIENT,ANNEE,TRIMESTRE,MOIS,SUM(PORT) PORT FROM COMMANDES
2 WHERE CODE_CLIENT BETWEEN 'ALFKI' AND 'ANTON'
3 GROUP BY GROUPING SETS(ROLLUP(CODE_CLIENT,ANNEE),
4 ROLLUP(ANNEE,TRIMESTRE,MOIS))
5 ORDER BY CODE_CLIENT, ANNEE, TRIMESTRE, MOIS;
```

CODE_	ANNEE	TRIMESTRE	MOIS	PORT
ALFKI	2010			7105,8
ALFKI	2011			3511,3
<b>ALFKI</b>				<b>10617,1</b>
ANATR	2010			6120,6
ANATR	2011			4532,5
<b>ANATR</b>				<b>10653,1</b>
ANTON	2010			6967,9
ANTON	2011			3774,2
<b>ANTON</b>				<b>10742,1</b>
	2010	1	1	1563,7
	2010	1	2	1816,2
	2010	1	3	1710,3
	<b>2010</b>	<b>1</b>		<b>5090,2</b>
	2010	2	4	1471,5
	2010	2	5	2232,6
	2010	2	6	1572,9
	<b>2010</b>	<b>2</b>		<b>5277</b>
	2010	3	7	1419,7
	2010	3	8	1358,9
	2010	3	9	1762,2
	<b>2010</b>	<b>3</b>		<b>4540,8</b>
	2010	4	10	1786
	2010	4	11	1845
	2010	4	12	1655,3

2010	4		5286,3
2010			20194,3
2011	1	1	872,4
2011	1	2	3657,3
2011	1	3	1516,2
2011	1		6045,9
2011	2	4	1889,4
2011	2	5	1694,4
2011	2	6	2188,3
2011	2		5772,1
2011			11818
			32012,3
			32012,3

## Fonctions de groupes

Deux problèmes surviennent lors de l'utilisation de « **ROLLUP** » et « **CUBE** ». Premièrement, il est très difficile de déterminer au niveau de la présentation du résultat quels sont les sous-totaux ou quel est le niveau exact d'agrégation. Il nous faut un moyen pour déterminer quelles sont les lignes qui affichent les sous-totaux. Deuxièmement, comment fait-on pour différencier les valeurs « **NULL** » stockées des valeurs « **NULL** » créées par « **ROLLUP** » ou « **CUBE** ».

Il existe à partir de la version Oracle 8i la fonction « **GROUPING** », et dès la version Oracle 9i, les fonctions « **GROUPING\_ID** » et « **GROUP\_ID** ».

### GROUPING

La fonction « **GROUPING** » permet d'identifier si la colonne argument doit être traitée pour sa valeur propre « **0** » ou pour un ensemble de plusieurs valeurs « **1** ».



```
SQL> SELECT GROUPING(NO_FOURNISSEUR) "GF",GROUPING(CODE_CATEGORIE) "GC",
2      NO_FOURNISSEUR "Fournisseur",CODE_CATEGORIE "Catégorie",
3      SUM(UNITES_STOCK) "Stock" FROM PRODUITS
4  GROUP BY GROUPING SETS(ROLLUP(NO_FOURNISSEUR), CODE_CATEGORIE)
5  ORDER BY NO_FOURNISSEUR, CODE_CATEGORIE;
```

GF	GC	Fournisseur	Catégorie	Stock
0	1	1		69
0	1	2		213
0	1	3		141
0	1	4		255
0	1	5		228
0	1	6		298
0	1	7		130
0	1	8		149
0	1	9		165
0	1	10		
0	1	11		240
0	1	12		179
0	1	13		75
0	1	14		408
0	1	15		164
0	1	16		423
0	1	17		324

0	1	18	246
0	1	19	328
0	1	20	64
0	1	21	180
0	1	22	51
0	1	23	332
0	1	24	98
0	1	25	176
0	1	26	197
0	1	27	192
0	1	28	98
0	1	29	130
1	0	1	874
1	0	2	1032
1	0	3	466
1	0	4	433
1	0	5	642
1	0	6	136
1	0	7	589
1	0	8	701
1	0	9	460
1	0	10	220
1	1		5553

Vous pouvez utiliser les fonctions conditionnelles pour une présentation plus soignée.



```
SQL> SELECT CODE_CLIENT "Client",
2      CASE GROUPING(ANNEE)
3        WHEN 0 THEN TO_CHAR(ANNEE)
4        ELSE 'Toutes les Années'
5        END "Années",
6      CASE GROUPING( MOIS)
7        WHEN 0 THEN TO_CHAR(MOIS)
8        ELSE
9          CASE GROUPING(ANNEE)
10           WHEN 0 THEN 'Tous les Mois'
11           ELSE 'Toutes les Années'
12          END
13      END "Mois", SUM(PORT) "Port" FROM COMMANDES
14 WHERE CODE_CLIENT = 'HUNGO'
15 GROUP BY GROUPING SETS ( CODE_CLIENT, ROLLUP(ANNEE,MOIS));
```

Clien	Années	Mois	Port
HUNGO	Toutes les Années	Toutes les Années	12101
	2010	1	687,5
	2010	2	620,6
	2010	3	315,2
	2010	4	502,5
	2010	5	418,9
	2010	6	473,2
	2010	7	751,9
	2010	8	558,9
	2010	9	232,8
	2010	10	771,8
	2010	11	680,9
	2010	12	647

2010	Tous les Mois	6661,2
2011	1	479,3
2011	2	931,8
2011	3	1179,4
2011	4	858,1
2011	5	788,7
2011	6	1202,5
2011	Tous les Mois	5439,8
Toutes les Années	Toutes les Années	12101

### Note



Pour trouver le niveau du « **GROUP BY** » pour une ligne particulière, une requête doit retourner une information de la fonction « **GROUPING** » pour chaque colonne du « **GROUP BY** ». Dans ce cas, avec la fonction « **GROUPING** », chaque colonne du « **GROUP BY** » requière une autre colonne utilisant la fonction « **GROUPING** ».

Pour résoudre ce problème, Oracle9i introduit la fonction « **GROUPING\_ID** » qui permet de déterminer le niveau exact du « **GROUP BY** ».

## GROUPING\_ID

La fonction « **GROUPING\_ID** » permet d'identifier si une des colonnes passée en argument est traitée pour un ensemble de plusieurs valeurs.

**GROUPING\_ID ( EXPRESSION[,...] )**

La fonction retourne une valeur numérique décimale calculée à partir des valeurs des fonctions « **GROUPING** » pour les expressions passées comme arguments.

<i>GROUPING</i>			<i>GROUPING_ID</i>
<i>CLIENT</i> $2^2=4$	<i>ANNEE</i> $2^1=2$	<i>TRIMESTRE</i> $2^0=1$	
0	0	0	0
0	0	1	1
0	1	1	3
1	1	1	7



```
SQL> SELECT GROUPING(NO_FOURNISSEUR) "GF", GROUPING(ANNEE) "GA",
2  GROUPING(TRIMESTRE) "GT", GROUPING_ID(CODE_CLIENT, ANNEE, TRIMESTRE) "GI",
3  CODE_CLIENT CLIENT, ANNEE, TRIMESTRE, SUM(PORT) PORT
4  FROM COMMANDES WHERE CODE_CLIENT BETWEEN 'ALFKI' AND 'ANTON'
5  GROUP BY ROLLUP(CODE_CLIENT, ANNEE, TRIMESTRE);
```

GC	GA	GT	GI	CLIENT	ANNEE	TRIMESTRE	PORT
0	0	0	0	ALFKI	2010	1	2444,3
0	0	0	0	ALFKI	2010	2	2028,9
0	0	0	0	ALFKI	2010	3	1261,7
0	0	0	0	ALFKI	2010	4	1370,9
0	0	1	1	ALFKI	2010		7105,8
0	0	0	0	ALFKI	2011	1	1738,6
0	0	0	0	ALFKI	2011	2	1772,7



0	0	1	1	ALFKI	2011		3511,3
0	1	1	3	ALFKI			10617,1
0	0	0	0	ANATR	2010	1	1768,3
0	0	0	0	ANATR	2010	2	1137
0	0	0	0	ANATR	2010	3	1628,7
0	0	0	0	ANATR	2010	4	1586,6
0	0	1	1	ANATR	2010		6120,6
0	0	0	0	ANATR	2011	1	2059,2
0	0	0	0	ANATR	2011	2	2473,3
0	0	1	1	ANATR	2011		4532,5
0	1	1	3	ANATR			10653,1
0	0	0	0	ANTON	2010	1	877,6
0	0	0	0	ANTON	2010	2	2111,1
0	0	0	0	ANTON	2010	3	1650,4
0	0	0	0	ANTON	2010	4	2328,8
0	0	1	1	ANTON	2010		6967,9
0	0	0	0	ANTON	2011	1	2248,1
0	0	0	0	ANTON	2011	2	1526,1
0	0	1	1	ANTON	2011		3774,2
0	1	1	3	ANTON			10742,1
1	1	1	7				32012,3

Dans l'exemple précédent, la fonction utilisée est « **ROLLUP** » qui considère les arguments comme faisant partie d'une hiérarchie, ainsi les valeurs de la fonction « **GROUPING\_ID** » sont toujours impaires, alors que dans le cas d'une fonction « **CUBE** » toutes les valeurs sont possibles.



```
SQL> SELECT GROUPING(NO_FOURNISSEUR) "GF", GROUPING(ANNEE) "GA",
2  GROUPING(TRIMESTRE) "GT", GROUPING_ID(CODE_CLIENT,ANNEE,TRIMESTRE) "GI
3  CODE_CLIENT CLIENT, ANNEE, TRIMESTRE, SUM(PORT) PORT
4  FROM COMMANDES WHERE CODE_CLIENT BETWEEN 'ALFKI' AND 'ANTON'
5  GROUP BY CUBE(CODE_CLIENT,ANNEE,TRIMESTRE);
```

GC	GA	GT	GI	CLIENT	ANNEE	TRIMESTRE	PORT
1	1	1	7				32012,3
1	1	0	6			1	11136,1
1	1	0	6			2	11049,1
1	1	0	6			3	4540,8
1	1	0	6			4	5286,3
1	0	1	5		2010		20194,3
1	0	0	4		2010	1	5090,2
1	0	0	4		2010	2	5277
1	0	0	4		2010	3	4540,8
1	0	0	4		2010	4	5286,3
1	0	1	5		2011		11818
1	0	0	4		2011	1	6045,9
1	0	0	4		2011	2	5772,1
0	1	1	3	ALFKI			10617,1
0	1	0	2	ALFKI		1	4182,9
0	1	0	2	ALFKI		2	3801,6
0	1	0	2	ALFKI		3	1261,7
0	1	0	2	ALFKI		4	1370,9
0	0	1	1	ALFKI	2010		7105,8
0	0	0	0	ALFKI	2010	1	2444,3
...							



## Attention

La fonction « **GROUPING\_ID** » calcule le niveau de regroupement à partir des arguments que vous lui avez transmis et non à partir des informations des fonctions « **ROLLUP** », « **CUBE** » ou « **GROUPING SETS** ».

Ainsi vous devez faire très attention aux arguments que vous saisissez pour cette fonction.



```
SQL> SELECT GROUPING_ID(CODE_CLIENT,ANNEE) "GCA",
2          GROUPING_ID(ANNEE,TRIMESTRE) "GAT",
3          GROUPING_ID(CODE_CLIENT,TRIMESTRE) "GCT",
4          GROUPING_ID(CODE_CLIENT,ANNEE,TRIMESTRE) "GI",
5          CODE_CLIENT CLIENT, ANNEE, TRIMESTRE, SUM(PORT) PORT
6 FROM COMMANDES WHERE CODE_CLIENT BETWEEN 'ALFKI' AND 'ANTON'
7 GROUP BY CUBE(CODE_CLIENT,ANNEE,TRIMESTRE);
```

GCA	GAT	GCT	GI	CLIENT	ANNEE	TRIMESTRE	PORT
3	3	3	7				32012,3
3	2	2	6			1	11136,1
3	2	2	6			2	11049,1
3	2	2	6			3	4540,8
3	2	2	6			4	5286,3
2	1	3	5		2010		20194,3
2	0	2	4		2010	1	5090,2
2	0	2	4		2010	2	5277
2	0	2	4		2010	3	4540,8
2	0	2	4		2010	4	5286,3
2	1	3	5		2011		11818
2	0	2	4		2011	1	6045,9
2	0	2	4		2011	2	5772,1
1	3	1	3	ALFKI			10617,1
1	2	0	2	ALFKI		1	4182,9
1	2	0	2	ALFKI		2	3801,6
...							

## GROUP\_ID

Les fonctions « **ROLLUP** », « **CUBE** » et « **GROUPING SETS** » offrent plus de puissance et de flexibilité, ils donnent aussi des résultats plus complexes incluant des groupes dupliqués.

La fonction « **GROUP\_ID** » permet d'identifier la duplication des groupes ; elle assigne la valeur « **0** » à toutes les lignes du premier jeu de lignes et tous les autres jeux de données dupliqués pour un groupe particulier sont mis à une valeur plus grande, en débutant avec « **1** ».

**GROUP\_ID ( )**



```
SQL> SELECT GROUP_ID( ) GROUP_ID,CODE_CLIENT,ANNEE,TRIMESTRE,SUM(PORT) PORT
2 FROM COMMANDES WHERE CODE_CLIENT BETWEEN 'ALFKI' AND 'ANTON'
3 GROUP BY GROUPING SETS(CUBE(CODE_CLIENT,ANNEE),
4                        CUBE(ANNEE,TRIMESTRE));
```

GROUP_ID	CODE_	ANNEE	TRIMESTRE	PORT
0	ALFKI	2010		7105,8
0	ALFKI	2011		3511,3
0	ALFKI			10617,1
0	ANATR	2010		6120,6
0	ANATR	2011		4532,5

0	ANATR		10653,1
0		1	8010,4
0		2	7411,9
0		3	2890,4
0		4	2957,5
0	2010	1	4212,6
0	2011	1	3797,8
0	2010	2	3165,9
0	2011	2	4246
0	2010	3	2890,4
0	2010	4	2957,5
0	2010		13226,4
0	2011		8043,8
1	2010		13226,4
1	2011		8043,8
0			21270,2
1			21270,2

La fonction « **GROUP\_ID** » peut-être utilisée dans la clause « **HAVING** », et ainsi vous pouvez filtrer les enregistrements en double.



```
SQL> SELECT GROUP_ID() GROUP_ID, CODE_CLIENT, ANNEE, TRIMESTRE, SUM(PORT) PORT
2 FROM COMMANDES WHERE CODE_CLIENT BETWEEN 'ALFKI' AND 'ANATR'
3 GROUP BY GROUPING SETS( CUBE(CODE_CLIENT, ANNEE),
4 CUBE(ANNEE, TRIMESTRE)) HAVING GROUP_ID() = 0;
```

GROUP_ID	CODE_	ANNEE	TRIMESTRE	PORT
0	ALFKI	2010		7105,8
0	ALFKI	2011		3511,3
0	ALFKI			10617,1
0	ANATR	2010		6120,6
0	ANATR	2011		4532,5
0	ANATR			10653,1
0			1	8010,4
0			2	7411,9
0			3	2890,4
0			4	2957,5
0		2010	1	4212,6
0		2011	1	3797,8
0		2010	2	3165,9
0		2011	2	4246
0		2010	3	2890,4
0		2010	4	2957,5
0		2010		13226,4
0		2011		8043,8
0				21270,2

- *La jointure*
- *JOIN ON*
- *USING*
- *NATURAL JOIN*
- *FULL OUTER JOIN*

# 7

## Les requêtes multi-tables



### Objectifs

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- Effectuer des requêtes multi-tables.
- Sélectionner des lignes sans créer de produit cartésien.
- Effectuer des interrogations avec des aliasses de tables.
- Effectuer des interrogations en utilisant la syntaxe d'Oracle pour les jointures.
- Effectuer des interrogations en utilisant la syntaxe ANSI SQL :1999.



### Contenu

Requêtes multi-tables	7-2	L'opérateur NATURAL JOIN	7-10
La jointure sans condition	7-3	L'auto-jointure	7-11
La jointure avec condition	7-4	La jointure externe	7-12
L'opérateur JOIN ON	7-6	L'opérateur OUTER JOIN	7-14
L'opérateur JOIN USING	7-9		

# Requêtes multi-tables

Jusqu'ici, nous avons extrait des données, brutes ou dérivées, issues d'une seule table. Nous examinerons dans cette section comment coupler les lignes de deux ou plusieurs tables afin d'en extraire des données corrélées.

Dans un environnement réel de production, les informations utiles sont souvent contenues dans plusieurs tables. Par exemple, vous pouvez avoir besoin du nom des catégories de produits alors que la table `PRODUITS` ne contient que les codes des catégories. Vous devez alors coupler les deux tables, `PRODUITS` et la table `CATEGORIES`. Les bases de données relationnelles permettent dans leur principe d'associer deux ou plusieurs tables par des colonnes communes et qui participent à la formation de clés.

Il existe deux types de clés, primaire et étrangère. Une clé primaire, composée d'un ou de plusieurs champs, permet d'identifier de façon unique un enregistrement de la table. Dans la table `CATEGORIES`, la clé primaire est représentée par une seule colonne, `CODE_CATEGORIE`. La table `PRODUITS` contient aussi cette colonne, mais il s'agit pour elle d'une clé étrangère. Une clé étrangère permet d'extraire des informations contenues dans une autre table (étrangère). Une telle opération d'association de tables porte le nom de **jointure**.

Il existe deux manières d'écrire en SQL la jointure entre plusieurs tables de la base de données. La forme historique et la plus utilisée est la syntaxe de jointure relationnelle (basé sur la norme ANSI SQL/86) ; la deuxième syntaxe est une jointure plus verbale qui respecte la norme ANSI SQL/92 et qui a été introduite dans Oracle 9i.

La syntaxe de la jointure relationnelle est la suivante :

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
FROM NOM_TABLE1, NOM_TABLE2 [,... ]WHERE prédicat de jointure
```

La deuxième syntaxe vous permet de sélectionner des colonnes dans plusieurs tables en développant la clause « **FROM** » de l'instruction « **SELECT** » respectant la syntaxe suivante :

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
FROM TABLE1[{ { INNER | {LEFT | RIGHT | FULL} OUTER } ] JOIN TABLE2
{ ON (TABLE1.NOM_COLONNE = TABLE2.NOM_COLONNE[,...])
| USING (NOM_COLONNE1[,...]) }
| { CROSS JOIN | NATURAL[{INNER|{LEFT | RIGHT | FULL} OUTER } ]}
JOIN TABLE2}{[,... ] ...
```

<b>CROSS JOIN</b>	Le résultat est le même que celui d'une requête sans condition qui affiche pour chaque ligne de la première table l'ensemble des lignes de la deuxième.
<b>JOIN ON</b>	La jointure entre les tables est effectuée à l'aide de la condition spécifiée.
<b>JOIN USING</b>	La jointure entre les tables est effectuée à l'aide de la ou des colonnes spécifiées.
<b>NATURAL JOIN</b>	La jointure entre les tables est effectuée à l'aide des colonnes qui portent le même nom.
<b>OUTER JOIN</b>	La jointure externe entre les tables est effectuée à l'aide de la condition spécifiée.

Les jointures permettent de définir le mode d'association d'une ou plusieurs tables, suivant les opérateurs utilisés et sont classifiées comme suit :

- La jointure sans condition. Il s'agit d'assembler les enregistrements de la première table avec ceux des suivantes sans définir aucune règle d'assemblage ; c'est un produit cartésien.
- La jointure avec condition. Il existe deux types de jointures : l'équijointure, la plus connue, qui utilise un opérateur d'égalité, et l'inéquijointure qui fait appel aux autres opérateurs ( <>, >, <, >=, <=, **BETWEEN**, **IN**, **LIKE** ... ).
- L'auto-jointure. Il s'agit d'un cas particulier de l'équijointure qui relie une table à elle-même.
- La jointure externe. Ce type de jointure permet de récupérer des enregistrements qui ne correspondent pas aux critères de jointure.

Dans la suite du module, sont présentés les différents types de jointure, d'abord avec la syntaxe relationnelle, suivie par les syntaxes de la norme AINSI SQL/92.

## La jointure sans condition

Une requête sans condition affiche pour chaque ligne de la première table l'ensemble des lignes de la deuxième, si d'autres tables sont définies dans la clause « **FROM** », pour chaque ligne du résultat précédent les lignes de la table suivante, etc.



```
SQL> SELECT COUNT(*) FROM PRODUITS;
```

```

COUNT(*)
-----
      120

```

```
SQL> SELECT COUNT(*) FROM CATEGORIES;
```

```

COUNT(*)
-----
       10

```

```
SQL> SELECT NOM_PRODUIT, NOM_CATEGORIE FROM PRODUITS, CATEGORIES;
```

NOM_PRODUIT	NOM_CATEGORIE
Chai	Boissons
Chang	Boissons
Aniseed Syrup	Boissons
Chef Anton's Cajun Seasoning	Boissons
Grandma's Boysenberry Spread	Boissons
Uncle Bob's Organic Dried Pears	Boissons
...	

1200 ligne(s) sélectionnée(s).

La requête précédente risque cependant d'être extrêmement coûteuse (le résultat contiendrait ici  $120 \times 10 = 1200$  lignes) et n'offrirait aucun intérêt, cette opération est appelée le **produit cartésien**.

L'opérateur « **CROSS JOIN** » est un produit cartésien ; il donne le même résultat que celui d'une requête sans condition. La syntaxe est la suivante :

```

SELECT [ALL | DISTINCT]{*, [EXPRESSION1 [AS] ALIAS1[, ...]]}
FROM NOM_TABLE1 CROSS JOIN NOM_TABLE2 ;

```



```
SQL> SELECT COUNT(*) FROM DETAILS_COMMANDES;

COUNT(*)
-----
476091

SQL> SELECT COUNT(*) FROM PRODUITS CROSS JOIN DETAILS_COMMANDES;

COUNT(*)
-----
57130920
```

## La jointure avec condition

Pour coupler deux tables, il faut d'abord préciser les tables dans la clause « **FROM** », ainsi que la règle d'association des lignes de ces deux tables, les conditions correspondantes dans la clause « **WHERE** », dont les valeurs sont extraites. Cette règle se présente sous la forme d'une égalité des valeurs de deux ensembles d'une ou plusieurs colonnes, généralement les clés primaires comparées aux clés étrangères. Ce type de jointure est l'équijointure.



```
SQL> SELECT A.CODE_CATEGORIE CAT, NOM_PRODUIT "Produit",
2  NOM_CATEGORIE "Categorie" FROM PRODUITS A, CATEGORIES B
3  WHERE A.CODE_CATEGORIE = B.CODE_CATEGORIE ORDER BY NOM_PRODUIT;
```

CAT	Produit	Categorie
6	Alice Mutton	Viandes
7	Amandes	Produits secs
2	Aniseed Syrup	Condiments
1	Beer	Boissons
8	Boston Crab Meat	Poissons et fruits de mer
9	Boysenberry Spread	Conserves
3	Brownie Mix	Desserts
2	Cajun Seasoning	Condiments
3	Cake Mix	Desserts
...		

La requête précédente affiche, pour chaque produit, le nom de la catégorie correspondante. Dans la clause « **FROM** », vous remarquerez les alias A et B pour les tables PRODUITS et CATEGORIES, utilisées pour faciliter l'écriture de la requête et pour lever certaines ambiguïtés, comme dans notre cas où les deux tables possèdent une colonne de même nom CODE\_CATEGORIE.



```
SQL> SELECT NOM||'|'||PRENOM "Vendeur", SOCIETE "Client",
2  TO_CHAR( DATE_COMMANDE,'DD/MM/YYYY') "Commande", PORT "Port"
3  FROM CLIENTS A, EMPLOYES B, COMMANDES C
4  WHERE A.CODE_CLIENT = C.CODE_CLIENT AND B.NO_EMPLOYE = C.NO_EMPLOYE
5  AND DATE_COMMANDE > '29/06/2011';
```

Vendeur	Client	Commande	Port
Cazade Anne-Claire	Piccolo und mehr	30/06/2011	59.6
Cazade Anne-Claire	Ernst Handel	30/06/2011	83.9
Twardowski Colette	Wolski Zajazd	30/06/2011	54.9
Coutou Myriam	Old World Delicatessen	30/06/2011	98.5

Thomas Patricia	Suprêmes délices	30/06/2011	89.6
Marielle Michel	Toms Spezialitäten	30/06/2011	68.2
Marielle Michel	QUICK-Stop	30/06/2011	74.8
Thimoleon Georges	Blauer See Delikatessen	30/06/2011	79.1
Viry Yvan	Ricardo Adocicados	30/06/2011	64.9
Viry Yvan	Que Delícia	30/06/2011	68.0
Tourtrel Nicole	Hungry Owl All-Night Grocers	30/06/2011	66.3
Regner Charles	France restauration	30/06/2011	93.0
Cleret Doris	Océano Atlántico Ltda.	30/06/2011	79.8
Poidatz Benoît	Queen Cozinha	30/06/2011	81.4
Capharsie Gérard	GROSELLA-Restaurante	30/06/2011	75.6
Perny Sylvie	Centro comercial Moctezuma	30/06/2011	90.8
Gerard Sylvie	Antonio Moreno Taquería	30/06/2011	63.1
Brunet Jean-Luc	Mère Paillarde	30/06/2011	63.0
Jenny Michel	Comércio Mineiro	30/06/2011	85.1
Gregoire Renée	Bon app'	30/06/2011	98.8
Mure Guy	Alfreds Futterkiste	30/06/2011	64.8
Mure Guy	Ottilies Käseladen	30/06/2011	54.7
Kessler Marc	North/South	30/06/2011	93.3
Urbaniak Isabelle	Rattlesnake Canyon Grocery	30/06/2011	94.0
Clement Valérie	Bólido Comidas preparadas	30/06/2011	59.7

Les conditions telles que `A.CODE_CLIENT = C.CODE_CLIENT` sont appelées conditions de jointure, car elles régissent les associations entre les lignes de `COMMANDES` et `CLIENTS`. Le champ `CODE_CLIENT` est clé primaire dans la table `CLIENTS` et clé étrangère dans la table `COMMANDES`.

Techniquement, cependant, il s'agit d'une condition ordinaire appliquée aux colonnes de chaque couple de lignes, et qui peut apparaître au milieu d'autres conditions de sélection, comme dans la requête précédente, pour limiter le résultat aux commandes ultérieures au 29 juin 2011.

La mise en relation des colonnes communes à deux tables ne s'établit pas forcément par l'intermédiaire d'une opération d'égalité. On parle alors d'inéquijointure.

Dans la requête suivante on recherche les produits et le nombre des commandes passées pour toutes les commandes qui ont des produits avec un `PRIX_UNITAIRE` supérieur au `PRIX_UNITAIRE` du produit 2.



```
SQL> SELECT DC.REF_PRODUIT, COUNT( DISTINCT NO_COMMANDE) COM
2 FROM DETAILS_COMMANDES DC, PRODUITS P
3 WHERE DC.PRIX_UNITAIRE > P.PRIX_UNITAIRE AND P.REF_PRODUIT = 2
4 GROUP BY DC.REF_PRODUIT;
```

REF_PRODUIT	COM
-----	-----
120	3799
20	4133
96	4271

Vous pouvez également utiliser un autre opérateur comme par exemple l'opérateur « **BETWEEN** » pour retrouver les produits avec un `PRIX_UNITAIRE` compris entre plus ou moins 5% du `PRIX_UNITAIRE` du produit 57.



```
SQL> SELECT DC.REF_PRODUIT, COUNT( DISTINCT NO_COMMANDE) COM
2 FROM DETAILS_COMMANDES DC, PRODUITS P
3 WHERE DC.PRIX_UNITAIRE BETWEEN P.PRIX_UNITAIRE*0.95 AND
4 P.PRIX_UNITAIRE*1.05 AND P.REF_PRODUIT = 57
5 GROUP BY DC.REF_PRODUIT;
```

REF_PRODUIT	COM
-----	-----



120	3799
20	4133
96	4271
111	3997
62	3722

Les conditions utilisées pour les jointures peuvent être autre que les clés primaires et les clés étrangères. Par exemple on veut afficher les sociétés clientes et les fournisseurs qui habitent dans la même ville.



```
SQL> SELECT C.SOCIETE CLIENT, F.SOCIETE FOURNISSEUR, C.VILLE
2 FROM CLIENTS C, FOURNISSEURS F WHERE C.VILLE = F.VILLE;
```

CLIENT	FOURNISSEUR	VILLE
Consolidated Holdings	Exotic Liquids	London
Eastern Connection	Exotic Liquids	London
Familia Arquibaldo	Refrescos Americanas LTDA	São Paulo
Lehmans Marktstand	Plutzer Lebensmittelgroßmärkte AG	Frankfurt
Mère Paillarde	Ma Maison	Montréal
North/South	Exotic Liquids	London
Paris spécialités	Aux joyeux ecclésiastiques	Paris
Queen Cozinha	Refrescos Americanas LTDA	São Paulo
Seven Seas Imports	Exotic Liquids	London
Spécialités du monde	Aux joyeux ecclésiastiques	Paris
Tradição Hipermercados	Refrescos Americanas LTDA	São Paulo
Alfreds Futterkiste	Heli Süßwaren GmbH Co. KG	Berlin
Around the Horn	Exotic Liquids	London
B's Beverages	Exotic Liquids	London
Comércio Mineiro	Refrescos Americanas LTDA	São Paulo

## L'opérateur JOIN ON

L'opérateur « **JOIN ON** » effectue la jointure entre deux tables en se servant des conditions spécifiées respectant la syntaxe suivante :

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
```

```
FROM NOM_TABLE1[ JOIN NOM_TABLE2 ON
```

```
(NOM_TABLE1.NOM_COLONNE = NOM_TABLE2.NOM_COLONNE)] ;
```



```
SQL> SELECT SOCIETE FOURNISSEUR, NOM_CATEGORIE, SUM( UNITES_STOCK) UC
2 FROM CATEGORIES C JOIN PRODUITS P
3 ON ( C.CODE_CATEGORIE = P.CODE_CATEGORIE)
4 JOIN FOURNISSEURS F ON ( P.NO_FOURNISSEUR = F.NO_FOURNISSEUR)
5 WHERE UNITES_STOCK <> 0 GROUP BY NOM_CATEGORIE, SOCIETE;
```

FOURNISSEUR	NOM_CATEGORIE	UC
Tokyo Traders	Conserves	40
Tokyo Traders	Desserts	20
Pavlova, Ltd.	Desserts	49
Specialty Biscuits, Ltd.	Produits secs	75
Plutzer Lebensmittelgroßmärkte AG	Condiments	32
Plutzer Lebensmittelgroßmärkte AG	Boissons	125
Formaggi Fortini s.r.l.	Viande en conserve	50

Karkki Oy	Boissons	57
Escargots Nouveaux	Condiments	40
Nouvelle-Orléans Cajun Delights	Produits secs	20
Cooperativa de Quesos 'Las Cabras	Produits secs	40
...		

```
SQL> SELECT SOCIETE FOURNISSEUR, NOM_CATEGORIE, SUM( UNITES_STOCK) UC
2 FROM CATEGORIES C,PRODUITS P,FOURNISSEURS F
3 WHERE C.CODE_CATEGORIE = P.CODE_CATEGORIE
4 AND P.NO_FOURNISSEUR = F.NO_FOURNISSEUR
5 AND UNITES_STOCK <> 0 GROUP BY NOM_CATEGORIE, SOCIETE;
```

La requête précédente affiche les fournisseurs, les catégories des produits et la somme des unités en stock pour les produits qui ont un stock ; la deuxième requête est la traduction dans l'ancienne syntaxe.



```
SQL> SELECT P.NOM_PRODUIT, SUM( D.QUANTITE) QUANTITE,
2 AVG(D.PRIX_UNITAIRE) PRIX_UNITAIRE
3 FROM PRODUITS P JOIN DETAILS_COMMANDES D
4 ON P.REF_PRODUIT = D.REF_PRODUIT
5 WHERE P.CODE_CATEGORIE = 1 GROUP BY P.NOM_PRODUIT;
```

NOM_PRODUIT	QUANTITE	PRIX_UNITAIRE
Beer	446119	62,76
Coffee	413199	47,4
Chai	416105	72,6
Ipoh Coffee	450562	88,32
Tea	449445	64,56
Sasquatch Ale	451057	37,08
Chartreuse verte	435834	76,8
Côte de Blaye	465860	64,92
Lakkalikööri	396703	63,72
Chang	455970	51,48
Guaraná Fantástica	428101	79,08
Rhönbräu Klosterbier	410993	42,48
Steeleye Stout	422337	41,52
Laughing Lumberjack Lager	432044	78,48
Outback Lager	489108	47,76
Green Tea	448015	42,6

Dans l'exemple précédent, la requête joint les tables DETAILS\_COMMANDE et PRODUITS à l'aide de l'opérateur « **JOIN ON** » affichant uniquement les produits pour la catégorie 1 et la somme des quantités vendues et la moyenne du prix unitaire.



```
SQL> SELECT C.SOCIETE CLIENT, F.SOCIETE FOURNISSEUR, C.VILLE
2 FROM CLIENTS C, FOURNISSEURS F ON C.VILLE = F.VILLE;
```

CLIENT	FOURNISSEUR	VILLE
Consolidated Holdings	Exotic Liquids	London
Eastern Connection	Exotic Liquids	London
Familia Arquibaldo	Refrescos Americanas LTDA	São Paulo
Lehmans Marktstand	Plutzer Lebensmittelgroßmärkte AG	Frankfurt
Mère Paillarde	Ma Maison	Montréal
North/South	Exotic Liquids	London
Paris spécialités	Aux joyeux ecclésiastiques	Paris
Queen Cozinha	Refrescos Americanas LTDA	São Paulo
Seven Seas Imports	Exotic Liquids	London

Spécialités du monde	Aux joyeux ecclésiastiques	Paris
Tradição Hipermercados	Refrescos Americanas LTDA	São Paulo
Alfreds Futterkiste	Heli Süßwaren GmbH Co. KG	Berlin
Around the Horn	Exotic Liquids	London
B's Beverages	Exotic Liquids	London
Comércio Mineiro	Refrescos Americanas LTDA	São Paulo

L'opérateur « **JOIN ON** » effectue la jointure entre deux tables en se servant des conditions spécifiées respectant la syntaxe suivante :

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
FROM NOM_TABLE1[ JOIN NOM_TABLE2 ON
    (NOM_TABLE1.NOM_COLONNE = NOM_TABLE2.NOM_COLONNE)
    [{AND | OR} EXPRESSION ]] ;
```



```
SQL> SELECT NOM||' '||PRENOM "Vendeur", SOCIETE "Client",
2 TO_CHAR( DATE_COMMANDE,'FMDD Mon YYYY') "Commande", PORT "Port"
3 FROM CLIENTS A JOIN COMMANDES B ON A.CODE_CLIENT = B.CODE_CLIENT
4 JOIN EMPLOYES C ON B.NO_EMPLOYE = C.NO_EMPLOYE
5 AND DATE_COMMANDE > '29/06/2011' AND PORT BETWEEN 70 AND 80;
```

Vendeur	Client	Commande	Port
Marielle Michel	QUICK-Stop	30 Juin 2011	74.8
Thimoleon Georges	Blauer See Delikatessen	30 Juin 2011	79.1
Cleret Doris	Océano Atlántico Ltda.	30 Juin 2011	79.8
Capharsie Gérard	GROSELLA-Restaurante	30 Juin 2011	75.6

Vous avez pu remarquer que les parenthèses qui suivent le mot clé « **ON** » sont facultatives, mais elles permettent une meilleure lisibilité du code si vous prenez soin de les positionner.



```
SQL> SELECT DC.REF_PRODUIT, COUNT( DISTINCT NO_COMMANDE) COM
2 FROM DETAILS_COMMANDES DC JOIN PRODUITS PR
3 ON ( DC.PRIX_UNITAIRE BETWEEN PR.PRIX_UNITAIRE*0.95
4 AND PR.PRIX_UNITAIRE*1.05 AND PR.REF_PRODUIT = 57 )
5 JOIN COMMANDES CO
6 ON ( CO.NO_COMMANDE = DC.NO_COMMANDE AND CO.ANNEE = 2011 )
7 JOIN CLIENTS CL
8 ON ( CO.CODE_CLIENT = CL.CODE_CLIENT AND CL.PAYS LIKE 'A%')
9 GROUP BY DC.REF_PRODUIT;
```

REF_PRODUIT	COM
20	276
62	259
96	292
111	228
120	274

## L'opérateur JOIN USING

L'opérateur « **JOIN USING** » effectue la jointure entre deux tables en se servant des colonnes homonymes dans les deux tables spécifiées respectant la syntaxe suivante :

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
```



```
FROM NOM_TABLE1[ JOIN NOM_TABLE2 USING (NOM_COLONNE1[,...])] ;
SQL> SELECT C.SOCIETE CLIENT, F.SOCIETE FOURNISSEUR, VILLE
2 FROM CLIENTS C JOIN FOURNISSEURS F USING(VILLE);
```

CLIENT	FOURNISSEUR	VILLE
Consolidated Holdings	Exotic Liquids	London
Eastern Connection	Exotic Liquids	London
Familia Arquibaldo	Refrescos Americanas LTDA	São Paulo
Lehmanns Marktstand	Plutzer Lebensmittelgroßmärkte AG	Frankfurt
Mère Paillard	Ma Maison	Montréal
North/South	Exotic Liquids	London
Paris spécialités	Aux joyeux ecclésiastiques	Paris
Queen Cozinha	Refrescos Americanas LTDA	São Paulo
Seven Seas Imports	Exotic Liquids	London
Spécialités du monde	Aux joyeux ecclésiastiques	Paris
Tradição Hipermercados	Refrescos Americanas LTDA	São Paulo
Alfreds Futterkiste	Heli Süßwaren GmbH Co. KG	Berlin
Around the Horn	Exotic Liquids	London
B's Beverages	Exotic Liquids	London
Comércio Mineiro	Refrescos Americanas LTDA	São Paulo

La requête précédente affiche les clients qui sont localisés dans une ville d'un fournisseur.

### Attention

Il faut faire attention toutes les champs homonymes qui ont été utilisés dans l'opérateur « **USING** » ne peuvent plus être préfixés par le nom d'une des deux tables.



```
SQL> SELECT PRODUITS.REF_PRODUIT FROM DETAILS_COMMANDES
2 JOIN PRODUITS USING ( REF_PRODUIT );
SELECT PRODUITS.REF_PRODUIT
*
ERREUR à la ligne 1 :
ORA-25154: la partie colonne de la clause USING ne peut pas avoir de
qualificatif
```

La requête suivante affiche la somme des ventes par pays du produit 57 dans l'année 2011 uniquement pour les pays qui commencent par une lettre de A÷C. Il convient de remarquer l'utilisation des filtres à l'aide de la clause « **WHERE** ».



```
SQL> SELECT CL.PAYS, SUM( DC.PRIX_UNITAIRE*DC.QUANTITE) CA
2 FROM DETAILS_COMMANDES DC JOIN PRODUITS PR USING ( REF_PRODUIT )
3 JOIN COMMANDES CO USING ( NO_COMMANDE )
4 JOIN CLIENTS CL USING ( CODE_CLIENT )
5 WHERE REF_PRODUIT=57 AND CO.ANNEE=2011 AND CL.PAYS BETWEEN 'A' AND 'D'
6 GROUP BY CL.PAYS;
```

PAYS	CA
Allemagne	1199077,44
Argentine	288452,4
Autriche	196615,92
Canada	387313,2
Brésil	1030623,84
Belgique	240843,12

La requête suivante affiche la somme des ventes par pays, par ville et par année pour les produits vendus dans la même ville ou ils ont été produits. L'utilisation du champ homonyme VILLE dans

l'opérateur « **USING** » est possible car seule la table **CLIENTS** et la table **FOURNISSEURS** possèdent ce champ. Sinon il est impératif d'utiliser l'opérateur « **ON** » en préfixant le nom du champ avec le nom de la table ou son alias.



```
SQL> SELECT CL.PAYS, VILLE, CO.ANNEE, SUM( DC.PRIX_UNITAIRE*DC.QUANTITE) CA
2 FROM DETAILS_COMMANDES DC JOIN PRODUITS PR USING ( REF_PRODUIT )
3 JOIN COMMANDES CO USING ( NO_COMMANDE )
4 JOIN CLIENTS CL USING ( CODE_CLIENT )
5 JOIN FOURNISSEURS FO USING ( NO_FOURNISSEUR, VILLE)
6 GROUP BY CL.PAYS, VILLE, CO.ANNEE;
```

PAYS	VILLE	ANNEE	CA
France	Paris	2011	1200918,72
Royaume-Uni	London	2011	2009131,32
Allemagne	Berlin	2010	613423,92
Royaume-Uni	London	2010	3062000,28
Brésil	São Paulo	2010	1005976,68
Allemagne	Frankfurt a.M.	2011	753836,52
Canada	Montréal	2011	596944,32
France	Paris	2010	1884362,4
Canada	Montréal	2010	508980
Allemagne	Frankfurt a.M.	2010	801738,12
Allemagne	Berlin	2011	317073
Brésil	São Paulo	2011	525328,44

## L'opérateur NATURAL JOIN

L'opérateur « **NATURAL JOIN** » effectue la jointure entre deux tables en se servant de toutes les colonnes des deux tables qui portent le même nom avec la syntaxe suivante :

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
FROM NOM_TABLE1 NATURAL JOIN NOM_TABLE2 ;
```



```
SQL> SELECT PAYS, ANNEE, SUM( PRIX_UNITAIRE*QUANTITE) CA
2 FROM CLIENTS NATURAL JOIN COMMANDES NATURAL JOIN DETAILS_COMMANDES
3 GROUP BY PAYS, ANNEE;
```

PAYS	ANNEE	CA
États-Unis	2010	295589486
Allemagne	2010	248741421
Canada	2010	60489582,5
Belgique	2011	30031948,4
Venezuela	2011	65413824,7
Belgique	2010	54458225,8
Autriche	2010	43122999,8
Danemark	2010	41867956,9
Norvège	2010	16362449,2
Royaume-Uni	2011	112585949
...		

La requête précédente affiche, pour chaque produit, le nom de la catégorie correspondante.



## Attention

L'opérateur « **NATURAL JOIN** » réalise la jointure entre deux tables en utilisant des noms de colonnes identiques, et non pas l'intermédiaire de l'intégrité référentielle, à savoir les clés primaires et les clés étrangères.

```
SQL> SELECT PAYS, ANNEE, SUM( PRIX_UNITAIRE*QUANTITE) CA
  2   FROM CLIENTS NATURAL JOIN COMMANDES NATURAL JOIN DETAILS_COMMANDES
  3   NATURAL JOIN PRODUITS GROUP BY PAYS, ANNEE;
SELECT PAYS, ANNEE,
*
ERREUR à la ligne 1 :
ORA-01722: Nombre non valide
```

Dans l'exemple précédent, la requête joint les tables DETAILS\_COMMANDE et PRODUITS à l'aide de l'opérateur « **NATURAL JOIN** ». Vous pouvez remarquer que la jointure donne un message d'erreur de nombre invalide. En effet la colonne QUANTITE est prise en compte pour la jointure ; son nom est identique dans les deux tables, mais le type de la colonne est différent.

## L'auto-jointure

L'**auto-jointure** met en corrélation les lignes d'une table avec d'autres lignes de la même table. Elle permet donc de ramener sur la même ligne de résultat des informations venant d'une ligne plus des informations venant d'une autre ligne de la même table.

La jointure d'une table à elle-même n'est possible qu'à condition d'utiliser des "alias" ou abréviations de table pour faire référence à une même table sous des noms différents.

L'utilisation d'un alias (ou nom d'emprunt ou synonyme) permet de renommer une des tables et évite les problèmes d'ambiguïté pour les noms de colonnes qui doivent être préfixés par le synonyme des différentes tables.



```
SQL> SELECT A.NOM "Employé", B.NOM "Supérieur" FROM EMPLOYES A,EMPLOYES B
  2   WHERE A.REND_COMPTE = B.NO_EMPLOYE ORDER BY "Employé";
```

Employé	Supérieur
Alvarez	Chambaud
Arrambide	Chambaud
Aubert	Belin
Aubry	Leger
Barre	Leger
Bazart	Ragon
Belin	Brasseur
Berlioz	Leger
Besse	Ragon
...	

La requête comporte une jointure externe pour pouvoir afficher tous les employés, y compris ceux qui n'ont pas de supérieur hiérarchique.



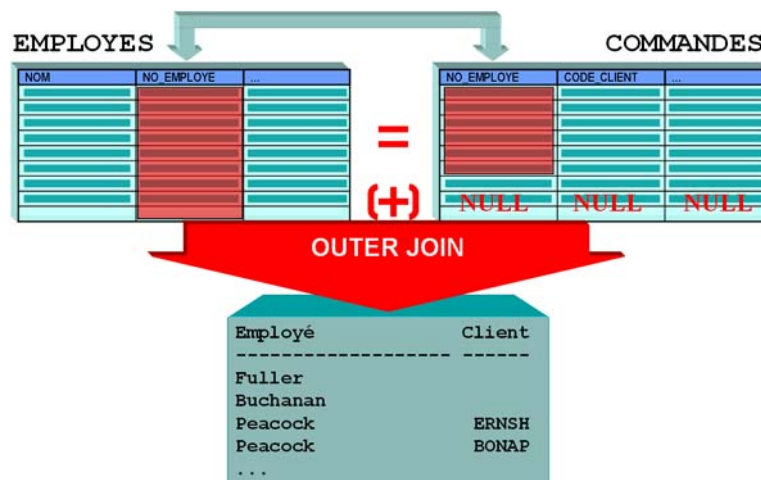
```
SQL> SELECT A.NOM "Employé", B.NOM "Supérieur"
  3   FROM EMPLOYES A JOIN EMPLOYES B ON ( A.REND_COMPTE = B.NO_EMPLOYE);
```

Employé	Supérieur
Dupouey	Ragon
Kessler	Ragon

Bazart	Ragon
Canu	Ragon
Rivat	Ragon
Gregoire	Ragon
Michiels	Ragon
...	

## La jointure externe

Dans le cas d'une jointure classique, lorsqu'une ligne d'une table ne satisfait pas à la condition de jointure, cette ligne n'apparaît pas dans le résultat final.



Il peut cependant être souhaitable de conserver les lignes d'une table qui ne répondent pas à la condition de jointure. On parle alors de **jointure externe** (outer join).

```
SQL> SELECT NOM "Employé", ANNEE, SUM( PORT) FROM EMPLOYES A, COMMANDES B
2 WHERE A.NO_EMPLOYE = B.NO_EMPLOYE (+) GROUP BY NOM, ANNEE;
```

Employé	ANNEE	SUM( PORT)
...		
Burst	2011	2886,9
Seigne	2011	1079,6
Idesheim	2011	1316,2
Kremser	2011	5805,3
Petit	2011	1714,2
Michiels	2011	795,2
Zonca	2011	875,1
Fuller		
Buchanan		
Rollet	2010	8263,9
Duriez	2010	10990,7
...		

Dans l'exemple précédent la jointure externe consiste à ajouter au résultat de la jointure normale, l'ensemble des employés qui n'ont pas effectué de ventes. Ce résultat est obtenu en ajoutant le symbole « + » juste après **COMMANDES.NO\_EMPLOYE**.

Le signe « + » permet d'ajouter une ligne fictive de valeur nulle, pour chaque enregistrement de la première table qui n'a pas de correspondance dans la deuxième.



```
SQL> SELECT NOM "Employé", SOCIETE "Client"
  2 FROM EMPLOYES A,COMMANDES B,CLIENTS C
  3 WHERE A.NO_EMPLOYE = B.NO_EMPLOYE (+)
  4       AND B.CODE_CLIENT = C.CODE_CLIENT
  5       AND DATE_COMMANDE >= '30/06/2011'
  6       AND A.NO_EMPLOYE BETWEEN 6 AND 20;
```

Employé	Client
Cleret	Océano Atlántico Ltda.
Poidatz	Queen Cozinha
Capharsie	GROSELLA-Restaurante
Perny	Centro comercial Moctezuma
Gerard	Antonio Moreno Taquería

La requête précédente retourne uniquement les employés qui ont effectué des ventes, malgré la jointure externe entre la table EMPLOYES et COMMANDES. En effet la condition B.CODE\_CLIENT = C.CODE\_CLIENT impose l'existence d'un enregistrement dans la table B (COMMANDE). Pour pouvoir visualiser l'ensemble des employés et pour ceux qui ont effectués des ventes les clients correspondants, il faut transformer la jointure entre les tables COMMANDES et CLIENTS en jointure externe.



```
SQL> SELECT NOM "Employé", SOCIETE "Client"
  2 FROM EMPLOYES A,COMMANDES B,CLIENTS C
  3 WHERE A.NO_EMPLOYE = B.NO_EMPLOYE (+)
  4       AND B.CODE_CLIENT = C.CODE_CLIENT(+)
  5       AND DATE_COMMANDE >= '30/06/2011'
  6       AND A.NO_EMPLOYE BETWEEN 6 AND 20;
```

Employé	Client
Cleret	Océano Atlántico Ltda.
Perny	Centro comercial Moctezuma
Poidatz	Queen Cozinha
Capharsie	GROSELLA-Restaurante
Gerard	Antonio Moreno Taquería

Toutefois la requête précédente ne retourne toujours pas les employés qui n'ont pas passé de commande, car la condition DATE\_COMMANDE >= '30/06/2011' impose également l'existence d'un enregistrement dans la table B (COMMANDE).



```
SQL> SELECT NOM "Employé", NVL(SOCIETE, '- Pas de client-') "Client"
  2 FROM EMPLOYES A,COMMANDES B,CLIENTS C
  3 WHERE A.NO_EMPLOYE = B.NO_EMPLOYE (+)
  4       AND B.CODE_CLIENT = C.CODE_CLIENT(+)
  5       AND NVL(DATE_COMMANDE, '30/06/2011') >= '30/06/2011'
  6       AND A.NO_EMPLOYE BETWEEN 6 AND 20;
```

Employé	Client
Cleret	Océano Atlántico Ltda.
Perny	Centro comercial Moctezuma
Poidatz	Queen Cozinha
Capharsie	GROSELLA-Restaurante
Gerard	Antonio Moreno Taquería
Brasseur	- Pas de client-
Fuller	- Pas de client-
Belin	- Pas de client-



Pour la requête précédente on veut afficher uniquement les clients qui ont dans leur nom la lettre A.



```
SQL> SELECT NOM "Employé", NVL(SOCIETE, '- Pas de client-') "Client"
2 FROM EMPLOYES A,COMMANDES B,CLIENTS C
3 WHERE A.NO_EMPLOYE = B.NO_EMPLOYE (+)
4 AND B.CODE_CLIENT = C.CODE_CLIENT(+)
5 AND NVL(Date_COMMANDE,'30/06/2011') >= '30/06/2011'
6 AND A.NO_EMPLOYE BETWEEN 6 AND 20 AND SOCIETE LIKE '%A%' ;
```

Employé	Client
Cleret	Océano Atlántico Ltda.
Capharsie	GROSELLA-Restaurante
Gerard	Antonio Moreno Taquería

### Note



La notation « (+) » doit accompagner toute colonne qui est utilisée dans la clause « **WHERE** » et appartient à une table qui n'a pas de correspondances pour tous les enregistrements dans la table maître.

La notation qui détermine une jointure externe « (+) » peut être positionnée à droite de l'égalité ou à gauche. Ainsi les deux syntaxes suivantes sont équivalentes :

**A.NOM = B.NOM (+) ou A.NOM (+) = B.NOM**



```
SQL> SELECT NOM "Employé", NVL(SOCIETE, '- Pas de client-') "Client"
2 FROM EMPLOYES A,COMMANDES B,CLIENTS C
3 WHERE A.NO_EMPLOYE = B.NO_EMPLOYE (+)
4 AND B.CODE_CLIENT = C.CODE_CLIENT(+)
5 AND NVL(Date_COMMANDE,'30/06/2011') >= '30/06/2011'
6 AND A.NO_EMPLOYE BETWEEN 6 AND 20 AND SOCIETE (+) LIKE '%A%' ;
```

Employé	Client
Capharsie	GROSELLA-Restaurante
Cleret	Océano Atlántico Ltda.
Gerard	Antonio Moreno Taquería
Belin	- Pas de client-
Fuller	- Pas de client-
Brasseur	- Pas de client-
Poidatz	- Pas de client-
Perny	- Pas de client-

## L'opérateur OUTER JOIN

L'opérateur « **OUTER JOIN ON** » effectue une jointure externe entre deux tables en se servant des conditions spécifiées respectant la syntaxe suivante :

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
FROM NOM_TABLE1[ {LEFT | RIGHT | FULL} OUTER JOIN NOM_TABLE2
{ ON (NOM_TABLE1.NOM_COLONNE = NOM_TABLE2.NOM_COLONNE) ]
| USING (NOM_COLONNE1[,...]) ]};
```

**LEFT | RIGHT**

Indique que la table de gauche/droite est dominante, celle dont on affiche tous les enregistrements.

**FULL**

Cette option est l'union des deux requêtes,  
« **LEFT OUTER JOIN** » et « **RIGHT OUTER JOIN** ».



```
SQL> SELECT A.NOM "Employé",
2         NVL(B.NOM, '-----') "Supérieur",
3         NVL(C.NOM, '-----') "Manager"
4 FROM EMPLOYES A LEFT OUTER JOIN EMPLOYES B
5      ON (A.REND_COMPTE = B.NO_EMPLOYE)
6 LEFT OUTER JOIN EMPLOYES C
7      ON (B.REND_COMPTE = C.NO_EMPLOYE)
8      AND A.NO_EMPLOYE BETWEEN 6 AND 20;
```

Employé	Supérieur	Manager
-----	-----	-----
...		
Malejac	Chambaud	Brasseur
Suyama	Chambaud	Brasseur
Letertre	Chambaud	-----
Lombard	Chambaud	-----
Arrambide	Chambaud	-----
Dohr	Chambaud	-----
Mangeard	Chambaud	-----
Guerdon	-----	-----
Grangirard	-----	-----
Etienne	-----	-----
Giroux	-----	-----
Callahan	-----	-----
Maurer	-----	-----
...		

La requête précédente affiche tous les employés et leur supérieur hiérarchique s'il y a un et le manager de celui-ci si toutefois il existe.

La requête suivante affiche tous les clients français et les fournisseurs qui habitent dans la même ville s'ils existent.



```
SQL> SELECT NVL(C.SOCIETE, '-----') CLIENT,
2         NVL(F.SOCIETE, '-----') FOURNISSEUR
3 FROM CLIENTS C LEFT OUTER JOIN FOURNISSEURS F
4      USING( VILLE,PAYS) WHERE PAYS = 'France';
```

CLIENT	FOURNISSEUR
-----	-----
Spécialités du monde	Aux joyeux ecclésiastiques
Paris spécialités	Aux joyeux ecclésiastiques
Bon app'	-----
Blondel père et fils	-----
La corne d'abondance	-----
Vins et alcools Chevalier	-----
Folies gourmandes	-----
Victuailles en stock	-----
France restauration	-----
Du monde entier	-----
La maison d'Asie	-----

La requête suivante affiche tous les fournisseurs français et les clients qui habitent dans la même ville s'ils existent.

```
SQL> SELECT NVL(C.SOCIETE, '-----') CLIENT,
2         NVL(F.SOCIETE, '-----') FOURNISSEUR
```



```

3 FROM CLIENTS C RIGHT OUTER JOIN FOURNISSEURS F
4 USING( VILLE,PAYS) WHERE PAYS = 'France';

```

CLIENT	FOURNISSEUR
-----	-----
Paris spécialités	Aux joyeux ecclésiastiques
Spécialités du monde	Aux joyeux ecclésiastiques
-----	Gai pâturage
-----	Escargots Nouveaux

La requête suivante affiche tous les fournisseurs français et les clients qui habitent dans la même ville s'ils existent.

```

SQL> SELECT NVL(C.SOCIETE,'-----') CLIENT,
2         NVL(F.SOCIETE,'-----') FOURNISSEUR
3 FROM CLIENTS C FULL OUTER JOIN FOURNISSEURS F
4 USING( VILLE,PAYS) WHERE PAYS = 'France';

```

CLIENT	FOURNISSEUR
-----	-----
Du monde entier	-----
Folies gourmandes	-----
France restauration	-----
La corne d'abondance	-----
La maison d'Asie	-----
Paris spécialités	Aux joyeux ecclésiastiques
Spécialités du monde	Aux joyeux ecclésiastiques
Victuailles en stock	-----
Vins et alcools Chevalier	-----
Blondel père et fils	-----
Bon app'	-----
-----	Escargots Nouveaux
-----	Gai pâturage

Le filtre sur la colonne PAYS est simple de traiter car il est utilisé dans la clause « **USING** ». Si vous utiliser la jointure en précisant la clause « **ON** » il peut y avoir des problèmes car on ne parle plus d'une seule colonne mais des deux.

```

SQL> SELECT NVL(C.SOCIETE,'-----') CLIENT,
2         NVL(F.SOCIETE,'-----') FOURNISSEUR
3 FROM CLIENTS C LEFT OUTER JOIN FOURNISSEURS F
4 ON (C. VILLE = F.VILLE AND C.PAYS = F.PAYS) WHERE C.PAYS = 'France';

```

CLIENT	FOURNISSEUR
-----	-----
Spécialités du monde	Aux joyeux ecclésiastiques
Paris spécialités	Aux joyeux ecclésiastiques
Bon app'	-----
Blondel père et fils	-----
La corne d'abondance	-----
Vins et alcools Chevalier	-----
Folies gourmandes	-----
Victuailles en stock	-----
France restauration	-----
Du monde entier	-----
La maison d'Asie	-----

```

SQL> SELECT NVL(C.SOCIETE,'-----') CLIENT,

```

```

2      NVL(F.SOCIETE,'-----') Fournisseur
3  FROM CLIENTS C LEFT OUTER JOIN FOURNISSEURS F
4  ON (C.VILLE = F.VILLE AND C.PAYS = F.PAYS) WHERE F.PAYS = 'France';

```

CLIENT	FOURNISSEUR
-----	-----
Paris spécialités	Aux joyeux ecclésiastiques
Spécialités du monde	Aux joyeux ecclésiastiques

Dans le premier exemple le filtre est sur une colonne de la table maître. Comme pour cette table on récupère tous les enregistrements le filtre ne perturbe pas la jointure. Dans le deuxième cas la le filtre transforme la jointure externe dans une équijointure. Le filtre ne doit pas être écrit dans la clause « **WHERE** » mais inclus directement dans la déclaration de la jointure.



```

SQL> SELECT NVL(C.SOCIETE,'-----') CLIENT,
2      NVL(F.SOCIETE,'-----') Fournisseur
3  FROM CLIENTS C LEFT OUTER JOIN FOURNISSEURS F
4  ON (C.VILLE = F.VILLE AND C.PAYS = F.PAYS AND F.PAYS = 'France' );

```

CLIENT	FOURNISSEUR
-----	-----
Spécialités du monde	Aux joyeux ecclésiastiques
Paris spécialités	Aux joyeux ecclésiastiques
Great Lakes Food Market	-----
Königlich Essen	-----
Victuailles en stock	-----
Godos Cocina Típica	-----
La maison d'Asie	-----
Wilman Kala	-----
...	

```
SQL >SELECT COUNT(*) FROM CLIENTS;
```

```

COUNT(*)
-----
91

```

```
SQL >SELECT COUNT(*) FROM FOURNISSEURS;
```

```

COUNT(*)
-----
29

```

```

SQL >SELECT COUNT(DISTINCT C.SOCIETE) C,COUNT(DISTINCT F.SOCIETE) F,
2  COUNT(*) TOTAL FROM CLIENTS C RIGHT OUTER JOIN FOURNISSEURS F
3  ON (C.VILLE = F.VILLE );

```

C	F	TOTAL
-----	-----	-----
15	29	38

```

SQL >SELECT COUNT(DISTINCT C.SOCIETE) C,COUNT(DISTINCT F.SOCIETE) F,
2  COUNT(*) TOTAL FROM CLIENTS C LEFT OUTER JOIN FOURNISSEURS F
3  ON (C.VILLE = F.VILLE );

```

C	F	TOTAL
-----	-----	-----
91	6	91

```

SQL >SELECT COUNT(DISTINCT C.SOCIETE) C,COUNT(DISTINCT F.SOCIETE) F,
2  COUNT(*) TOTAL FROM CLIENTS C FULL OUTER JOIN FOURNISSEURS F
3  ON (C.VILLE = F.VILLE );

```

C	F	TOTAL
91	29	114

```

SQL >SELECT COUNT(DISTINCT C.SOCIETE) C,COUNT(DISTINCT F.SOCIETE) F,
2 COUNT(*) TOTAL FROM CLIENTS C RIGHT OUTER JOIN FOURNISSEURS F
3 ON (C. VILLE = F.VILLE AND F.PAYS = 'France');

```

C	F	TOTAL
2	29	30

```

SQL >SELECT COUNT(DISTINCT C.SOCIETE) C,COUNT(DISTINCT F.SOCIETE) F,
2 COUNT(*) TOTAL FROM CLIENTS C LEFT OUTER JOIN FOURNISSEURS F
3 ON (C. VILLE = F.VILLE AND F.PAYS = 'France');

```

C	F	TOTAL
91	1	91

```

SQL >SELECT COUNT(DISTINCT C.SOCIETE) C,COUNT(DISTINCT F.SOCIETE) F,
2 COUNT(*) TOTAL FROM CLIENTS C FULL OUTER JOIN FOURNISSEURS F
3 ON (C. VILLE = F.VILLE AND F.PAYS = 'France');

```

C	F	TOTAL
91	29	119

- *Opérateurs ensemblistes*
- *Sous-requêtes*
- *IN et EXIST*
- *ALL et ANY*
- *WITH*

# 8

## Les jointures complexes



### Objectifs

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- Effectuer des interrogations avec les opérateurs ensemblistes.
- Sélectionner des lignes en utilisant les sous-requêtes.
- Effectuer des interrogations avec les opérateurs IN, ALL et ANY.
- Sélectionner des lignes en utilisant les sous-requêtes dans la clause FROM.
- Sélectionner des lignes en utilisant les sous-requêtes synchronisées.



### Contenu

Les opérateurs ensemblistes	8-2	Sous-requête renvoyant un tableau	8-13
Les sous-requêtes	8-7		
Sous-requête monolignes	8-7	Sous-requête synchronisée	8-14
Sous-requête multilignes	8-10	Sous-requête dans la clause FROM	8-17
Les opérateurs ANY et ALL	8-11	La clause WITH	8-19

## Les opérateurs ensemblistes

---

Il est parfois nécessaire de combiner des informations de même type à partir de plusieurs tables. Un exemple classique est la fusion de plusieurs listes de mailing en vue d'un envoi en masse de publicité. Les conditions d'envoi suivantes doivent généralement pouvoir être spécifiées :

- à toutes les personnes dans les deux listes (en évitant d'envoyer la lettre deux fois à une même personne) ;
- seulement aux personnes qui se trouvent dans les deux listes ;
- seulement aux personnes qui se trouvent dans une des deux listes.

Dans Oracle, ces trois conditions sont définies à l'aide des opérateurs :

- « **UNION** »
- « **INTERSECT** »
- « **MINUS** »

La syntaxe de l'instruction « **SELECT** » :

```
SELECT {*,[EXPRESSION1 [AS] ALIAS1[,...]]} FROM NOM_TABLE
WHERE PREDICAT
GROUP BY [NOM_COLONNE1|EXPRESSION1][,...]
HAVING PREDICAT
```

OPERATEUR [ALL|DISTINCT]

```
SELECT {*,[EXPRESSION1 [AS] ALIAS1[,...]]} FROM NOM_TABLE
WHERE PREDICAT
GROUP BY [NOM_COLONNE1|EXPRESSION1][,...]
HAVING PREDICAT
```

ORDER BY [POSITION1] [ASC|DESC][,...] ;

Dans une requête utilisant des opérateurs ensemblistes :

- Tous les ordres « **SELECT** » doivent avoir le même nombre de colonnes sélectionnées, et leurs types doivent être compatibles. Les conversions éventuelles doivent être faites à l'intérieur de l'ordre « **SELECT** » à l'aide des fonctions de conversion « **TO\_CHAR** », « **TO\_DATE** », etc.
- Aucun attribut ne peut être de type « **LONG** », « **BLOB** », « **CLOB** », « **BFILE** ».
- Les doublons sont éliminés, « **DISTINCT** » est implicite.
- Les noms des colonnes ou alias sont ceux du premier ordre « **SELECT** ».
- La largeur de chaque colonne est donnée par la plus grande de tous ordres « **SELECT** » confondus.

- Si une clause « **ORDER BY** » est utilisée, elle doit faire référence au numéro de la colonne et non à son nom, car le nom peut être différent dans chacun des ordres « **SELECT** ».

## L'opérateur UNION

L'opérateur d'union « **UNION** » entre deux requêtes permet de retrouver l'ensemble des lignes des deux requêtes de départ. Les attributs de même rang des requêtes de départ doivent être compatibles, c'est-à-dire définis de même type.



```
SQL> SELECT SOCIETE,VILLE,'Client' FROM CLIENTS
```

```
2 UNION
```

```
3 SELECT SOCIETE,VILLE,'Fournisseur' FROM FOURNISSEURS;
```

SOCIETE	VILLE	'CLIENT'
Alfreds Futterkiste	Berlin	Client
Ana Trujillo Emparedados y helad	México D.F.	Client
Antonio Moreno Taquería	México D.F.	Client
Around the Horn	London	Client
Aux joyeux ecclésiastiques	Paris	Fournisseur
B's Beverages	London	Client
Berglunds snabbköp	Luleå	Client
Bigfoot Breweries	Bend	Fournisseur
Blauer See Delikatessen	Mannheim	Client
Blondel père et fils	Strasbourg	Client
Bon app'	Marseille	Client
...		

Dans l'exemple précédent, la requête affiche l'ensemble des tiers de l'entreprise, aussi bien des clients que des fournisseurs.

### Note



Les noms des colonnes sont ceux de la première requête ainsi que les alias qui sont utilisées dans les ordres de tri.

Il faut se rappeler que la clause « **ORDER BY** » ne peut figurer qu'une fois en fin du bloque SQL, car elle opère sur le résultat concaténé des différents « **SELECT** ».



```
SQL> SELECT SOCIETE,VILLE,'Client' "Cli/Four" FROM CLIENTS
```

```
2 UNION
```

```
3 SELECT SOCIETE,VILLE,'Fournisseur' FROM FOURNISSEURS
```

```
4 ORDER BY VILLE, SOCIETE;
```

SOCIETE	VILLE	Cli/Four
Drachenblut Delikatessen	Aachen	Client
Rattlesnake Canyon Grocery	Albuquerque	Client
Old World Delicatessen	Anchorage	Client
Grandma Kelly's Homestead	Ann Arbor	Fournisseur
Gai pâturage	Annecy	Fournisseur
Vaffeljernet	Århus	Client
Galería del gastrónomo	Barcelona	Client
LILA-Supermercado	Barquisimeto	Client
Bigfoot Breweries	Bend	Fournisseur
Magazzini Alimentari Riuniti	Bergamo	Client
Alfreds Futterkiste	Berlin	Client
...		





## Attention

L'opérateur d'union « **UNION** » entre deux requêtes permet de concaténer tous les types de données sans aucun contrôle de la pertinence de cet assemblage.

En d'autres termes on peut mélanger 'les choux' et 'les carottes' ; les informations sont affichées ensemble sans aucun contrôle.



```
SQL> SELECT SOCIETE,VILLE,'Client' "Cli/Four" FROM CLIENTS
2 UNION
3 SELECT NOM,FONCTION,'Employé' FROM EMPLOYES
4 UNION
5 SELECT NOM_PRODUIT,QUANTITE,'Produit' FROM PRODUITS;
```

SOCIETE	VILLE	Cli/Fou
Alfreds Futterkiste	Berlin	Client
Alice Mutton	20 boîtes (1 kg)	Produit
Ana Trujillo Emparedados y helad	México D.F.	Client
Aniseed Syrup	12 bouteilles (550	Produit
Antonio Moreno Taquería	México D.F.	Client
Around the Horn	London	Client
B's Beverages	London	Client
Berglunds snabbköp	Luleå	Client
Blauer See Delikatessen	Mannheim	Client
Blondel père et fils	Strasbourg	Client
Bon app'	Marseille	Client
Boston Crab Meat	24 boîtes (4 onces	Produit
Bottom-Dollar Markets	Tsawassen	Client
Buchanan	Chef des ventes	Employé
Bólido Comidas preparadas	Madrid	Client
Cactus Comidas para llevar	Buenos Aires	Client
Callahan	Assistante commerc	Employé
...		



## Attention

L'opérateur « **UNION** » comporte, comme l'ordre « **SELECT**, » la possibilité d'utiliser les options « **ALL** » ou « **DISTINCT** ».

Dans le cas de l'option « **DISTINCT** », l'option par défaut, les enregistrements en double sont éliminés ; c'est pour éliminer les doublons qu'Oracle effectue un tri des enregistrements.

Comme l'opérateur « **UNION** » c'est en effet « **UNION DISTINCT** » il est préférable d'utiliser « **UNION ALL** » chaque fois qu'il n'est pas nécessaire d'éliminer les doublons.

## L'opérateur INTERSECT

L'opérateur d'intersection entre deux requêtes permet de retrouver le résultat composé des lignes qui appartiennent simultanément aux deux requêtes de départ.



```
SQL> SELECT VILLE, NOM_PRODUIT
2 FROM CLIENTS NATURAL JOIN COMMANDES
3 JOIN DETAILS_COMMANDES USING(NO_COMMANDE)
4 JOIN PRODUITS USING(REF_PRODUIT)
5 INTERSECT
6 SELECT VILLE, NOM_PRODUIT
7 FROM PRODUITS NATURAL JOIN FOURNISSEURS;
```

VILLE	NOM_PRODUIT
-----	-----
Berlin	Granola
Berlin	Gumbär Gummibärchen
Berlin	NuNuCa Nuß-Nougat-Creme
Berlin	Schoggi Schokolade
Frankfurt a.M.	Original Frankfurter grüne Soße
Frankfurt a.M.	Rhönbräu Klosterbier
Frankfurt a.M.	Rössle Sauerkraut
Frankfurt a.M.	Thüringer Rostbratwurst
Frankfurt a.M.	Wimmers gute Semmelknödel
London	Aniseed Syrup
London	Chai
London	Chang
Montréal	Pears
Montréal	Pâté chinois
...	

La première requête retrouve la ville de résidence des clients et les noms des produits commandés. La deuxième requête retrouve la ville de résidence des fournisseurs et les noms des tous les produits commandés. L'intersection des deux requêtes affiche les villes des clients et le nom du produit pour les produits achetés par les clients qui habitent dans la même ville que le fournisseur.

## L'opérateur DIFFERENCE

L'opérateur différence entre deux requêtes permet de retrouver le résultat composé des lignes qui appartiennent à la première requête et qui n'appartiennent pas à la deuxième requête. L'opérateur différence est le seul opérateur ensembliste non commutatif.



```
SQL> SELECT A.NO_COMMANDE,DATE_COMMANDE
2 FROM COMMANDES A,DETAILS_COMMANDES B,PRODUITS C
3 WHERE A.NO_COMMANDE = B.NO_COMMANDE AND B.REF_PRODUIT = C.REF_PRODUIT
4 AND C.CODE_CATEGORIE = 1
5 MINUS
6 SELECT A.NO_COMMANDE,DATE_COMMANDE
7 FROM COMMANDES A,DETAILS_COMMANDES B,PRODUITS C
8 WHERE A.NO_COMMANDE = B.NO_COMMANDE AND B.REF_PRODUIT = C.REF_PRODUIT
9 AND C.CODE_CATEGORIE = 2 ;
```

NO\_COMMANDE DATE\_COMMA

```
-----
216015 18/02/2010
217796 12/05/2010
217798 12/05/2010
218715 22/06/2010
218722 22/06/2010
218734 22/06/2010
218914 02/07/2010
218915 02/07/2010
218919 02/07/2010
218926 02/07/2010
220669 22/09/2010
220670 22/09/2010
222147 23/11/2010
222154 23/11/2010
222155 23/11/2010
222156 23/11/2010
```

```
222163 23/11/2010
226990 14/05/2011
226993 14/05/2011
227008 14/05/2011
227014 14/05/2011
227017 14/05/2011
227887 13/06/2011
227891 13/06/2011
227894 13/06/2011
227897 13/06/2011
227903 13/06/2011
```

Dans l'exemple précédent, la requête affiche l'ensemble des commandes comportant des produits de catégorie 1 sans comporter des produits de catégories 3.

## Combinaison de plusieurs opérateurs ensemblistes

On peut utiliser, dans une même requête, plusieurs opérateurs « **UNION** », « **INTERSECT** » ou « **MINUS** », combinés avec des opérations de projection, de sélection ou de jointure. Dans ce cas, la requête est évaluée en combinant les deux premiers ordres « **SELECT** » à partir de la gauche avec le premier opérateur ensembliste, puis en combinant le résultat avec le troisième ordre « **SELECT** », etc.

Comme dans une expression arithmétique, il est possible de modifier l'ordre d'évaluation en utilisant des parenthèses.



```
SQL> SELECT A.NO_COMMANDE,DATE_COMMANDE
2 FROM COMMANDES A,DETAILS_COMMANDES B,PRODUITS C
3 WHERE A.NO_COMMANDE = B.NO_COMMANDE AND B.REF_PRODUIT = C.REF_PRODUIT
4 AND C.CODE_CATEGORIE = 10
5 MINUS
6 SELECT A.NO_COMMANDE,DATE_COMMANDE
7 FROM COMMANDES A,DETAILS_COMMANDES B,PRODUITS C
8 WHERE A.NO_COMMANDE = B.NO_COMMANDE AND B.REF_PRODUIT = C.REF_PRODUIT
9 AND C.CODE_CATEGORIE = 9
10 INTERSECT
11 SELECT A.NO_COMMANDE,DATE_COMMANDE
12 FROM COMMANDES A,DETAILS_COMMANDES B,PRODUITS C
13 WHERE A.NO_COMMANDE = B.NO_COMMANDE AND B.REF_PRODUIT = C.REF_PRODUIT
14 AND C.REF_PRODUIT = 106;
```

```
NO_COMMANDE DATE_COMMA
-----
216916 31/03/2010
216921 31/03/2010
219330 22/07/2010
219337 22/07/2010
219346 23/07/2010
219347 23/07/2010
219359 23/07/2010
221214 14/10/2010
221217 14/10/2010
221218 14/10/2010
221219 14/10/2010
222678 18/12/2010
228328 27/06/2011
228343 27/06/2011
228345 27/06/2011
```

228348 27/06/2011

228350 27/06/2011

La requête précédente affiche toutes les commandes et la date à laquelle ces commandes ont été effectuées, pour les commandes qui contiennent des produits de la catégorie 10 et pas de produits de la catégorie 9, et qui en sus contiennent le produit numéro 106.

## Les sous-requêtes

La jointure peut aussi être exprimée d'une manière plus procédurale avec des blocs imbriqués reliés par l'opérateur « **IN** ». On dit alors que la requête, dont le résultat sert de valeur de référence dans le prédicat, est une requête imbriquée ou une sous-requête.

Il est possible d'imbriquer plusieurs requêtes, le résultat de chaque requête imbriquée servant de valeur de référence dans la condition de sélection de la requête de niveau supérieur, appelée requête principale.

Il existe en fait plusieurs types de requêtes imbriquées, suivant les valeurs retournées, la dépendance ou non de la requête principale ou l'emplacement de la sous-requête.

Une sous-requête peut être exécutée une seule fois pour toute ligne évaluée de la requête mère. Mais si la sous-requête est corrélée, elle s'exécute pour chaque ligne de la requête mère du fait que son contexte d'évaluation est susceptible de changer à chaque ligne.

### Attention

Suivant le type de résultat qu'une sous-requête offre, on peut la placer dans les différentes clauses, à l'exception des clauses « **GROUP BY** » et « **ORDER BY** ».

Par définition, une sous-requête ne peut pas contenir de clause « **ORDER BY** », car elle ne produit pas un résultat destiné à l'affichage.



## Sous-requête monolignes

Une sous-requête de ce type s'utilise lorsque la valeur de référence de la condition de sélection doit être unique.

La sous-requête est entièrement évaluée avant la requête principale. Le résultat est identique à celui obtenu en exécutant dans une première étape la sous-requête pour obtenir la valeur de référence et en utilisant cette valeur dans la seconde étape pour exécuter la requête principale.

```
SQL> SELECT NOM_PRODUIT FROM PRODUITS
2 WHERE UNITES_STOCK = (SELECT MAX(UNITES_STOCK) FROM PRODUITS) ;
```

NOM\_PRODUIT

-----  
Potato Chips

Pour que la requête s'exécute correctement, il faut que la sous-requête retourne une ligne et une seule qu'elle ne renvoie rien. Si la sous-requête renvoie plusieurs lignes, SQL génère une erreur.

```
SQL> SELECT NOM_PRODUIT FROM PRODUITS
2 WHERE UNITES_STOCK = (SELECT UNITES_STOCK FROM PRODUITS WHERE 1=2);
```

aucune ligne sélectionnée

```
SQL> SELECT NOM_PRODUIT FROM PRODUITS
```



```

2 WHERE UNITES_STOCK = (SELECT UNITES_STOCK FROM PRODUITS);
WHERE UNITES_STOCK = (SELECT UNITES_STOCK FROM PRODUITS)
                        *
ERREUR à la ligne 2 :
ORA-01427: sous-interrogation ramenant un enregistrement de plus d'une
ligne

```

Dans l'exemple suivant, la première requête affiche le nombre des produits pour la commande numéro 224975. La deuxième requête affiche tous les clients et les numéros des commandes qui ont un nombre égal ou supérieur de produits.



```

SQL> SELECT COUNT( REF_PRODUIT) FROM COMMANDES NATURAL JOIN
2 DETAILS_COMMANDES WHERE NO_COMMANDE= 224975;

COUNT(REF_PRODUIT)
-----
                    52

SQL> SELECT SOCIETE, NO_COMMANDE, COUNT( REF_PRODUIT)
2 FROM CLIENTS NATURAL JOIN COMMANDES JOIN DETAILS_COMMANDES
3 USING(NO_COMMANDE) GROUP BY SOCIETE, NO_COMMANDE
4 HAVING COUNT( REF_PRODUIT) >=( SELECT COUNT( REF_PRODUIT)
5 FROM COMMANDES NATURAL JOIN DETAILS_COMMANDES
6 WHERE NO_COMMANDE = 224975);

```

SOCIETE	NO_COMMANDE	COUNT(REF_PRODUIT)
The Big Cheese	224975	52
Hungry Owl All-Night Grocers	220267	52
Furia Bacalhau e Frutos do Mar	225524	52
Rattlesnake Canyon Grocery	228419	52
Berglunds snabbköp	216798	52
Blondel père et fils	224968	52
Rancho grande	215629	52
Laughing Bacchus Wine Cellars	224974	52
LINO-Delicatesses	224963	52
Du monde entier	216814	52
White Clover Markets	220279	52
Die Wandernde Kuh	216810	52

La requête suivante affiche la moyenne des frais de ports annuels par employé pour les enregistrements qui ont une moyenne supérieure à '96%' de la moyenne maximum par année et par employé.



```

SQL> SELECT TO_CHAR(MAX(AVG(PORT)), '99D00') MAX_PORT,
2 TO_CHAR(MAX(AVG(PORT))*0.96, '9999D00') MIN_PORT
3 FROM COMMANDES GROUP BY NO_EMPLOYE, ANNEE;

MAX_PORT MIN_PORT
-----
      80,91      77,68

SQL> SELECT ANNEE,NOM||' '||PRENOM EMPLOYE,TO_CHAR(AVG(PORT), '99D00') PORT
2 FROM EMPLOYES NATURAL JOIN COMMANDES GROUP BY ANNEE, NOM, PRENOM
3 HAVING AVG(PORT) > 0.96 *( SELECT MAX(AVG(PORT)) FROM COMMANDES
4 GROUP BY NO_EMPLOYE, ANNEE);

ANNEE EMPLOYE PORT

```

2011	Montesinos Aline	77,82
2010	Capharsie Gérard	79,42
2010	Chaussende Maurice	79,26
2011	Aubert Maria	78,01
2010	Aubry Jean-Claude	80,02
2011	Thomas Patricia	78,85
2011	Leverling Janet	77,77
2011	Capharsie Gérard	78,76
2011	Zonca Virginie	79,55
2010	Aubert Maria	80,91
2011	Herve Didier	78,01

La requête suivante retourne les enregistrements de la table EMPLOYES qui ont une région d'affectation qui commence par la région de l'employé 'Vice-Président' qui s'appelle 'Brasseur'.



```
SQL> SELECT NO_EMPLOYE, NOM || ' ' || PRENOM EMPLOYE, REGION FROM EMPLOYES
2 WHERE REGION LIKE ( SELECT REGION || '%' FROM EMPLOYES
3 WHERE FONCTION='Vice-Président' AND NOM='Brasseur') ORDER BY 1;
```

NO_EMPLOYE	EMPLOYE	REGION
1	Besse José	Europe de l'Ouest
2	Destenay Agnès	Europe du Nord
3	Letertre Sylvie	Europe du Sud
4	Kremser Arnaud	Europe centrale
5	Lamarre Eric	Europe du Nord
8	Messelier Philippe	Europe centrale
11	Belin Chantal	Europe centrale
13	Courty Jean-Louis	Europe de l'Ouest
15	Suyama Michael	Europe du Sud
16	Malejac Yannick	Europe du Sud
17	Blard Jean-Benoît	Europe centrale
18	<b>Brasseur Hervé</b>	<b>Europe</b>
...		

L'exemple suivant affiche la moyenne des produits en stock pour chaque fournisseur dans le cas où sa moyenne des produits en stock est comprise entre  $\pm 10\%$  de la moyenne des produits en stock.



```
SQL> SELECT TO_CHAR(AVG(UNITES_STOCK)* 0.9, '99D00') MIN_US,
2 TO_CHAR(AVG(UNITES_STOCK)* 1.1, '99D00') MAX_US FROM PRODUITS;
```

MIN_US	MAX_US
44,23	54,06

```
SQL> SELECT SOCIETE FOURNISSEUR, TO_CHAR(AVG(UNITES_STOCK), '99D00') AVG_US
2 FROM PRODUITS NATURAL JOIN FOURNISSEURS GROUP BY SOCIETE
3 HAVING AVG(UNITES_STOCK) BETWEEN (SELECT AVG(UNITES_STOCK)
4 FROM PRODUITS)*0.9 AND (SELECT AVG(UNITES_STOCK) FROM PRODUITS)*1.1;
```

FOURNISSEUR	AVG_US
Aux joyeux ecclésiastiques	49,20
Lyngbysild	45,00
Escargots Nouveaux	48,00
Mayumi's	49,67

Grandma Kelly's Homestead	47,00
Gai pâturage	49,00
Cooperativa de Quesos 'Las Cabras'	45,60
Formaggi Fortini s.r.l.	45,33
Pasta Buttini s.r.l.	49,25

## Sous-requête multilignes

Une sous-requête de ce type s'utilise lorsque la condition de sélection fait référence à une liste de valeurs.

La sous-requête est entièrement évaluée avant la requête principale. Le résultat est identique à celui obtenu en exécutant, dans une première étape, la sous-requête pour obtenir la liste des valeurs et en utilisant cette liste dans la seconde étape pour exécuter la requête principale.

La condition de sélection emploie alors un opérateur « **IN** » ou un opérateur simple « **=** », « **!=** », « **<>** », « **<** », « **>** », « **<=** », « **>=** » précédé de « **ALL** » ou de « **ANY** ».

### L'opérateur IN

L'opérateur « **IN** » compare une expression à une donnée quelconque d'une liste ramenée par la sous-requête. Il est équivalent d'une jointure entre les deux ensembles des données représentées par les deux requêtes.



```
SQL> SELECT SOCIETE, VILLE, PAYS FROM CLIENTS
      2 WHERE CODE_CLIENT IN( SELECT CODE_CLIENT FROM COMMANDES
      3 NATURAL JOIN EMPLOYES WHERE REND_COMPTE = 11);
```

SOCIETE	VILLE	PAYS
Frankenversand	München	Allemagne
Ottilies Käseladen	Köln	Allemagne
Chop-suey Chinese	Bern	Suisse
Blauer See Delikatessen	Mannheim	Allemagne
Lehmans Marktstand	Frankfurt a.M.	Allemagne
Morgenstern Gesundheitskost	Leipzig	Allemagne
Königlich Essen	Brandenburg	Allemagne
Ernst Handel	Graz	Autriche
Alfreds Futterkiste	Berlin	Allemagne
Drachenblut Delikatessen	Aachen	Allemagne
Toms Spezialitäten	Münster	Allemagne
Wolski Zajazd	Warszawa	Pologne
QUICK-Stop	Cunewalde	Allemagne
Die Wandernde Kuh	Stuttgart	Allemagne
Piccolo und mehr	Salzburg	Autriche
Richter Supermarkt	Genève	Suisse

Dans l'exemple précédent vous pouvez observer la liste des clients des employés qui ont le supérieur hiérarchique 11.

### Attention

La négation de l'opérateur « **IN** », à savoir « **NOT IN** », doit être utilisée avec prudence car elle retourne « **FALSE** » si une des valeurs ramenées par la sous-interrogation est « **NULL** ».

Il est préférable de s'assurer qu'aucune des valeurs retournées par la sous-requête n'est « **NULL** ».



La requête suivante affiche les employés qui ne sont supérieur hiérarchique d'aucun autre employé.



```
SQL> SELECT DISTINCT REND_COMPTE FROM EMPLOYES;
```

```
REND_COMPTE
```

```
-----
```

```
11
```

```
95
```

```
86
```

```
14
```

```
24
```

```
23
```

```
33
```

```
18
```

```
SQL> SELECT NOM||' '||PRENOM EMPLOYE, NO_EMPLOYE, REND_COMPTE
2 FROM EMPLOYES WHERE NO_EMPLOYE NOT IN
3 ( SELECT REND_COMPTE FROM EMPLOYES);
```

aucune ligne sélectionnée

```
SQL> SELECT NOM||' '||PRENOM EMPLOYE, NO_EMPLOYE, REND_COMPTE
2 FROM EMPLOYES WHERE NO_EMPLOYE NOT IN
3 ( SELECT REND_COMPTE FROM EMPLOYES
5 WHERE REND_COMPTE IS NOT NULL);
```

EMPLOYE	NO_EMPLOYE	REND_COMPTE
-----	-----	-----
Berlioz Jacques	70	95
Nocella Guy	71	86
Herve Didier	72	24
Mangeard Jocelyne	73	33
Cazade Anne-Claire	74	11
Devie Thérèse	75	
Peacock Margaret	76	95
Idesheim Annick	77	11
...		

## Les opérateurs ANY et ALL

### L'opérateur ANY

L'opérateur « **ANY** » compare une expression à chaque valeur de la liste des valeurs ramenée par la sous-requête, la condition sera vraie si elle est vraie pour au moins une des valeurs renvoyées par la sous-requête. L'opérateur « **SOME** » est identique à l'opérateur « **ANY** ».

L'opérateur « **= ANY** » est équivalent à l'opérateur « **IN** ».



```
SQL> SELECT SOCIETE, VILLE, PAYS FROM CLIENTS
2 WHERE CODE_CLIENT IN ( SELECT CODE_CLIENT FROM EMPLOYES E,COMMANDES C
3 WHERE E.NO_EMPLOYE = C.NO_EMPLOYE AND C.DATE_COMMANDE > '25/06/2011'
5 AND E.PAYS = 'France');
```



SOCIETE	VILLE	PAYS
Bon app'	Marseille	France
La maison d'Asie	Toulouse	France
France restauration	Nantes	France
Blondel père et fils	Strasbourg	France

```

SQL> SELECT SOCIETE, VILLE, PAYS FROM CLIENTS
2 WHERE CODE_CLIENT=ANY( SELECT CODE_CLIENT FROM EMPLOYES E,COMMANDES C
3 WHERE E.NO_EMPLOYE = C.NO_EMPLOYE AND C.DATE_COMMANDE > '25/06/2011'
4 AND E.PAYS = 'France');

```

L'opérateur « **< ANY** » signifie que l'expression est inférieure à au moins une des valeurs donc inférieure au maximum des valeurs de la liste. Dans l'exemple suivant vous pouvez voir les employés qui ont un salaire supérieur à la moyenne des salaires pour les employés qui n'ont pas de supérieur hiérarchique.



```

SQL> SELECT NOM || ' ' || PRENOM EMPLOYE, SALAIRE FROM EMPLOYES
2 WHERE SALAIRE > ANY ( SELECT AVG(SALAIRE) FROM EMPLOYES
3 WHERE REND_COMPTE IS NULL);

```

EMPLOYE	SALAIRE
Fuller Andrew	96000
Brasseur Hervé	147000
Giroux Jean-Claude	150000

L'opérateur « **> ANY** » signifie que l'expression est supérieure à au moins une des valeurs donc supérieure au minimum.



```

SQL> SELECT NOM_PRODUIT, UNITES_COMMANDEES FROM PRODUITS
2 WHERE UNITES_COMMANDEES >ANY( SELECT AVG(UNITES_COMMANDEES)*5
3 FROM PRODUITS GROUP BY CODE_CATEGORIE
4 HAVING AVG(UNITES_COMMANDEES) IS NOT NULL);

```

NOM_PRODUIT	UNITES_COMMANDEES
Vegetable Soup	100
Green Tea	100
Louisiana Hot Spiced Okra	100
Chicken Soup	90
Wimmers gute Semmelknödel	80
Aniseed Syrup	70
Chocolade	70
Røgede sild	70
Gorgonzola Telino	70
Maxilaku	60

La requête précédente affiche les produits et leurs unités commandées pour les commandes qui dépassent 5 fois la plus petite des moyennes des unités commandées par catégorie.

## L'opérateur ALL

L'opérateur « **ALL** » compare une expression à chaque valeur de la liste des valeurs ramenée par la sous-requête ; la condition sera vraie si elle est vraie pour chacune des valeurs renvoyées par la sous-requête.

L'opérateur « **< ALL** » signifie que l'expression est inférieure au minimum et « **> ALL** » signifie que l'expression est supérieure au maximum.



```
SQL> SELECT NOM_PRODUIT, UNITES_STOCK FROM PRODUITS
2 WHERE UNITES_STOCK > ALL ( SELECT UNITES_STOCK FROM PRODUITS
3 WHERE CODE_CATEGORIE = 2 );
```

NOM_PRODUIT	UNITES_STOCK
Boston Crab Meat	123
Rhönbräu Klosterbier	125
Green Tea	125
Potato Chips	200

Dans l'exemple précédent la requête affiche les produits pour lesquels la quantité du stock est supérieure à toutes les quantités des produits de la catégorie 2.



```
SQL> SELECT CODE_CLIENT, NO_EMPLOYE, ANNEE, SUM(PORT) FROM COMMANDES
2 WHERE CODE_CLIENT = 'HANAR'
3 AND NO_EMPLOYE != ALL (SELECT NO_EMPLOYE FROM EMPLOYES
4 WHERE DATE_EMBAUCHE < '01/05/1992')
5 GROUP BY CODE_CLIENT, NO_EMPLOYE, ANNEE;
```

CODE_	NO_EMPLOYE	ANNEE	SUM(PORT)
HANAR	105	2011	1375,7
HANAR	105	2010	2595,7
HANAR	7	2011	1021,9
HANAR	7	2010	1601,8

Dans l'exemple précédent, la requête affiche les commandes pour le client 'HANAR' vendues par un employé embauché avant '01/05/1992'.

L'opérateur « **NOT IN** » est équivalent à l'opérateur « **!= ALL** ».

## Sous-requête renvoyant un tableau

Oracle autorise la présence de plusieurs colonnes dans la clause SELECT d'une sous-requête. Il convient dès lors de préciser dans le premier terme de comparaison de la requête, la liste des colonnes qui doivent être comparées aux lignes des valeurs renvoyées par la sous-requête.



```
SQL> SELECT CODE_CLIENT, NOM_PRODUIT, DATE_COMMANDE,
4 DETAILS_COMMANDES.PRIX_UNITAIRE *
5 DETAILS_COMMANDES.QUANTITE "Achat"
6 FROM COMMANDES NATURAL JOIN DETAILS_COMMANDES JOIN
8 PRODUITS USING( REF_PRODUIT)
9 WHERE (CODE_CLIENT, NO_FOURNISSEUR) IN
10 ( SELECT CODE_CLIENT, NO_FOURNISSEUR
11 FROM CLIENTS JOIN FOURNISSEURS
12 USING(VILLE));
```

CODE_	NOM_PRODUIT	DATE_COMMA	Achat
EASTC	Chang	11/03/2010	1081,08
SEVES	Chang	07/01/2010	9626,76
SEVES	Chai	07/01/2010	2976,6
PARIS	Beer	07/01/2010	12238,2
CONSH	Aniseed Syrup	30/03/2010	4056,48
EASTC	Chang	29/01/2010	4581,72

ALFKI	NuNuCa	Nuß-Nougat-Creme	29/01/2010	7963,68
ALFKI	Granola		29/01/2010	2367,36
SPECD	Mozzarella		29/01/2010	5838,48
SPECD	Beer		29/01/2010	2510,4
SPECD	Hot Cereal		29/01/2010	5018,4
CONSH	Chai		07/02/2010	10018,8
CONSH	Chang		07/02/2010	2728,44
NORTS	Chai		23/03/2010	8639,4
QUEEN	Guaraná Fantástica		07/01/2010	12178,32
ALFKI	Schoggi	Schokolade	23/03/2010	2548,2
ALFKI	Gumbär	Gummibärchen	23/03/2010	12252,24
SPECD	Beer		07/01/2010	8974,68
SPECD	Mozzarella		07/01/2010	8151,84
SPECD	Chartreuse verte		07/01/2010	13440
MEREP	Pears		19/02/2010	7855,68
SPECD	Hot Cereal		22/02/2010	5018,4
SPECD	Mozzarella		22/02/2010	5177,52
EASTC	Chang		11/01/2010	7413,12
EASTC	Chai		11/01/2010	3484,8
AROUT	Chang		24/02/2010	6692,4
...				

Dans l'exemple précédent, la requête affiche les clients, produits, date de commande et valeur partielle de la commande pour les produits achetés par les clients qui habitent dans la même ville que le fournisseur.

## Sous-requête synchronisée

Oracle autorise également le traitement d'une sous-requête faisant référence à une colonne de la table de l'interrogation principale. Le traitement est plus complexe dans ce cas, car il faut évaluer la sous-requête pour chaque ligne traitée par la requête principale. On dit alors que la sous-requête est synchronisée avec la requête principale. La sous-requête est évaluée pour **chaque ligne** de la requête principale.



```
SQL> SELECT CODE_CATEGORIE "Cat", NOM_PRODUIT, UNITES_STOCK "Stock",
2     PRIX_UNITAIRE "Prix" FROM PRODUITS P1
3     WHERE UNITES_STOCK > ( SELECT AVG(UNITES_STOCK)*2 FROM PRODUITS P2
4                           WHERE P2.CODE_CATEGORIE = P1.CODE_CATEGORIE);
```

Cat	NOM_PRODUIT	Stock	Prix
-----			
2	Grandma's Boysenberry Spread	120	125
4	Queso Manchego La Pastora	86	190
3	NuNuCa Nuß-Nougat-Creme	76	70
8	Boston Crab Meat	123	92
4	Raclette Courdavault	79	275
4	Geitost	112	13
3	Valkoinen suklaa	65	81
2	Sirop d'érable	113	143
1	Rhönbräu Klosterbier	125	39
2	Syrup	100	7,5
9	Boysenberry Spread	100	18,75
1	Green Tea	125	2
7	Potato Chips	200	,49

Dans l'exemple précédent la synchronisation entre la requête principale et la sous-requête est indiquée ici par l'utilisation, dans la sous-requête, de la colonne `CODE_CATEGORIE` de la table `PRODUITS` de la requête principale.

### Attention

Il faut faire attention dans le cas des sous-requêtes synchronisées car elles font partie du même espace de nommage que la requête principale. Ainsi il est impératif d'utiliser des alias pour les tables de la requête principale mais également pour les tables des sous-requêtes pour lever l'ambiguïté dans la condition de synchronisation.

L'utilisation des ces alias sont nécessaires uniquement dans les conditions de synchronisation.

## L'opérateur EXISTS

Une des formes particulière de la sous-requête synchronisée est celle testant l'existence de lignes de valeurs répondant à telle ou telle condition.

L'opérateur « **EXISTS** » permet de construire un prédicat évalué à « **TRUE** » si la sous-requête renvoie au moins une ligne.

```
SQL> SELECT SOCIETE, REF_PRODUIT, SUM(PORT)
  2   FROM CLIENTS CL, COMMANDES CO, DETAILS_COMMANDES DC
  3   WHERE CL.CODE_CLIENT    = CO.CODE_CLIENT
  4         AND CO.NO_COMMANDE = DC.NO_COMMANDE
  5         AND CO.DATE_COMMANDE > '28/06/2011'
  6         AND EXISTS ( SELECT * FROM PRODUITS PR, FOURNISSEURS FR
  7                       WHERE PR.NO_FOURNISSEUR = FR.NO_FOURNISSEUR
  8                             AND DC.REF_PRODUIT = PR.REF_PRODUIT AND FR.VILLE = CL.VILLE)
  9   GROUP BY SOCIETE, REF_PRODUIT;
```

SOCIETE	REF_PRODUIT	SUM(PORT)
Alfreds Futterkiste	25	64,8
Alfreds Futterkiste	26	120,4
Alfreds Futterkiste	104	55,6
North/South	3	93,3
Mère Paillarde	54	76,8
Mère Paillarde	109	63
Comércio Mineiro	24	85,1
Queen Cozinha	24	81,4

Dans l'exemple précédent, la requête affiche les clients, la référence produit et les frais de port pour les produits achetés par les clients qui habitent dans la même ville que le fournisseur après '28/06/2011'.

Il est à noter que la projection totale (\*) de la sous-requête est sans signification, puisque seul compte le fait que la sous-requête renvoie ou non une ligne. La projection peut donc être une constante quelconque.

```
SQL> SELECT SOCIETE, REF_PRODUIT, SUM(PORT)
  2   FROM CLIENTS CL, COMMANDES CO, DETAILS_COMMANDES DC
  3   WHERE CL.CODE_CLIENT    = CO.CODE_CLIENT
  4         AND CO.NO_COMMANDE = DC.NO_COMMANDE
  5         AND CO.DATE_COMMANDE > '25/06/2011'
  6         AND EXISTS ( SELECT 'constante'
  7                       FROM PRODUITS PR, FOURNISSEURS FR
  8                       WHERE PR.NO_FOURNISSEUR = FR.NO_FOURNISSEUR
  9                             AND DC.REF_PRODUIT = PR.REF_PRODUIT
 10                             AND FR.VILLE     = CL.VILLE)
```

```

11 AND EXISTS ( SELECT 'constante'
12              FROM EMPLOYES EM, COMMANDES CO1
13              WHERE EM.NO_EMPLOYE = CO1.NO_EMPLOYE
14                    AND REGION LIKE
15                      (SELECT REGION||'%' FROM EMPLOYES
16                      WHERE FONCTION = 'Vice-Président'
17                        AND NOM      = 'Brasseur')
18              AND CO.NO_COMMANDE = CO1.NO_COMMANDE)
19 GROUP BY SOCIETE, REF_PRODUIT;

```

SOCIETE	REF_PRODUIT	SUM(PORT)
Alfreds Futterkiste	26	120,4
Alfreds Futterkiste	104	55,6
Alfreds Futterkiste	25	64,8
Lehmanns Marktstand	75	57
Around the Horn	2	120,7
Around the Horn	1	52,1
Lehmanns Marktstand	64	131,2
Around the Horn	3	148,8
North/South	3	93,3
Lehmanns Marktstand	29	57
Lehmanns Marktstand	77	57

Dans l'exemple précédent, la requête affiche les clients, la référence produit et les frais de port pour les produits achetés par les clients qui habitent dans la même ville que le fournisseur après '25/06/2011' et les ventes effectuées par les employés qui ont une région d'affectation qui commence par la région de l'employé 'Vice-Président' qui s'appelle 'Brasseur'.



```

SQL> SELECT NOM||' '||PRENOM EMPLOYE
2 FROM EMPLOYES
3 WHERE NO_EMPLOYE IN
4 ( SELECT NO_EMPLOYE
5   FROM COMMANDES CO1, CLIENTS CL, DETAILS_COMMANDES DC
6   WHERE CO1.CODE_CLIENT = CL.CODE_CLIENT
7         AND CO1.NO_COMMANDE = DC.NO_COMMANDE
8         AND PORT >
9           ( SELECT AVG(PORT)*1.38 FROM COMMANDES CO2
10             WHERE CO1.CODE_CLIENT = CO2.CODE_CLIENT
11               AND CO1.ANNEE = CO2.ANNEE)
12 AND EXISTS
13   ( SELECT 'constante'
14     FROM PRODUITS PR1
15     WHERE UNITES_STOCK >
16       ( SELECT AVG(UNITES_STOCK)*2
17         FROM PRODUITS PR2
18         WHERE PR1.NO_FOURNISSEUR = PR2.NO_FOURNISSEUR)
19       AND PR1.REF_PRODUIT = DC.REF_PRODUIT)
20 );

```

EMPLOYE

```

-----
King Robert
Marielle Michel
Viry Yvan
Regner Charles

```

Kremser Arnaud  
 Poidatz Benoît  
 Gregoire Renée  
 Mure Guy  
 Urbaniak Isabelle

La requête précédente retourne la liste des employés qui ont effectué des commandes avec des frais de port supérieurs à 38% de la moyenne de frais de port pour le même client dans l'année. En sus les commandes comportent des produits pour lesquels les unités en stock sont supérieures à deux fois la moyenne des produits en stock pour le même fournisseur. Il faut également remarquer qu'il s'agit d'une sous-requête qui elle-même comporte deux sous-requêtes synchronisées.

## Sous-requête dans la clause FROM

Depuis la version 7.2 d'Oracle, vous pouvez utiliser directement une sous-requête dans la clause « **FROM** » de la requête principale.



```
SQL> SELECT A.NO_FOURNISSEUR, CODE_CATEGORIE, SUM(UNITES_STOCK) "Stock",
2  ROUND( 100*SUM(UNITES_STOCK)/B.SUM_FOUR) "%Stock"
3  FROM PRODUITS A,( SELECT NO_FOURNISSEUR, SUM(UNITES_STOCK) SUM_FOUR
4                    FROM PRODUITS GROUP BY NO_FOURNISSEUR) B
5  WHERE A.NO_FOURNISSEUR = B.NO_FOURNISSEUR
6  GROUP BY A.NO_FOURNISSEUR, CODE_CATEGORIE, B.SUM_FOUR ;
```

NO_FOURNISSEUR	CODE_CATEGORIE	Stock	%Stock
6	8	24	8
16	1	283	67
26	5	57	29
7	1	15	12
6	7	35	12
7	8	42	32
29	2	113	87
26	9	100	51
26	2	40	20
14	2	80	20
4	2	60	24
14	1	175	43
23	7	200	60
20	3	20	31
25	9	40	23
27	9	40	21
1	2	13	19
12	7		

...

Dans l'exemple précédent, la sous-requête calcule, pour chaque fournisseur, la somme des produits en stock ; cette somme est utilisée dans la requête principale pour calculer le pourcentage par fournisseur du stock de chaque catégorie.



```
SQL> SELECT SOCIETE, ANNEE, TRIMESTRE, TO_CHAR(PORT,'9999D00') PORT
2  FROM ( SELECT SOCIETE, ANNEE, TRIMESTRE, SUM(PORT) PORT
3        FROM COMMANDES CO1, CLIENTS CL
4        WHERE CO1.CODE_CLIENT = CL.CODE_CLIENT
5        GROUP BY SOCIETE, ANNEE, TRIMESTRE ) SR1
6  WHERE PORT IN
```

```

7      ( SELECT MAX(PORT)
8      FROM ( SELECT SOCIETE, ANNEE, TRIMESTRE, SUM(PORT) PORT
9      FROM COMMANDES CO1, CLIENTS CL
10     WHERE CO1.CODE_CLIENT = CL.CODE_CLIENT
11     GROUP BY SOCIETE, ANNEE, TRIMESTRE ) SR2
12     GROUP BY ANNEE, TRIMESTRE )
13 ORDER BY ANNEE, TRIMESTRE;

```

SOCIETE	ANNEE	TRIMESTRE	PORT
Ernst Handel	2010	1	2670,80
Frankenversand	2010	2	2531,50
Seven Seas Imports	2010	3	3179,80
Berglunds snabbköp	2010	4	2995,80
HILARIÓN-Abastos	2011	1	3742,50
Mère Paillarde	2011	2	3780,00

L'exemple précédent affiche les sociétés qui ont la plus grande somme des frais de port par année et par trimestre. La première sous-requête récupère les sommes des frais de port pour tous les clients par année et par trimestre. La deuxième sous-requête permet de filtrer uniquement les enregistrements qui ont les sommes des frais de port maximum pour chaque année et par trimestre.



```

SQL> SELECT * FROM
2      SELECT GROUPING_ID(ANNEE,TRIMESTRE,CODE_CLIENT) "GI",
3      ANNEE, TRIMESTRE, CODE_CLIENT CLIENT, SUM(PORT) PORT
4      FROM COMMANDES GROUP BY ROLLUP(ANNEE,TRIMESTRE,CODE_CLIENT) ) SR1
5      WHERE CLIENT BETWEEN 'ALFKI' AND 'ANATR' OR GI > 0;

```

GI	ANNEE	TRIMESTRE	CLIEN	PORT
0	2010	1	ALFKI	2444,3
0	2010	1	ANATR	1768,3
<b>1</b>	<b>2010</b>	<b>1</b>		<b>145808,3</b>
0	2010	2	ALFKI	2028,9
0	2010	2	ANATR	1137
<b>1</b>	<b>2010</b>	<b>2</b>		<b>147223,4</b>
0	2010	3	ALFKI	1261,7
0	2010	3	ANATR	1628,7
<b>1</b>	<b>2010</b>	<b>3</b>		<b>150755,8</b>
0	2010	4	ALFKI	1370,9
0	2010	4	ANATR	1586,6
<b>1</b>	<b>2010</b>	<b>4</b>		<b>152525,2</b>
<b>3</b>	<b>2010</b>			<b>596312,7</b>
0	2011	1	ALFKI	1738,6
0	2011	1	ANATR	2059,2
<b>1</b>	<b>2011</b>	<b>1</b>		<b>205558,4</b>
0	2011	2	ALFKI	1772,7
0	2011	2	ANATR	2473,3
<b>1</b>	<b>2011</b>	<b>2</b>		<b>205582,4</b>
<b>3</b>	<b>2011</b>			<b>411140,8</b>
<b>7</b>				<b>1007453,5</b>

La requête précédente permet d'afficher le cumul des frais de port par année et trimestre pour les clients 'ALFKI' et 'ANATR'. Tous les enregistrements avec le champ 'GI' supérieure à 0 représentent les sommes des frais de port pour tous les clients. Ainsi il est possible de filtrer pour l'affichage une partie des enregistrements et de garder les valeurs des enregistrements agrégées pour l'ensemble des valeurs.

# La clause WITH

La clause « **WITH** » permet d'assigner à une ou plusieurs sous-requêtes un alias afin de pouvoir l'utiliser à différents endroits dans la requête principale ou dans les sous-requêtes utilisées dans celle-ci.

La syntaxe de la clause est la suivante :

```
WITH alias_sous-requête AS ( sous-requête )[,...]
SELECT ...
```

L'exemple suivant affiche les sociétés qui ont la plus grande somme des frais de port par année et par trimestre.



```
SQL> WITH SAT_P AS
2      ( SELECT SOCIETE, ANNEE, TRIMESTRE, SUM(PORT) PORT
3        FROM COMMANDES CO1, CLIENTS CL
4        WHERE CO1.CODE_CLIENT = CL.CODE_CLIENT
5        GROUP BY SOCIETE, ANNEE, TRIMESTRE )
6  SELECT SOCIETE, ANNEE, TRIMESTRE, TO_CHAR(PORT,'9999D00') PORT
7  FROM SAT_P
8  WHERE PORT IN ( SELECT MAX(PORT) FROM SAT_P
9                  GROUP BY ANNEE, TRIMESTRE )
10 ORDER BY ANNEE, TRIMESTRE;
```

SOCIETE	ANNEE	TRIMESTRE	PORT
Ernst Handel	2010	1	2670,80
Frankenversand	2010	2	2531,50
Seven Seas Imports	2010	3	3179,80
Berglunds snabbköp	2010	4	2995,80
HILARIÓN-Abastos	2011	1	3742,50
Mère Paillard	2011	2	3780,00

L'utilisation de la clause « **WITH** » permet d'optimiser les requêtes dans le cas d'utilisations multiples de la même sous-requête. Voici un exemple avec l'utilisation de la commande « **SET TIMING ON** » qui permet d'afficher le temps d'exécution d'une requête ou d'un bloc **PL/SQL**.



```
SQL> SET TIMING ON
SQL> WITH SAT_P AS
2      ( SELECT SOCIETE, ANNEE, TRIMESTRE, SUM(PORT) PORT
3        FROM COMMANDES CO1, CLIENTS CL
4        WHERE CO1.CODE_CLIENT = CL.CODE_CLIENT
5        GROUP BY SOCIETE, ANNEE, TRIMESTRE )
6  SELECT SOCIETE, ANNEE, TRIMESTRE, TO_CHAR(PORT,'9999D00') PORT
7  FROM SAT_P
8  WHERE PORT IN ( SELECT MAX(PORT) FROM SAT_P
9                  GROUP BY ANNEE, TRIMESTRE );
```

...  
Ecoulé : 00 :00 :00.20

```
SQL> SELECT SOCIETE, ANNEE, TRIMESTRE, TO_CHAR(PORT,'9999D00') PORT
2  FROM ( SELECT SOCIETE, ANNEE, TRIMESTRE, SUM(PORT) PORT
3        FROM COMMANDES CO1, CLIENTS CL
4        WHERE CO1.CODE_CLIENT = CL.CODE_CLIENT
```



```

5      GROUP BY SOCIETE, ANNEE, TRIMESTRE ) SR1
6  WHERE PORT IN
7      ( SELECT MAX(PORT)
8          FROM ( SELECT SOCIETE, ANNEE, TRIMESTRE, SUM(PORT) PORT
9                  FROM COMMANDES CO1, CLIENTS CL
10                 WHERE CO1.CODE_CLIENT = CL.CODE_CLIENT
11                   GROUP BY SOCIETE, ANNEE, TRIMESTRE ) SR2
12         GROUP BY ANNEE, TRIMESTRE
13         );
...

```

Ecoulé : 00 :01 :21.29

La requête suivante permet d'afficher la somme des ventes par années, par client et par produit. Seuls sont affichés les enregistrements pour les clients qui ont la moyenne des frais de port annuels supérieurs à  $\text{'MAX(AVG(PORT)) + AVG(AVG(PORT)) / 2'}$ , ainsi que la moyenne annuelle des quantités supérieure à  $\text{'MAX(AVG(QUANTITE)) + AVG(AVG(QUANTITE)) / 2'}$ .



```

SQL> WITH ATCP_SCA AS
2      ( SELECT ANNEE, CODE_CLIENT CLIENT,
3              REF_PRODUIT PRODUIT, AVG(PORT) MP,
4              AVG(QUANTITE) MQ,
5              SUM(PRIX_UNITAIRE*QUANTITE) CA
6          FROM COMMANDES NATURAL JOIN DETAILS_COMMANDES
7          GROUP BY ANNEE, CODE_CLIENT, REF_PRODUIT ),
8  AT_MQ  AS
9      ( SELECT ANNEE, PRODUIT, (MAX(MQ)+AVG(MQ))/2 MQ
10        FROM ATCP_SCA GROUP BY ANNEE, PRODUIT ),
11  AT_MP  AS
12      ( SELECT ANNEE, CLIENT, (MAX(MP)+AVG(MP))/2 MP
13        FROM ATCP_SCA GROUP BY ANNEE, CLIENT )
14  SELECT ANNEE, CLIENT, PRODUIT,
15         TO_CHAR(CA, '999G999D00') CA
16  FROM ATCP_SCA SQ1
17  WHERE MQ > ( SELECT MQ FROM AT_MQ SQ2
18               WHERE SQ1.ANNEE      = SQ2.ANNEE
19                 AND SQ1.PRODUIT     = SQ2.PRODUIT)
20  AND MP > ( SELECT MP FROM AT_MP SQ3
21             WHERE SQ1.ANNEE      = SQ3.ANNEE
22               AND SQ1.CLIENT      = SQ3.CLIENT)
23  ORDER BY ANNEE, CLIENT, PRODUIT;

```

ANNEE	CLIEN	PRODUIT	CA
2010	ANATR	75	132 410,16
2010	ANATR	76	124 763,76
2010	ANATR	113	162 624,48
2010	BERGS	36	168 150,00
2010	BLONP	21	193 637,52
2010	BOLID	16	153 763,20
2010	BONAP	82	185 581,44
2010	BOTTM	10	202 351,32
2010	BOTTM	19	264 228,12
2010	BOTTM	37	246 648,84

...

- *Fonctions de classement*
- *Partitionnement*
- *Fenêtres*
- *PARTITION BY OUTER*

# 9

## Les fonctions analytiques



### Objectifs

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- Utiliser les fonctions d'agrégation multidimensionnelles.
- Effectuer des regroupements et calculer les résultats intermédiaires.
- Utiliser les fonctions de classement.
- Effectuer des requêtes avec des valeurs détaillées et résultats des regroupements.
- Effectuer des calculs suivant un partitionnement de données.
- Effectuer des calculs utilisant le concept de fenêtrage.



### Contenu

Les fonctions analytiques	9-2	L'opérateur OUTER JOIN	9-15
La clause de partitionnement	9-2	Fonctions de classement	9-18
La clause d'ordre	9-6	Fonctions de fenêtre	9-27
La clause de fenêtrage	9-8		

# Les fonctions analytiques

Oracle propose plusieurs fonctions de classement et de statistiques qui auraient nécessité par le passé de nombreuses lignes de code SQL ou d'importer les données dans une application tierce.

Les fonctions analytiques opèrent sur un ensemble de lignes définies de manière relative; elles se distinguent des fonctions agrégats qui opèrent sur l'ensemble d'un groupe défini par le « **GROUP BY** ». En ce sens, les fonctions analytiques sont d'une granularité plus fine permettent d'analyser des données, notamment de calculer des moyennes mobiles, des classements et des valeurs cumulées. Ces fonctions fournissent un accès à plus d'une ligne d'une table sans auto-jointure.

Vous pouvez inclure une fonction dans une sélection ou dans la clause « **ORDER BY** » d'une instruction « **SELECT** ».

La syntaxe est la suivante :

**FONCTION ( EXPRESSION[,...] ) OVER ( [PARTITION] [ORDRE] [FENETRE])**

<b>OVER</b>	Indique qu'on utilise une fonction analytique.
<b>PARTITION</b>	La clause de partitionnement définit un découpage des données suivant les valeurs des expressions. Chaque valeur définit un groupe logique à l'intérieur duquel est appliquée la fonction analytique. C'est analogue à la clause « <b>GROUP BY</b> » pour les calculs dans les fonctions analytiques.
<b>ORDRE</b>	La clause d'ordre indique comment les données sont triées à l'intérieur de chaque partition.
<b>FENETRE</b>	La clause de fenêtrage indique l'ensemble des lignes sur lesquelles doit être appliquée la fonction. Si une clause de fenêtrage est spécifiée, une clause d'ordre doit obligatoirement l'être aussi.

Le module va commencer par l'étude de l'extension analytique utilisée avec les fonctions d'agrégat présentes précédemment.

## La clause de partitionnement

Les fonctions analytiques permettent aux utilisateurs de diviser des jeux de résultat en groupes de lignes, appelés partitions. Ce terme de partitions utilisé dans les fonctions analytiques n'a aucun lien avec les tables partitionnées d'Oracle.

Les partitions sont créées seulement après que les groupes aient été définis par la clause « **GROUP BY** » ; de ce fait, elles sont disponibles pour n'importe quel agrégat, tels que les sommes et moyennes. La division des partitions peut être basée sur n'importe quelle colonne ou expression. Le jeu de résultat d'une requête peut être partitionné dans une partition unique contenant toutes les lignes, ou plusieurs partitions plus petites, contenant quelques lignes seulement.

La syntaxe pour la clause de partitionnement est :

**PARTITION BY EXPRESSION[,...]**

```
SQL> SELECT NO_EMPLOYE, FONCTION, SALAIRE,
2      SUM( SALAIRE) OVER ( PARTITION BY FONCTION) "Somme Salaire"
3      FROM EMPLOYES;
```

```
NO_EMPLOYE  FONCTION                                SALAIRE  Somme Salaire
-----
```



```

...
64 Assistante commerciale          1700          16540
21 Assistante commerciale          1800          16540
89 Assistante commerciale          2000          16540
27 Assistante commerciale          1400          16540
33 Chef des ventes                 12000          83000
24 Chef des ventes                 13000          83000
23 Chef des ventes                 16000          83000
11 Chef des ventes                 10000          83000
95 Chef des ventes                 19000          83000
86 Chef des ventes                 13000          83000
37 Président                      150000         150000
70 Représentant(e)                 9400          692900
...

```

Dans l'exemple, la fonction « **SUM** » est la fonction analytique; elle a le même nom que la fonction d'agrégat « **SUM** » mais le mot-clé « **OVER** » indique qu'on utilise une fonction analytique. « **PARTITION BY** » indique un découpage logique; la fonction analytique utilisée ne calcule qu'à l'intérieur d'une partition. C'est tout à fait similaire au « **GROUP BY** » dans une requête traditionnelle.



```

SQL> SELECT NOM_PRODUIT "Produit", CODE_CATEGORIE "Cat",
2         NO_FOURNISSEUR "Four", UNITES_STOCK "US",
3         SUM(UNITES_STOCK) OVER (PARTITION BY CODE_CATEGORIE) "SUSC",
4         AVG(UNITES_STOCK) OVER (PARTITION BY NO_FOURNISSEUR) "AUSF"
5         FROM PRODUITS;

```

Produit	Cat	Four	US	SUSC	AUSF
Chang	1	1	17	874	23.00
Aniseed Syrup	2	1	13	1032	23.00
Chai	1	1	39	874	23.00
Amandes	7	2	20	589	30.43
Cherry Pie Filling	9	2	40	460	30.43
Chef Anton's Cajun Seasoning	2	2	53	1032	30.43
Chef Anton's Gumbo Mix	2	2	0	1032	30.43
Brownie Mix	3	2	20	466	30.43
...					
Guaraná Fantástica	1	10		874	
Granola	5	11	100	642	60.00
Schoggi Schokolade	3	11	49	466	60.00
Gumbär Gummibärchen	3	11	15	466	60.00
NuNuCa Nuß-Nougat-Creme	3	11	76	466	60.00
Rhönbräu Klosterbier	1	12	125	874	59.67
Thüringer Rostbratwurst	6	12		136	59.67
Wimmers gute Semmelknödel	5	12	22	642	59.67
Original Frankfurter grüne Soße	2	12	32	1032	59.67
Rössle Sauerkraut	7	12		589	59.67
...					

Comme vous pouvez le constater dans l'exemple, on peut, grâce aux fonctions analytiques, comparer des valeurs non agrégées avec des valeurs agrégées.

Le traitement de la valeur « **NULL** » dans les fonctions analytiques, correspondent effectuée dans les fonctions SQL d'agrégation. D'autres types de traitements peuvent être obtenus à l'aide des fonctions « **DECODE** » ou « **CASE** ».



## Conseil

Vous pouvez utiliser n'importe quelle fonction d'agrégat « **SUM** », « **AVG** », « **COUNT** » ... comme des fonctions analytiques avec l'indicateur « **OVER** ».

Ainsi à l'aide des fonctions analytiques, vous pouvez calculer des moyennes mobiles, des classements et des valeurs cumulées mais également calculer des expressions avec les valeurs non agrégées.



```
SQL> SELECT NOM, FONCTION, SALAIRE, SALAIRE /
2      SUM( SALAIRE) OVER ( PARTITION BY FONCTION) "% Fonction",
3      SALAIRE / SUM( SALAIRE) OVER () "% Total" FROM EMPLOYES;
```

NOM	FONCTION	SALAIRE	% Fonction	% Total
Etienne	Assistante commerciale	2000	0,1209	0,0017
Grangirard	Assistante commerciale	1700	0,1028	0,0014
Ziliox	Assistante commerciale	1900	0,1149	0,0016
Callahan	Assistante commerciale	1200	0,0726	0,0010
Devie	Assistante commerciale	1540	0,0931	0,0013
Lampis	Assistante commerciale	1300	0,0786	0,0011
Guerdon	Assistante commerciale	1700	0,1028	0,0014
Poupard	Assistante commerciale	1800	0,1088	0,0015
Pouetre	Assistante commerciale	2000	0,1209	0,0017
Maurer	Assistante commerciale	1400	0,0846	0,0012
Chambaud	Chef des ventes	12000	0,1446	0,0101
Buchanan	Chef des ventes	13000	0,1566	0,0110
Splingart	Chef des ventes	16000	0,1928	0,0135
Belin	Chef des ventes	10000	0,1205	0,0084
Leger	Chef des ventes	19000	0,2289	0,0160
Ragon	Chef des ventes	13000	0,1566	0,0110
Giroux	Président	150000	1,0000	0,1265
Berlioz	Représentant(e)	9400	0,0136	0,0079
Nocella	Représentant(e)	7600	0,0110	0,0064
Herve	Représentant(e)	6700	0,0097	0,0057
...				

## Attention



Les fonctions analytiques peuvent être considérées comme des fonctions "horizontales" car elles ne nécessitent pas l'utilisation de la clause « **GROUP BY** ». Mais le comportement des ces fonctions permet de calculer des agrégats suivant la syntaxe utilisée.

Toutefois il faut faire attention aux arguments utilisés avec ces fonctions car eux doivent être au niveau d'agrégat de l'enregistrement de la requête.

Ainsi tout argument passé à une fonction analytique doit pouvoir être affiché sur le même enregistrement.

Dans l'exemple suivant, vous pouvez voir l'expression `SUM(SUM(PORT))` qui est nécessaire car il s'agit d'un enregistrement d'une requête déjà agrégé. Ainsi l'argument de la fonction analytique est `SUM(PORT)` qui respecte le niveau d'agrégation de l'enregistrement.



```
SQL> SELECT PAYS, ANNEE, TRIMESTRE T, SUM(PORT) PORT,
2      SUM(SUM(PORT)) OVER ( PARTITION BY PAYS) "S Pays",
3      SUM(SUM(PORT)) OVER ( PARTITION BY ANNEE) "S Année"
4 FROM COMMANDES CO1, CLIENTS CL WHERE CO1.CODE_CLIENT = CL.CODE_CLIENT
5 GROUP BY PAYS, ANNEE, TRIMESTRE;
```

PAYS	ANNEE	T	PORT	S Pays	S Année
------	-------	---	------	--------	---------

Allemagne	2010	2	19374,6	123	888,90	596	312,70
Allemagne	2011	2	24553,9	123	888,90	411	140,80
Allemagne	2011	1	25502,9	123	888,90	411	140,80
Allemagne	2010	1	20301,7	123	888,90	596	312,70
Allemagne	2010	3	17477,2	123	888,90	596	312,70
Allemagne	2010	4	16678,6	123	888,90	596	312,70
Argentine	2011	1	5627,5	32	640,40	411	140,80
Argentine	2010	4	4828,6	32	640,40	596	312,70
Argentine	2010	3	4669,1	32	640,40	596	312,70
Argentine	2010	1	5042,4	32	640,40	596	312,70
Argentine	2011	2	6912,5	32	640,40	411	140,80
Argentine	2010	2	5560,3	32	640,40	596	312,70
Autriche	2010	4	2829,5	19	691,20	596	312,70
Autriche	2010	3	2703,9	19	691,20	596	312,70
...							

### Attention

Les fonctions analytiques peuvent permettre de calculer les agrégats uniquement pour les enregistrements retournés par la requête suivant les filtres définis dans la clause « **WHERE** » ou dans la clause « **HAVING** ».

Ainsi, comme pour les fonctions d'agrégat, il faut bien définir l'ensemble des enregistrements pour lesquels les agrégats sont calculés.

Dans l'exemple suivant, la première requête retourne tous les enregistrements des trois tables CLIENTS, COMMANDES et DETAILS\_COMMANDES regroupés par PAYS, ANNEE et TRIMESTRE. Ainsi les valeurs retournées par les deux fonctions analytiques portent sur l'ensemble des enregistrements.

```
SQL> SELECT PAYS, ANNEE, TRIMESTRE T, SUM(PRIX_UNITAIRE*QUANTITE) CA,
2 SUM(SUM(PRIX_UNITAIRE*QUANTITE)) OVER( PARTITION BY PAYS) "S Pays",
3 SUM(SUM(PRIX_UNITAIRE*QUANTITE)) OVER( PARTITION BY ANNEE) "S Année"
4 FROM CLIENTS NATURAL JOIN COMMANDES NATURAL JOIN DETAILS_COMMANDES
5 GROUP BY PAYS, ANNEE, TRIMESTRE ORDER BY PAYS, ANNEE, TRIMESTRE;
```

PAYS	ANNEE	T	CA	S Pays	S Année
Allemagne	2010	1	68 206 106	420 063 829	2 019 971 906
Allemagne	2010	2	66 363 057	420 063 829	2 019 971 906
Allemagne	2010	3	58 831 185	420 063 829	2 019 971 906
Allemagne	2010	4	55 341 074	420 063 829	2 019 971 906
Allemagne	2011	1	87 169 620	420 063 829	1 403 112 414
Allemagne	2011	2	84 152 788	420 063 829	1 403 112 414
Argentine	2010	1	17 302 196	112 213 362	2 019 971 906
Argentine	2010	2	19 789 972	112 213 362	2 019 971 906
Argentine	2010	3	14 884 704	112 213 362	2 019 971 906
Argentine	2010	4	16 707 965	112 213 362	2 019 971 906
Argentine	2011	1	19 582 983	112 213 362	1 403 112 414
...					

La deuxième requête ne porte que sur les ventes du produit numéro 1 ; ainsi les fonctions analytiques ne portent que sur ces enregistrements.

```
SQL> SELECT PAYS, ANNEE, TRIMESTRE T, SUM(PRIX_UNITAIRE*QUANTITE) CA,
2 SUM(SUM(PRIX_UNITAIRE*QUANTITE)) OVER( PARTITION BY PAYS) "S Pays",
3 SUM(SUM(PRIX_UNITAIRE*QUANTITE)) OVER( PARTITION BY ANNEE) "S Année"
4 FROM CLIENTS NATURAL JOIN COMMANDES NATURAL JOIN DETAILS_COMMANDES
5 WHERE REF_PRODUIT = 1
```

6 GROUP BY PAYS, ANNEE, TRIMESTRE ORDER BY PAYS, ANNEE, TRIMESTRE;

PAYS	ANNEE	T	CA	S Pays	S Année
-----	-----	-----	-----	-----	-----
Allemagne	2010	1	523 228	3 520 664	17 912 671
Allemagne	2010	2	568 168	3 520 664	17 912 671
Allemagne	2010	3	465 076	3 520 664	17 912 671
Allemagne	2010	4	446 998	3 520 664	17 912 671
Allemagne	2011	1	822 921	3 520 664	12 296 552
Allemagne	2011	2	694 274	3 520 664	12 296 552
Argentine	2010	1	150 064	940 678	17 912 671
Argentine	2010	2	145 345	940 678	17 912 671
Argentine	2010	3	194 713	940 678	17 912 671
Argentine	2010	4	136 924	940 678	17 912 671
Argentine	2011	1	153 985	940 678	12 296 552
Argentine	2011	2	159 647	940 678	12 296 552
...					

La troisième requête ne porte que sur les ventes du premier trimestre de chaque année ; ainsi les fonctions analytiques ne portent que sur ces enregistrements.



```
SQL> SELECT PAYS, ANNEE, TRIMESTRE T, SUM(PRIX_UNITAIRE*QUANTITE) CA,
2 SUM(SUM(PRIX_UNITAIRE*QUANTITE)) OVER ( PARTITION BY PAYS) "S Pays",
3 SUM(SUM(PRIX_UNITAIRE*QUANTITE)) OVER ( PARTITION BY ANNEE) "S Année"
4 FROM CLIENTS NATURAL JOIN COMMANDES NATURAL JOIN DETAILS_COMMANDES
5 GROUP BY PAYS, ANNEE, TRIMESTRE HAVING TRIMESTRE = 1
6 ORDER BY PAYS, ANNEE, TRIMESTRE;
```

PAYS	ANNEE	T	CA	S Pays	S Année
-----	-----	-----	-----	-----	-----
Allemagne	2010	1	68 206 106	155 375 726	496 085 219
Allemagne	2011	1	87 169 620	155 375 726	701 034 567
Argentine	2010	1	17 302 196	36 885 179	496 085 219
Argentine	2011	1	19 582 983	36 885 179	701 034 567
Autriche	2010	1	14 329 620	25 726 680	496 085 219
Autriche	2011	1	11 397 060	25 726 680	701 034 567
Belgique	2010	1	13 136 272	31 205 199	496 085 219
Belgique	2011	1	18 068 927	31 205 199	701 034 567
...					

## La clause d'ordre

Le traitement d'une requête utilisant une fonction analytique requière trois étapes :

- Premièrement, toutes les jointures, les clauses « WHERE », « GROUP BY » et « HAVING » sont exécutées.
- Deuxièmement, le jeu de résultat est rendu disponible aux fonctions analytiques qui procèdent aux calculs.
- Troisièmement, si la requête inclut à la fin une clause « ORDER BY », celle-ci est exécutée.

La clause d'ordre indique comment les données sont triées à l'intérieur de chaque partition ; la syntaxe de la clause est :

```
ORDER BY EXPRESSION [ASC|DESC] [NULLS {FIRST|LAST}][,...]
```



```
SQL> SELECT SOCIETE, DATE_COMMANDE, PORT P,
2 SUM(PORT) OVER ( PARTITION BY CODE_CLIENT ORDER BY DATE_COMMANDE) SCC,
3 SUM(PORT) OVER ( ORDER BY CODE_CLIENT, DATE_COMMANDE) SCT
4 FROM CLIENTS NATURAL JOIN COMMANDES
5 WHERE ANNEE = 2011 AND MOIS = 5 AND PAYS = 'France';
```

SOCIETE	DATE_COMMA	P	SCC	SC	SCT
Blondel père et fils	12/05/2011	56,0	56,0	1 213,8	56,0
Blondel père et fils	13/05/2011	71,5	127,5	1 213,8	127,5
Blondel père et fils	15/05/2011	90,0	217,5	1 213,8	217,5
Blondel père et fils	16/05/2011	51,9	269,4	1 213,8	269,4
Blondel père et fils	17/05/2011	83,5	352,9	1 213,8	352,9
Blondel père et fils	19/05/2011	78,3	431,2	1 213,8	431,2
Blondel père et fils	20/05/2011	76,3	507,5	1 213,8	507,5
Blondel père et fils	21/05/2011	85,3	592,8	1 213,8	592,8
Blondel père et fils	22/05/2011	83,7	676,5	1 213,8	676,5
Blondel père et fils	23/05/2011	80,2	756,7	1 213,8	756,7
Blondel père et fils	24/05/2011	67,7	824,4	1 213,8	824,4
Blondel père et fils	25/05/2011	89,9	914,3	1 213,8	914,3
Blondel père et fils	26/05/2011	83,7	998,0	1 213,8	998,0
Blondel père et fils	27/05/2011	67,8	1 065,8	1 213,8	1 065,8
Blondel père et fils	29/05/2011	54,9	1 120,7	1 213,8	1 120,7
Blondel père et fils	30/05/2011	93,1	<b>1 213,8</b>	<b>1 213,8</b>	<b>1 213,8</b>
Bon app'	01/05/2011	90,4	90,4	1 418,9	1 304,2
Bon app'	02/05/2011	67,5	157,9	1 418,9	1 371,7
...					

Quand on utilise la clause d'ordre dans une fonction analytique et il n'y a pas de clause de fenêtrage ; toutes les lignes depuis la première jusqu'à celle en cours selon l'ordre induit par le « **ORDER BY** » sont passées à la fonction analytique qui calcule donc ici la somme cumulée. Pour la fonction analytique qui ne spécifié pas la clause d'ordre, toutes les lignes de la partition sont utilisées pour le calcul de la somme totale.

### Attention



Vous avez pu remarquer que suite à l'utilisation des fonctions analytiques qui comportent la clause « **ORDER BY** », l'affichage des enregistrements de la requête est effectué dans le même ordre de tri.

Attention si plusieurs fonctions analytiques qui utilisent des ordres de tris différents sont employées dans la même requête et si la clause « **ORDER BY** » au niveau de la requête n'est pas utilisée, alors les résultats sont affichés triés uniquement suivant le premier ordre de tri dans l'ordre de l'écriture de la requête.

Si vous voulez spécifier un autre tri, soit vous inversez l'ordre des fonctions analytiques, soit vous pouvez définir la clause « **ORDER BY** » au niveau de la requête. Cette clause d'ordre « **ORDER BY** » est nécessaire pour certaines fonctions. Quand elle est utilisée, elle modifie le comportement de la fonction. En effet, il faut savoir que si une clause « **ORDER BY** » est utilisée et qu'une clause de fenêtrage n'est pas spécifiée, une clause de fenêtrage implicite est appliquée « **UNBOUNDED PRECEDING** ».



```
SQL> SELECT SOCIETE, DATE_COMMANDE, NO_EMPLOYE E, PORT P,
2 SUM(PORT) OVER ( PARTITION BY CODE_CLIENT ORDER BY DATE_COMMANDE) SCC,
3 SUM(PORT) OVER ( PARTITION BY NO_EMPLOYE ORDER BY PORT DESC ) SCE
4 FROM CLIENTS NATURAL JOIN COMMANDES
5 WHERE ANNEE = 2011 AND MOIS = 5 AND PAYS = 'France';
```



SOCIETE	DATE_COMMA	E	P	SCC	SCE
Blondel père et fils	12/05/2011	110	56,0	56,0	1 848,9
Blondel père et fils	13/05/2011	111	71,5	127,5	2 376,2
Blondel père et fils	15/05/2011	111	90,0	217,5	932,4
Blondel père et fils	16/05/2011	45	51,9	269,4	2 612,0
Blondel père et fils	17/05/2011	29	83,5	352,9	981,2
Blondel père et fils	19/05/2011	110	78,3	431,2	1 268,0
...					

```
SQL> SELECT SOCIETE, DATE_COMMANDE, NO_EMPLOYE E, PORT P,
2 SUM(PORT) OVER ( PARTITION BY NO_EMPLOYE ORDER BY PORT DESC) SCE,
3 SUM(PORT) OVER ( PARTITION BY CODE_CLIENT ORDER BY DATE_COMMANDE) SCC
4 FROM CLIENTS NATURAL JOIN COMMANDES
5 WHERE ANNEE = 2011 AND MOIS = 5 AND PAYS = 'France';
```

SOCIETE	DATE_COMMA	E	P	SCE	SCC
Victuailles en stock	01/05/2011	29	95,6	95,6	95,6
Victuailles en stock	31/05/2011	29	95,3	190,9	1 258,9
La maison d'Asie	16/05/2011	29	93,5	284,4	527,7
Folies gourmandes	07/05/2011	29	92,0	376,4	546,9
Folies gourmandes	25/05/2011	29	90,7	467,1	1 264,8
...					

Attention : l'ordre de tri défini dans les fonctions analytiques doivent être distinctes, sinon pour tout un ensemble de valeurs non distinctes la fonction est calculée comme un seul enregistrement.



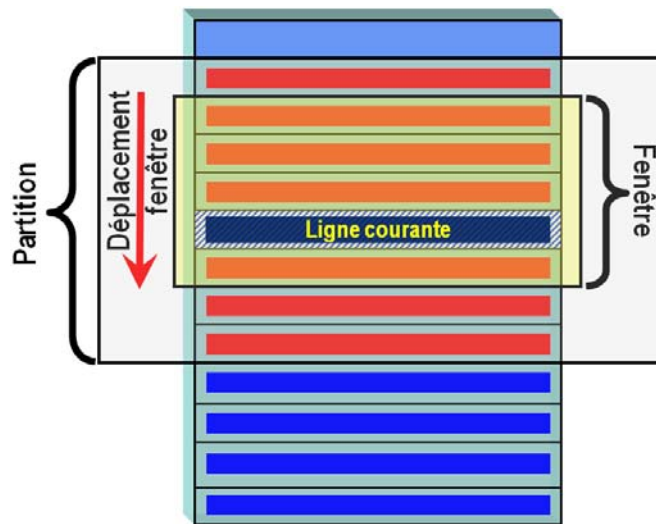
```
SQL >SELECT ANNEE, MOIS, NO_EMPLOYE E, SUM(PORT) P,
2 SUM(SUM(PORT)) OVER ( PARTITION BY ANNEE ORDER BY MOIS) SCA1,
3 SUM(SUM(PORT)) OVER ( PARTITION BY ANNEE
4 ORDER BY MOIS, NO_EMPLOYE) SCA2
5 FROM CLIENTS NATURAL JOIN COMMANDES WHERE ANNEE = 2011
6 AND PAYS = 'France' GROUP BY ANNEE, MOIS, NO_EMPLOYE ORDER BY 1,2,3;
```

ANNEE	MOIS	E	P	SCA1	SCA2
2011	1	29	2322,5	7698,2	2322,5
2011	1	45	2002	7698,2	4324,5
2011	1	110	1684,1	7698,2	6008,6
2011	1	111	1689,6	7698,2	7698,2
2011	2	29	2398,2	16446,1	10096,4
2011	2	45	2062	16446,1	12158,4
...					

## La clause de fenêtrage

La clause de fenêtrage vous permet d'analyser des données en calculant des valeurs agrégées sur les fenêtres entourant chaque ligne.

Pour chaque ligne dans une partition, on peut définir une fenêtre non fixe de données. Cette fenêtre détermine l'ensemble des lignes utilisées pour exécuter le calcul pour la ligne courante. La taille de la fenêtre peut être basée sur un nombre physique de lignes ou un intervalle (INTERVAL N DAY/MONTH/YEAR). La fenêtre possède une ligne de commencement et de fin.



La syntaxe de la clause est :

```
{ ROWS | RANGE }
{ BETWEEN
  {UNBOUNDED PRECEDING|CURRENT ROW|EXPRESSION {PRECEDING|FOLLOWING}}
AND
  {UNBOUNDED PRECEDING|CURRENT ROW|EXPRESSION {PRECEDING|FOLLOWING}}
|
  {UNBOUNDED PRECEDING|CURRENT ROW|EXPRESSION PRECEDING} }
```

<b>ROWS</b>	La taille de la fenêtre est définie par rapport à une constante ou une expression numérique.
<b>RANGE</b>	La taille de la fenêtre est définie par rapport à une expression de type « <b>INTERVAL</b> ».
<b>UNBOUNDED</b>	La limite de la fenêtre est la même que celle de la partition et s'il n'y a pas de clause « <b>PARTITION BY</b> » alors la limite est définie par la requête.
<b>CURRENT ROW</b>	La ligne courante.

La clause de fenêtrage est implicite pour les fonctions analytiques qui utilisent la clause « **ORDER BY** » car le calcul cumulatif c'est une fenêtre qui n'a aucune limite précédente et la limite inférieure est définie par la ligne courante.



```
SQL> SELECT DATE_COMMANDE, SUM(PORT) SP,
2 SUM(SUM(PORT)) OVER ( ORDER BY DATE_COMMANDE ) SC1,
3 SUM(SUM(PORT))OVER(ORDER BY DATE_COMMANDE ROWS UNBOUNDED PRECEDING)SC2,
4 SUM(SUM(PORT)) OVER ( ) ST FROM CLIENTS NATURAL JOIN COMMANDES
5 WHERE ANNEE = 2011 AND MOIS = 5 GROUP BY DATE_COMMANDE;
```

DATE_COMMA	SP	SC1	SC2	ST
01/05/2011	1 665,1	1 665,1	1 665,1	67 417,3
02/05/2011	1 838,3	3 503,4	3 503,4	67 417,3
03/05/2011	1 519,5	5 022,9	5 022,9	67 417,3
04/05/2011	1 502,4	6 525,3	6 525,3	67 417,3
...				
30/05/2011	2 111,4	64 812,8	64 812,8	67 417,3
31/05/2011	2 604,5	67 417,3	<b>67 417,3</b>	<b>67 417,3</b>

La requête suivante montre le calcul de la somme des frais de port par jour pour le mois de mai 2011, ainsi que la moyenne des frais de port calculée avec une fenêtre de trois jours, une journée qui précède la date du jour et le jour suivant. Il faut remarquer que le premier et le dernier enregistrement ne peuvent pas respecter cette règle car pour le premier enregistrement il n'y a pas de jour précédent et respectivement pour le dernier il n'y a pas de jour suivant.



```
SQL> SELECT DATE_COMMANDE, SUM(PORT) SP, AVG(SUM(PORT)) OVER (
2     ORDER BY DATE_COMMANDE ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING ) AP
3 FROM CLIENTS NATURAL JOIN COMMANDES
4 WHERE ANNEE = 2011 AND MOIS = 5 GROUP BY DATE_COMMANDE;
```

DATE_COMMA	SP	AP
01/05/2011	1 665,1	1 751,70
02/05/2011	1 838,3	1 674,30
03/05/2011	1 519,5	1 620,07
04/05/2011	1 502,4	1 630,77
...		
28/05/2011	2 008,8	2 154,27
29/05/2011	2 317,5	2 145,90
30/05/2011	2 111,4	2 344,47
31/05/2011	2 604,5	2 357,95

<= ( 1665,1 + 1838,3 + 1519,5 ) / 2

La requête suivante permet de calculer la moyenne des frais de port entre l'enregistrement précédent et l'enregistrement courant pour le client 'Folies gourmandes'.



```
SQL> SELECT MOIS, SUM(PORT) SP,
2     AVG(SUM(PORT)) OVER ( ORDER BY MOIS
3     ROWS BETWEEN 1 PRECEDING AND CURRENT ROW ) AP,
4     AVG(SUM(PORT)) OVER ( ORDER BY MOIS
5     ROWS 1 PRECEDING) AP
6 FROM CLIENTS NATURAL JOIN COMMANDES WHERE SOCIETE='Folies gourmandes'
7 GROUP BY MOIS;
```

MOIS	SP	AP	AP
1	2 717,3	2717,3	2717,3
2	803,3	1760,3	1760,3
3	789,5	796,4	796,4
4	1 331,2	1060,35	1060,35
...			

<= ( 2717,3 + 803,3 ) / 2

La requête suivante permet de calculer l'augmentation des frais de port par rapport au mois précédent pour le client 'Folies gourmandes'. Ainsi il faut d'abord retrouver la valeur de la somme des frais de port pour le mois précédent en utilisant la fonction analytique de moyenne avec une fenêtre comportant que l'enregistrement précédent.



```
SQL> SELECT MOIS, SUM(PORT) SP,
2     AVG(SUM(PORT)) OVER ( ORDER BY MOIS
3     ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING ) AP,
4     SUM(PORT) - NVL( AVG(SUM(PORT)) OVER ( ORDER BY MOIS
5     ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING ), 0) A
6 FROM CLIENTS NATURAL JOIN COMMANDES WHERE SOCIETE='Folies gourmandes'
7 GROUP BY MOIS;
```

MOIS	SP	AP	A
1	2 717,3		2 717,30
2	803,3	2717,3	-1 914,00

```

3      789,5      803,3      -13,80
4  1 331,2      789,5      541,70
...

```

L'exemple suivant permet d'afficher la somme des frais de port par année et mois, mais en même temps la valeur maximum et minimum dans l'année ainsi que le nombre d'enregistrements dans chaque année (en l'occurrence, ici l'année est le partitionnement de calcul pour les fonctions analytiques).



```

SQL> SELECT ANNEE, MOIS, SUM(PORT) SP,
2      MAX(SUM(PORT)) OVER( PARTITION BY ANNEE ORDER BY MOIS
3      ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) MAXP,
4      MIN(SUM(PORT)) OVER( PARTITION BY ANNEE ORDER BY MOIS
5      ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) MAXP,
6      COUNT(*) OVER ( PARTITION BY ANNEE ORDER BY MOIS
7      ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING ) COUNTP
8  FROM CLIENTS NATURAL JOIN COMMANDES GROUP BY ANNEE, MOIS;

```

ANNEE	MOIS	SP	MAXP	MAXP	COUNTP
2010	1	49 332,80	55 018,80	46 707,10	12
2010	2	46 707,10	55 018,80	46 707,10	12
2010	3	49 768,40	55 018,80	46 707,10	12
2010	4	46 904,00	55 018,80	46 707,10	12
2010	5	50 921,50	55 018,80	46 707,10	12
2010	6	49 397,90	55 018,80	46 707,10	12
2010	7	49 321,80	55 018,80	46 707,10	12
2010	8	53 680,40	55 018,80	46 707,10	12
2010	9	47 753,60	55 018,80	46 707,10	12
2010	10	55 018,80	55 018,80	46 707,10	12
2010	11	49 608,20	55 018,80	46 707,10	12
2010	12	47 898,20	55 018,80	46 707,10	12
2011	1	65 722,30	73 703,40	65 722,30	6
2011	2	66 132,70	73 703,40	65 722,30	6
2011	3	73 703,40	73 703,40	65 722,30	6
2011	4	68 543,40	73 703,40	65 722,30	6
2011	5	67 417,30	73 703,40	65 722,30	6
2011	6	69 621,70	73 703,40	65 722,30	6

ANNEE	MOIS	SP	MAXP	MAXP	COUNTP
2010	1	49 332,80	55 018,80	46 707,10	12
2010	2	46 707,10	55 018,80	46 707,10	12
2010	3	49 768,40	55 018,80	46 707,10	12
2010	4	46 904,00	55 018,80	46 707,10	12
2010	5	50 921,50	55 018,80	46 707,10	12
2010	6	49 397,90	55 018,80	46 707,10	12
2010	7	49 321,80	55 018,80	46 707,10	12
2010	8	53 680,40	55 018,80	46 707,10	12
2010	9	47 753,60	55 018,80	46 707,10	12
2010	10	55 018,80	55 018,80	46 707,10	12
2010	11	49 608,20	55 018,80	46 707,10	12
2010	12	47 898,20	55 018,80	46 707,10	12
2011	1	65 722,30	73 703,40	65 722,30	6
2011	2	66 132,70	73 703,40	65 722,30	6
2011	3	73 703,40	73 703,40	65 722,30	6
2011	4	68 543,40	73 703,40	65 722,30	6
2011	5	67 417,30	73 703,40	65 722,30	6
2011	6	69 621,70	73 703,40	65 722,30	6

Vous avez la possibilité, pour les fonctions analytiques, d'utiliser une fenêtre définie par rapport aux enregistrements de la requête ou par rapport à un intervalle de temps.

L'exemple suivant permet de calculer la moyenne glissante sur cinq jours pour le mois de février 2011. Pour avoir le calcul cohérent dans la requête, on reprend deux jour du mois de janvier et deux du mois d'avril.



```

SQL> SELECT DATE_COMMANDE, SUM(PORT) SP, AVG(SUM(PORT)) OVER
2  ( ORDER BY DATE_COMMANDE
3    RANGE BETWEEN INTERVAL '2' DAY PRECEDING AND
4    INTERVAL '2' DAY FOLLOWING ) AP_RANGE
5  FROM CLIENTS NATURAL JOIN COMMANDES
6  WHERE DATE_COMMANDE BETWEEN '30/01/2011' AND '02/03/2011'
7  GROUP BY DATE_COMMANDE;

```

DATE_COMMA	SP	AP_RANGE
30/01/2011	2 583,0	2 644,67
31/01/2011	2 520,2	2 555,63
01/02/2011	2 830,8	2 576,68

```
02/02/2011  2 288,5    2 398,88
03/02/2011  2 660,9    2 333,54
04/02/2011  1 694,0    2 096,42
...
28/02/2011  3 135,9    2 683,28
01/03/2011  2 148,5    2 561,50
02/03/2011  2 102,8    2 462,40
```

En utilisant la requête précédente comme une sous-requête, vous pouvez filtrer les enregistrements parasites mais toutefois nécessaires aux calculs.



```
SQL> SELECT * FROM ( SELECT DATE_COMMANDE, SUM(PORT) SP,
2          AVG(SUM(PORT)) OVER ( ORDER BY DATE_COMMANDE
3          RANGE BETWEEN INTERVAL '2' DAY PRECEDING AND
4          INTERVAL '2' DAY FOLLOWING ) AP_RANGE
5          FROM CLIENTS NATURAL JOIN COMMANDES
6          WHERE DATE_COMMANDE BETWEEN '30/01/2011' AND '02/03/2011'
7          GROUP BY DATE_COMMANDE )
8  WHERE DATE_COMMANDE BETWEEN '01/02/2011' AND '28/02/2011';
```

```
DATE_COMMA      SP      AP_RANGE
-----
01/02/2011    2 830,8    2 576,68
02/02/2011    2 288,5    2 398,88
...
27/02/2011    2 858,8    2 829,08
28/02/2011    3 135,9    2 683,28
```

Il faut faire attention avec la syntaxe de fenêtre logique car il peut y avoir une mauvaise interprétation des résultats. L'exemple suivant permet d'apprécier la différence entre les deux syntaxes.



```
SQL> SELECT DATE_COMMANDE, SUM(PORT) SP,
2          AVG(SUM(PORT)) OVER ( ORDER BY DATE_COMMANDE
3          ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING ) AP_ROWS,
4          AVG(SUM(PORT)) OVER ( ORDER BY DATE_COMMANDE
5          RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
6          INTERVAL '1' DAY FOLLOWING ) AP_RANGE
7          FROM CLIENTS NATURAL JOIN COMMANDES
8          WHERE ANNEE = 2011 AND MOIS = 5 AND SOCIETE = 'Folies gourmandes'
9          GROUP BY DATE_COMMANDE;
```

```
DATE_COMMA      SP      AP_ROWS      AP_RANGE
-----
...
09/05/2011      83,6      73,30      73,30
10/05/2011      57,8      80,43      80,43
11/05/2011      99,9      75,53      78,85      <= ( 99,9 + 57,8 ) / 2
21/05/2011      68,9      76,97      65,50      <= ( 68,9 + 62,1 ) / 2
22/05/2011      62,1      71,93      71,93
23/05/2011      84,8      79,50      79,50
...
```

### Attention



Attention, l'utilisation de la syntaxe de fenêtre logique pour les fonctions analytiques implique l'existence des toutes les valeurs séquentielles pour les intervalles de date précises. Ainsi les résultats restent cohérents avec les conditions de regroupement des fenêtres logiques demandées.

Pour contourner cet inconvénient, vous pouvez utiliser une jointure externe avec une table qui contient la liste des dates souhaitées.

La table DIM\_TEMPES est une table conçue spécialement pour être utilisée dans les jointures externes.



```
SQL> SELECT MIN(JOUR), MAX(JOUR) COUNT(*) FROM DIM_TEMPES;
2 WHERE ANNEE = 2011 AND MOIS_N = 5;
```

MIN(JOUR)	MAX(JOUR)	COUNT(*)
01/05/2011	31/05/2011	31

```
SQL> SELECT DT.JOUR, NVL(TO_CHAR(SUM(PORT),'99D0'),'----') SP,
2      AVG(NVL(SUM(PORT),0)) OVER ( ORDER BY DT.JOUR
3      RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
4      INTERVAL '1' DAY FOLLOWING ) AP_RANGE,
5      AVG(SUM(PORT)) OVER ( ORDER BY DT.JOUR
6      RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
7      INTERVAL '1' DAY FOLLOWING ) AP_RANGE
8 FROM CLIENTS CL JOIN COMMANDES CO
9 ON ( CL.CODE_CLIENT = CO.CODE_CLIENT
10      AND CO.ANNEE = 2011 AND CO.MOIS = 5
11      AND CL.SOCIETE = 'Folies gourmandes')
12 RIGHT OUTER JOIN
13 ( SELECT * FROM DIM_TEMPES
14      WHERE ANNEE = 2011 AND MOIS_N = 5 ) DT
15 ON ( DT.JOUR = CO.DATE_COMMANDE)
16 GROUP BY DT.JOUR;
```

JOUR	SP	AP_RANGE	AP_RANGE
01/05/2011	95,0	96,95	96,95
02/05/2011	98,9	90,37	90,37
03/05/2011	77,2	77,73	77,73
04/05/2011	57,1	64,80	64,80
05/05/2011	60,1	61,27	61,27
06/05/2011	66,6	72,90	72,90
07/05/2011	92,0	79,03	79,03
08/05/2011	78,5	84,70	84,70
09/05/2011	83,6	73,30	73,30
10/05/2011	57,8	80,43	80,43
11/05/2011	99,9	52,57	78,85
12/05/2011	-----	33,30	99,90
13/05/2011	-----	0,00	
...			

Annotations for rows 11 and 12:

- Row 11:  $(57,8 + 99,9 + 0) / 3$
- Row 12:  $(0 + 99,9 + 0) / 3$
- Row 11:  $\leq (57,8 + 99,9 + \text{NULL}) / 2$
- Row 12:  $\leq (99,9 + \text{NULL} + \text{NULL}) / 1$

Dans le cadre d'une jointure externe il faut faire attention au traitement des valeurs « **NULL** » car si vous utilisez des fonctions de type « **NVL** » les calculs peuvent être erronés, (voir l'exemple précédent).



```
SQL> SELECT CL.SOCIETE, DT.JOUR, SUM(PORT) SP, AVG(SUM(PORT)) OVER (
2      PARTITION BY CL.SOCIETE ORDER BY DT.JOUR NULLS FIRST
3      RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
4      INTERVAL '1' DAY FOLLOWING ) AP_RANGE
5 FROM CLIENTS CL JOIN COMMANDES CO
6 ON ( CL.CODE_CLIENT = CO.CODE_CLIENT
7      AND CO.ANNEE = 2011 AND CO.MOIS = 5
8      AND CL.SOCIETE IN ('Folies gourmandes',
9                          'France restauration'))
10 RIGHT OUTER JOIN
```

```

11      ( SELECT * FROM DIM_TEMPS
12        WHERE ANNEE = 2011 AND MOIS_N = 5 ) DT
13      ON ( DT.JOUR = CO.DATE_COMMANDE)
14  GROUP BY CL.SOCIETE, DT.JOUR;

```

SOCIETE	JOUR	SP	AP_RANGE
...			
Folies gourmandes	07/05/2011	92,00	79,03
Folies gourmandes	08/05/2011	78,50	84,70
Folies gourmandes	09/05/2011	83,60	73,30
Folies gourmandes	10/05/2011	57,80	80,43
Folies gourmandes	11/05/2011	99,90	78,85
Folies gourmandes	21/05/2011	68,90	65,50
Folies gourmandes	22/05/2011	62,10	71,93
Folies gourmandes	23/05/2011	84,80	79,50
Folies gourmandes	24/05/2011	91,60	89,03
Folies gourmandes	25/05/2011	90,70	91,15
France restauration	08/05/2011	97,50	86,80
France restauration	09/05/2011	76,10	86,40
France restauration	10/05/2011	85,60	77,20
France restauration	11/05/2011	69,90	77,75
France restauration	17/05/2011	63,50	77,00
France restauration	18/05/2011	90,50	68,30
France restauration	19/05/2011	50,90	65,70
France restauration	20/05/2011	55,70	53,30
	12/05/2011		
	13/05/2011		
...			

Comme vous pouvez voir dans l'exemple précédent, cette démarche est valable uniquement si vous utilisez un seul critère de regroupement. Pour plusieurs critères, vous devez d'abord créer une requête qui affiche toutes les combinaisons des critères pour l'utiliser comme base pour une jointure externe.



```

SQL> SELECT CL.SOCIETE, DT.JOUR, SUM(PORT) SP, AVG(SUM(PORT)) OVER (
2      PARTITION BY DT.SOCIETE ORDER BY DT.JOUR
3      RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
4      INTERVAL '1' DAY FOLLOWING ) AP_RANGE
5  FROM CLIENTS CL JOIN COMMANDES CO
6      ON ( CL.CODE_CLIENT = CO.CODE_CLIENT
7          AND CO.ANNEE = 2011 AND CO.MOIS = 5
8          AND CL.SOCIETE IN ('Folies gourmandes',
9                             'France restauration'))
10 RIGHT OUTER JOIN
11      ( SELECT DT2.JOUR JOUR, CL2.SOCIETE SOCIETE FROM
12        ( SELECT SOCIETE FROM CLIENTS
13          WHERE SOCIETE IN ('Folies gourmandes',
14                             'France restauration'))CL2
15        CROSS JOIN
16        ( SELECT JOUR FROM DIM_TEMPS
17          WHERE ANNEE = 2011 AND MOIS_N = 5 )DT2) DT
18      ON ( DT.JOUR = CO.DATE_COMMANDE)
19  GROUP BY DT.SOCIETE, DT.JOUR;

```

SOCIETE	JOUR	SP	AP_RANGE
...			
Folies gourmandes	01/05/2011	95,00	96,95

Folies gourmandes	02/05/2011	98,90	90,37
Folies gourmandes	03/05/2011	77,20	77,73
Folies gourmandes	04/05/2011	57,10	64,80
Folies gourmandes	05/05/2011	60,10	61,27
Folies gourmandes	06/05/2011	66,60	72,90
Folies gourmandes	07/05/2011	92,00	111,53
Folies gourmandes	08/05/2011	176,00	142,57
Folies gourmandes	09/05/2011	159,70	159,70
Folies gourmandes	10/05/2011	143,40	157,63
Folies gourmandes	11/05/2011	169,80	156,60
Folies gourmandes	12/05/2011		169,80
Folies gourmandes	13/05/2011		
Folies gourmandes	14/05/2011		
Folies gourmandes	15/05/2011		
Folies gourmandes	16/05/2011		63,50
Folies gourmandes	17/05/2011	63,50	77,00
...			

Une telle démarche peut s'avérer très vite difficile à mettre en œuvre du point de vue de l'écriture, mais surtout du point de vue des performances. Il est préférable d'utiliser les jointures externes avec la clause « **PARTITION BY** » traitée dans le chapitre suivant.

Les trois clauses de partitionnement, d'ordre et de fenêtrage sont facultatives en général, mais souvent nécessaires suivant la fonction utilisée.

## L'opérateur OUTER JOIN

L'opérateur « **OUTER JOIN ON** » effectue une jointure externe entre deux ensembles d'enregistrements ; il permet également de tenir compte du regroupement du premier ensemble en se servant des conditions spécifiées dans la clause « **PARTITION BY** » en respectant la syntaxe suivante :

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
FROM ( SOUS-REQUÊTE )
      [PARTITION BY(EXPRESSION[,...])][{LEFT|RIGHT|FULL} OUTER JOIN
      {NOM_TABLE2|(SOUS-REQUÊTE)}
      { ON ( NOM_TABLE1.COLONNE = NOM_TABLE2.COLONNE [,...]) ]
      | USING (NOM_COLONNE1[,...]) ] };
```

Dans la requête suivante, la sous-requête est déclarée à l'aide de la clause « **WITH** » ; ainsi par la suite on utilise uniquement l'alias pour la jointure.



```
SQL> WITH VENTES AS (
2      SELECT CL.SOCIETE, CO.DATE_COMMANDE, SUM(CO.PORT) SP
3      FROM CLIENTS CL JOIN COMMANDES CO
4      ON ( CL.CODE_CLIENT = CO.CODE_CLIENT
5          AND CO.DATE_COMMANDE BETWEEN '10/05/2011'
6          AND '15/05/2011'
7          AND CL.SOCIETE IN ('Folies gourmandes',
8                          'France restauration'))
9      GROUP BY CL.SOCIETE, CO.DATE_COMMANDE)
10 SELECT SOCIETE,JOUR,SP,
11      AVG(SP) OVER (PARTITION BY SOCIETE ORDER BY JOUR
12      RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
```



```

13          INTERVAL '1' DAY FOLLOWING ) AP_RANGE
14 FROM VENTES V
15     PARTITION BY (SOCIETE) RIGHT OUTER JOIN
16     ( SELECT JOUR FROM DIM_TEMPS
17       WHERE JOUR BETWEEN '10/05/2011' AND '15/05/2011') T
18     ON ( V.DATE_COMMANDE = T.JOUR);

```

SOCIETE	JOUR	SP	AP_RANGE
Folies gourmandes	10/05/2011	57,80	78,85
Folies gourmandes	11/05/2011	99,90	78,85
Folies gourmandes	12/05/2011		99,90
Folies gourmandes	13/05/2011		
Folies gourmandes	14/05/2011		
Folies gourmandes	15/05/2011		
France restauration	10/05/2011	85,60	77,75
France restauration	11/05/2011	69,90	77,75
France restauration	12/05/2011		69,90
France restauration	13/05/2011		
France restauration	14/05/2011		
France restauration	15/05/2011		

Si vous utilisez plusieurs critères de regroupement, tous doivent être spécifiés comme arguments pour « **PARTITION BY** ». Dans l'exemple suivant, vous pouvez voir les deux critères de regroupement PAYS et SOCIETE spécifiés dans la clause, et la DATE\_COMMANDE est utilisée pour la jointure externe avec la sous-requête sur la table DIM\_TEMPS.



```

SQL> WITH VENTES AS (
2     SELECT CL.PAYS,CL.SOCIETE,CO.DATE_COMMANDE,SUM(CO.PORT) SP
3     FROM CLIENTS CL JOIN COMMANDES CO
4          ON ( CL.CODE_CLIENT = CO.CODE_CLIENT
5              AND CO.DATE_COMMANDE BETWEEN
6                '01/05/2011' AND '10/05/2011'
7              AND CL.SOCIETE IN ('Morgenstern Gesundkost',
8                'QUICK-Stop','Folies gourmandes'))
9     GROUP BY CL.PAYS, CL.SOCIETE, CO.DATE_COMMANDE)
10 SELECT PAYS,SOCIETE,JOUR,SP,
11        SUM(SP) OVER ( PARTITION BY PAYS,SOCIETE ) SUM_PS,
12        SUM(SP) OVER ( PARTITION BY PAYS ) SUM_P
13 FROM VENTES V
14     PARTITION BY (PAYS,SOCIETE) RIGHT OUTER JOIN
15     ( SELECT JOUR FROM DIM_TEMPS
16       WHERE JOUR BETWEEN '01/05/2011' AND '10/05/2011') T
17     ON ( V.DATE_COMMANDE = T.JOUR)
18 ORDER BY PAYS,SOCIETE,JOUR;

```

PAYS	SOCIETE	JOUR	SP	SUM_PS	SUM_P
Allemagne	Morgenstern Gesundkost	01/05/2011	64,90	769,10	1 278,80
Allemagne	Morgenstern Gesundkost	02/05/2011	87,80	769,10	1 278,80
Allemagne	Morgenstern Gesundkost	03/05/2011	87,30	769,10	1 278,80
Allemagne	Morgenstern Gesundkost	04/05/2011	51,20	769,10	1 278,80
Allemagne	Morgenstern Gesundkost	05/05/2011	95,20	769,10	1 278,80
Allemagne	Morgenstern Gesundkost	06/05/2011	65,90	769,10	1 278,80
Allemagne	Morgenstern Gesundkost	07/05/2011	94,70	769,10	1 278,80
Allemagne	Morgenstern Gesundkost	08/05/2011	71,80	769,10	1 278,80
Allemagne	Morgenstern Gesundkost	09/05/2011	54,30	769,10	1 278,80

Allemagne	Morgenstern	Gesundkost	10/05/2011	96,00	769,10	1	278,80
Allemagne	QUICK-Stop		01/05/2011	75,80	509,70	1	278,80
Allemagne	QUICK-Stop		02/05/2011	94,60	509,70	1	278,80
Allemagne	QUICK-Stop		03/05/2011		509,70	1	278,80
Allemagne	QUICK-Stop		04/05/2011	58,50	509,70	1	278,80
Allemagne	QUICK-Stop		05/05/2011	99,30	509,70	1	278,80
Allemagne	QUICK-Stop		06/05/2011	99,50	509,70	1	278,80
Allemagne	QUICK-Stop		07/05/2011	82,00	509,70	1	278,80
Allemagne	QUICK-Stop		08/05/2011		509,70	1	278,80
Allemagne	QUICK-Stop		09/05/2011		509,70	1	278,80
Allemagne	QUICK-Stop		10/05/2011		509,70	1	278,80
France	Folies gourmandes		01/05/2011	95,00	766,80		766,80
France	Folies gourmandes		02/05/2011	98,90	766,80		766,80
France	Folies gourmandes		03/05/2011	77,20	766,80		766,80
France	Folies gourmandes		04/05/2011	57,10	766,80		766,80
...							

Dans l'exemple suivant deux autres critères PAYS et NO\_EMPLOYEES sont précisés dans la clause « **PARTITION BY** ». Les enregistrements sont utilisés pour effectuer une moyenne sur trois jours glissants AVSP, l'affichage de la somme des frais de port pour le jour précédent SPJP, et différence entre la somme du jour précédent et la somme du jour DSP.



```
SQL> WITH VENTES AS (
  2  SELECT CL.PAYS,CO.NO_EMPLOYE,CO.DATE_COMMANDE,SUM(CO.PORT) SP
  3      FROM CLIENTS CL JOIN COMMANDES CO
  4          ON ( CL.CODE_CLIENT = CO.CODE_CLIENT
  5              AND CO.DATE_COMMANDE BETWEEN
  6                  '01/05/2011' AND '10/05/2011'
  7              AND CL.PAYS = 'Allemagne'
  8              AND CO.NO_EMPLOYE IN ( 85,102,110,111 ))
  9  GROUP BY CL.PAYS,NO_EMPLOYE,CO.DATE_COMMANDE)
 10  SELECT PAYS,NO_EMPLOYE E,JOUR,SP,
 11      AVG(SP) OVER ( PARTITION BY NO_EMPLOYE ORDER BY JOUR
 12                      RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
 13                          INTERVAL '1' DAY FOLLOWING ) AVSP,
 14      AVG(SP) OVER ( PARTITION BY NO_EMPLOYE ORDER BY JOUR
 15                      RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
 16                          INTERVAL '1' DAY PRECEDING ) SPJP ,
 17      SP - AVG(SP) OVER ( PARTITION BY NO_EMPLOYE ORDER BY JOUR
 18                          RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
 19                              INTERVAL '1' DAY PRECEDING ) DSP
 20  FROM VENTES V
 21      PARTITION BY (PAYS,NO_EMPLOYE) RIGHT OUTER JOIN
 22      ( SELECT JOUR FROM DIM_TEMPS
 23          WHERE JOUR BETWEEN '01/05/2011' AND '10/05/2011') T
 24      ON ( V.DATE_COMMANDE = T.JOUR)
 25  ORDER BY PAYS,NO_EMPLOYE,JOUR;
```

PAYS	E	JOUR	SP	AVSP	SPJP	DSP
-----						
Allemagne	85	01/05/2011	122,10	122,10		
Allemagne	85	02/05/2011		104,70	122,10	
Allemagne	85	03/05/2011	87,30	69,25		
Allemagne	85	04/05/2011	51,20	69,25	87,30	-36,10
Allemagne	85	05/05/2011		63,95	51,20	
Allemagne	85	06/05/2011	76,70	68,70		
Allemagne	85	07/05/2011	60,70	65,43	76,70	-16,00

Allemagne	85	08/05/2011	58,90	67,47	60,70	-1,80
Allemagne	85	09/05/2011	82,80	79,10	58,90	23,90
Allemagne	85	10/05/2011	95,60	89,20	82,80	12,80
Allemagne	102	01/05/2011	75,80	129,10		
Allemagne	102	02/05/2011	182,40	129,10	75,80	106,60
Allemagne	102	03/05/2011		182,40	182,40	
Allemagne	102	04/05/2011		95,20		
Allemagne	102	05/05/2011	95,20	95,20		
Allemagne	102	06/05/2011		94,95	95,20	
Allemagne	102	07/05/2011	94,70	86,15		
Allemagne	102	08/05/2011	77,60	83,80	94,70	-17,10
Allemagne	102	09/05/2011	79,10	74,53	77,60	1,50
Allemagne	102	10/05/2011	66,90	73,00	79,10	-12,20

## Fonctions de classement

Les fonctions de classement permettent de calculer une valeur de classement pour chaque ligne en fonction d'un ordre spécifié dans la requête.

### ROW\_NUMBER

La fonction « **ROW\_NUMBER** » affecte à chaque ligne un numéro unique, déterminé par la clause « **ORDER BY** ». C'est le numéro d'ordre d'un enregistrement suivant l'ordre de tri spécifié et il est réinitialisé pour chaque partition que vous avez choisi de définir.



```
SQL> SELECT CODE_CATEGORIE CAT,NO_FOURNISSEUR NF,UNITES_STOCK US,
2          ROW_NUMBER() OVER ( PARTITION BY CODE_CATEGORIE
3                                ORDER BY UNITES_STOCK)          RN1,
4          ROW_NUMBER() OVER ( PARTITION BY CODE_CATEGORIE
5                                ORDER BY UNITES_STOCK NULLS FIRST) RN2,
6          ROW_NUMBER()OVER (
7                                ORDER BY CODE_CATEGORIE, UNITES_STOCK)          RN3,
8          ROW_NUMBER() OVER ( ORDER BY CODE_CATEGORIE,
9                                NO_FOURNISSEUR,UNITES_STOCK)          RN4,
10         ROW_NUMBER() OVER ( ORDER BY NO_FOURNISSEUR,
11                                UNITES_STOCK )          RN5
12 FROM PRODUITS WHERE CODE_CATEGORIE < 3;
```

CAT	NF	US	RN1	RN2	RN3	RN4	RN5
1	7	15	1	2	1	3	13
1	1	17	2	3	2	1	2
1	20	17	3	5	3	15	33
1	18	17	4	4	4	12	30
1	16	20	5	6	5	8	23
1	1	39	6	7	6	2	3
1	14	50	7	8	7	6	21
1	16	52	8	9	8	9	24
1	23	57	9	10	9	16	35
1	18	60	10	11	10	13	31
1	18	69	11	12	11	14	32
1	16	100	12	13	12	10	27
1	16	111	13	14	13	11	28
1	12	125	14	16	14	5	17

1	14	125	15	15	15	7	22
1	10		16	1	16	4	15
2	2	0	1	1	17	18	4
2	2	4	2	2	18	19	5
2	3	6	3	3	19	22	8
2	1	13	4	4	20	17	1
2	7	24	5	5	21	27	14
2	13	25	6	6	22	29	18
2	20	27	7	7	23	35	34
2	12	32	8	8	24	28	16
2	6	39	9	9	25	25	11
2	6	40	10	11	26	26	12
...							

Dans l'exemple précédent, vous pouvez remarquer pour les colonnes RN1 et RN2 le partitionnement suivant la colonne CODE\_CATEGORIE. Ainsi pour chaque changement de catégorie de produits, la numérotation recommence. La colonne RN2 commence par la valeur 2 et la valeur 1 est la dernière dans la partition car l'ordre de tri demandé est « **NULLS FIRST** ». La colonne RN3 n'a pas de partition et ainsi la numérotation est une séquence continue du début à la fin suivant l'ordre tri de cette fonction. Les deux derniers ordres de tri sont affichés dans les colonnes RN4 et RN5, leur logique étant moins aisée à suivre. Elles sont cependant très utiles pour effectuer des tris dans des outils client, par exemple SQLDeveloper.

The screenshot shows the SQL Developer interface. The Query Builder window contains the following SQL query:

```
SELECT CODE_CATEGORIE CAT, NO_FOURNISSEUR NF, UNITES_STOCK US,
ROW_NUMBER() OVER ( PARTITION BY CODE_CATEGORIE
ORDER BY UNITES_STOCK) RN1,
ROW_NUMBER() OVER ( PARTITION BY CODE_CATEGORIE
ORDER BY UNITES_STOCK NULLS FIRST) RN2,
ROW_NUMBER() OVER (
ORDER BY CODE_CATEGORIE, UNITES_STOCK) RN3,
ROW_NUMBER() OVER ( ORDER BY CODE_CATEGORIE,
NO_FOURNISSEUR, UNITES_STOCK) RN4,
ROW_NUMBER() OVER ( ORDER BY NO_FOURNISSEUR,
UNITES_STOCK ) RN5
FROM PRODUITS WHERE CODE_CATEGORIE < 3;
```

The Results window shows the output of the query. The columns are CAT, NF, US, RN1, RN2, RN3, RN4, and RN5. The data is as follows:

	CAT	NF	US	RN1	RN2	RN3	RN4	RN5
1	1	1	17	2	3	2	1	2
2	1	1	39	6	7	6	2	3
3	1	7	15	1	2	1	3	13
4	1	10 (null)	16	1	16	4	15	
5	1	12	125	14	16	14	5	17
6	1	14	50	7	8	7	6	21
7	1	14	125	15	15	15	7	22



### Conseil

Dans les requêtes récapitulatives qui utilisent les fonctions analytiques on a souvent besoin de calculs complexes avec des partitionnements différents et des ordres de tri complètement distincts. Ainsi il est difficile de contrôler les calculs pendant la conception des requêtes ou de trouver facilement les critères pour faire les tris une fois que la requête a été terminée.

La plupart des interfaces de développement fournissent des outils semblables au SQLDeveloper pourvu que vous ayez pris soin des calculer ces colonnes.

Il faut être attentif, car pour ces colonnes vous ne devez pas utiliser le partitionnement, mais il faut impérativement introduire dans l'ordre de tri toutes les colonnes déterminées par le tri. Il ne faut pas oublier que la clause « **PARTITION BY** » effectue un premier tri des enregistrements de la requête.



```
SQL> SELECT PAYS, NO_EMPLOYE E, ANNEE A, SUM(PORT) P,
2          SUM(SUM(PORT)) OVER ( PARTITION BY PAYS, ANNEE
3                                ORDER BY SUM(PORT) DESC NULLS LAST) SP1,
4          SUM(SUM(PORT)) OVER ( PARTITION BY NO_EMPLOYE, ANNEE
5                                ORDER BY SUM(PORT) NULLS FIRST ) SP2,
6          ROW_NUMBER() OVER ( ORDER BY PAYS, ANNEE,
7                                SUM(PORT) DESC NULLS LAST) RN1,
8          ROW_NUMBER() OVER ( ORDER BY NO_EMPLOYE, ANNEE,
9                                SUM(PORT) NULLS FIRST) RN2
10 FROM CLIENTS NATURAL JOIN COMMANDES
11 GROUP BY PAYS, NO_EMPLOYE, ANNEE;
```

PAYS	E	A	P	SP1	SP2	RN1	RN2
Royaume-Uni	1	2010	9 566,2	51 081,7	9 566,2	151	1
Royaume-Uni	1	2011	6 976,9	21 386,0	6 976,9	154	2
Danemark	2	2010	3 863,9	12 211,3	3 863,9	59	3
Danemark	2	2011	3 153,7	6 927,6	3 153,7	61	4
Espagne	3	2010	6 735,3	21 481,8	6 735,3	65	5
Espagne	3	2011	5 051,3	15 669,2	5 051,3	70	6
Allemagne	4	2010	10 754,3	44 844,1	10 754,3	4	7
Allemagne	4	2011	5 805,3	50 056,8	5 805,3	14	8
Suède	5	2010	3 488,8	10 671,3	3 488,8	159	9
Suède	5	2011	2 335,0	7 096,7	2 335,0	163	10
Argentine	6	2010	4 393,4	9 315,1	4 393,4	16	11
Argentine	6	2011	1 946,5	12 540,0	1 946,5	24	12
Brésil	7	2010	18 805,6	37 723,0	18 805,6	42	13
Brésil	7	2011	11 733,4	37 885,3	11 733,4	46	14
Pologne	8	2010	1 046,6	5 014,1	1 046,6	134	15
Pologne	8	2011	456,5	3 075,6	456,5	138	16
Canada	9	2010	3 689,5	8 377,3	3 689,5	48	17
Canada	9	2011	3 029,3	9 911,5	3 029,3	54	18
...							

## RANK

La fonction « **RANK** » calcule la valeur d'un rang au sein d'un groupe de valeurs. Dans le cas de valeurs à égalité, la fonction « **RANK** » laisse un vide dans la séquence de classement.



```
SQL> SELECT ROW_NUMBER() OVER ( ORDER BY SALAIRE DESC) RN, NOM, SALAIRE, RS
2 FROM (SELECT NOM, SALAIRE, RANK () OVER ( ORDER BY SALAIRE DESC) RS
3 FROM EMPLOYES) WHERE RS <= 10;
```

RN	NOM	SALAIRE	RS
1	Giroux	150 000	1
2	Brasseur	147 000	2
3	Fuller	96 000	3
4	Leger	19 000	4
5	Splingart	16 000	5
6	Ragon	13 000	6
7	Buchanan	13 000	6
8	Chambaud	12 000	8
9	Cheutin	10 000	9
10	Belin	10 000	9

Dans cet exemple, on peut voir un jeu de résultats donnant le classement, dans l'ordre décroissant du salaire des employés. L'affichage est effectué uniquement pour un palmarès des dix premiers employés. Vous pouvez remarquer que les valeurs 10 et 7 n'existent pas, mais que les valeurs 6 et 9 sont affichées deux fois.

## DENSE\_RANK

La fonction « **DENSE\_RANK** » calcule la valeur d'un rang au sein d'un groupe de valeurs. Dans le cas de valeurs à égalité, la fonction « **DENSE\_RANK** » ne laisse aucun vide dans la séquence de classement.



```
SQL> SELECT ROW_NUMBER() OVER
2          ( ORDER BY UNITES_STOCK NULLS LAST) RN,
3          NOM_PRODUIT, NO_FOURNISSEUR NF, UNITES_STOCK US,
4          RANK() OVER ( ORDER BY UNITES_STOCK) R,
5          DENSE_RANK() OVER (ORDER BY UNITES_STOCK) DR,
6          DENSE_RANK() OVER (ORDER BY UNITES_STOCK
7                               NULLS FIRST) DRI
8  FROM PRODUITS WHERE CODE_CATEGORIE = 1
9  ORDER BY US NULLS LAST;
```

RN	NOM_PRODUIT	NF	US	R	DR	DRI
1	Outback Lager	7	15	1	1	2
2	Chang	1	17	2	2	3
3	Côte de Blaye	18	17	2	2	3
4	Ipoh Coffee	20	17	2	2	3
5	Steeleye Stout	16	20	5	3	4
6	Chai	1	39	6	4	5
7	Tea	14	50	7	5	6
8	Laughing Lumberjack Lager	16	52	8	6	7
9	Lakkalikööri	23	57	9	7	8
10	Beer	18	60	10	8	9
11	Chartreuse verte	18	69	11	9	10
12	Coffee	16	100	12	10	11
13	Sasquatch Ale	16	111	13	11	12
14	Rhönbräu Klosterbier	12	125	14	12	13
15	Green Tea	14	125	14	12	13
16	Guaraná Fantástica	10		16	13	1

Dans l'exemple précédent, un jeu de résultats donnant le classement des unités en stock par fournisseur est renvoyé. Bien que le jeu de résultats contienne 16 enregistrements, seuls 13 rangs sont énumérés en raison d'une égalité des quantités en stock pour les enregistrements (2, 3, 4) et (14, 15).

## PERCENT\_RANK

La fonction « **PERCENT\_RANK** » calcule la position en pourcentage d'une ligne renvoyée à partir d'une requête par rapport aux autres lignes renvoyées par la requête, comme défini par la clause « **ORDER BY** ». Cette fonction renvoie une valeur décimale comprise entre 0 et 1.



```
SQL> SELECT FONCTION, PAYS, SUM(SALAIRE) SS,
2          PERCENT_RANK() OVER ( PARTITION BY FONCTION
3                                ORDER BY SUM(SALAIRE)) PERCENT_RANK
4  FROM EMPLOYES GROUP BY FONCTION, PAYS;
```

FONCTION	PAYS	SS	PERCENT_RANK
Assistante commerciale		16 540	0,000000000
Chef des ventes		83 000	0,000000000

Président		150 000	0,000000000
Représentant(e)	États-Unis	23 100	0,000000000
Représentant(e)	Brésil	23 100	0,000000000
Représentant(e)	Autriche	25 600	0,100000000
Représentant(e)	Belgique	27 000	0,150000000
Représentant(e)	Danemark	27 500	0,200000000
Représentant(e)	Mexique	28 900	0,250000000
Représentant(e)	Italie	29 000	0,300000000
Représentant(e)	Finlande	29 800	0,350000000
Représentant(e)	Suède	31 300	0,400000000
Représentant(e)	Portugal	31 600	0,450000000
Représentant(e)	France	32 200	0,500000000
Représentant(e)	Pologne	32 400	0,550000000
Représentant(e)	Norvège	35 300	0,600000000
Représentant(e)	Canada	35 500	0,650000000
Représentant(e)	Irlande	36 400	0,700000000
Représentant(e)	Espagne	36 700	0,750000000
Représentant(e)	Suisse	36 700	0,750000000
Représentant(e)	Venezuela	37 800	0,850000000
Représentant(e)	Argentine	38 900	0,900000000
Représentant(e)	Royaume-Uni	42 900	0,950000000
Représentant(e)	Allemagne	51 200	1,000000000
Vice-Président		243 000	0,000000000

L'exemple précédent montre un jeu de résultats indiquant le classement des salaires par fonction et par pays. Les résultats sont classés dans l'ordre sous forme de pourcentage décimal cumulatif et partitionnés par fonction de l'employé.

## CUME\_DIST

La fonction « **CUME\_DIST** » calcule la position d'une ligne renvoyée par une requête par rapport aux autres lignes renvoyées par la même requête, comme le définit la clause « **ORDER BY** ».



```
SQL> SELECT CODE_CATEGORIE CT, UNITES_STOCK US,
2          PERCENT_RANK() OVER ( PARTITION BY CODE_CATEGORIE
3                                ORDER BY UNITES_STOCK) PERCENT_RANK,
4          CUME_DIST() OVER ( PARTITION BY CODE_CATEGORIE
5                              ORDER BY UNITES_STOCK) CUME_DIST
6 FROM PRODUITS WHERE CODE_CATEGORIE < 3;
```

CT	US	PERCENT_RANK	CUME_DIST
1	15	0,000000000	0,062500000
1	17	0,066666667	0,250000000
1	17	0,066666667	0,250000000
1	17	0,066666667	0,250000000
1	20	0,266666667	0,312500000
1	39	0,333333333	0,375000000
1	50	0,400000000	0,437500000
1	52	0,466666667	0,500000000
1	57	0,533333333	0,562500000
1	60	0,600000000	0,625000000
1	69	0,666666667	0,687500000
1	100	0,733333333	0,750000000
1	111	0,800000000	0,812500000
1	125	0,866666667	0,937500000
1	125	0,866666667	0,937500000

1	1,000000000	1,000000000
2	0 0,000000000	0,045454545
2	4 0,047619048	0,090909091
2	6 0,095238095	0,136363636
2	13 0,142857143	0,181818182
...		

Dans l'exemple vous pouvez voir le renvoi d'un jeu de résultats fournissant la distribution cumulée des unités en stock par catégorie de produits.

## RATIO\_TO\_REPORT

La fonction « **RATIO\_TO\_REPORT** » calcule le ratio d'une valeur par rapport à la somme d'un jeu de valeurs. Si l'expression valeur expression est évaluée à NULL, alors « **RATIO\_TO\_REPORT** » est évaluée aussi à NULL, mais cela est traité comme un zéro pour calculer la somme des valeurs pour le dénominateur.

### EXPRESSION

Une expression impliquant des colonnes de références ou d'agrégats.

### OVER

La clause « **PARTITION BY** » définit les groupes dans lesquelles la fonction « **RATIO\_TO\_REPORT** » doit être calculée. Si la clause « **PARTITION BY** » est absente, alors la fonction est calculée sur tout le jeu de résultat.



```
SQL> SELECT SUBSTR(FONCTION,1,3) F,PAYS,
2          SUM(SALAIRE) S, SUM(SUM(SALAIRE)) OVER () T,
3          SUM(SUM(SALAIRE)) OVER
4              (PARTITION BY SUBSTR(FONCTION,1,3)) TF,
5          RATIO_TO_REPORT(SUM(SALAIRE)) OVER ()*100 RTR1,
6          RATIO_TO_REPORT(SUM(SALAIRE)) OVER
7              (PARTITION BY SUBSTR(FONCTION,1,3))*100 RTR2
8 FROM EMPLOYES GROUP BY SUBSTR(FONCTION,1,3), PAYS
9 ORDER BY SUBSTR(FONCTION,1,3), S;
```

F	PAYS	S	T	TF	RTR1	RTR2
Ass	-----	16 540	1 185 440	16 540	1,40	100,00
Che	-----	83 000	1 185 440	83 000	7,00	100,00
Pré	-----	150 000	1 185 440	150 000	12,65	100,00
Rep	Brésil	23 100	1 185 440	692 900	1,95	3,33
Rep	États-Unis	23 100	1 185 440	692 900	1,95	3,33
Rep	Autriche	25 600	1 185 440	692 900	2,16	3,69
Rep	Belgique	27 000	1 185 440	692 900	2,28	3,90
Rep	Danemark	27 500	1 185 440	692 900	2,32	3,97
Rep	Mexique	28 900	1 185 440	692 900	2,44	4,17
Rep	Italie	29 000	1 185 440	692 900	2,45	4,19
Rep	Finlande	29 800	1 185 440	692 900	2,51	4,30
Rep	Suède	31 300	1 185 440	692 900	2,64	4,52
Rep	Portugal	31 600	1 185 440	692 900	2,67	4,56
Rep	France	32 200	1 185 440	692 900	2,72	4,65
Rep	Pologne	32 400	1 185 440	692 900	2,73	4,68
Rep	Norvège	35 300	1 185 440	692 900	2,98	5,09
Rep	Canada	35 500	1 185 440	692 900	2,99	5,12
Rep	Irlande	36 400	1 185 440	692 900	3,07	5,25
Rep	Espagne	36 700	1 185 440	692 900	3,10	5,30
...						



## NTILE

La fonction « **NTILE** » effectue un partitionnement ordonné en un nombre spécifique de groupes appelés ‘buckets’, elle attribue le numéro unique de la partition à chaque ligne de cette partition. Elle est très utile parce qu'elle permet aux utilisateurs de diviser un jeu de donnée en quatre, trois ou différents groupes.

Les ‘buckets’ sont calculés de façon à ce que chaque ‘bucket’ ait exactement le même nombre de lignes attribuées ou au moins 1 ligne de plus que les autres.



```
SQL> SELECT ROW_NUMBER() OVER(ORDER BY SUM(PRIX_UNITAIRE*QUANTITE)) RN,
2      PAYS, SUM(PRIX_UNITAIRE*QUANTITE) CA,
3      NTILE(4) OVER(ORDER BY SUM(PRIX_UNITAIRE*QUANTITE)) NT
4  FROM CLIENTS NATURAL JOIN COMMANDES NATURAL JOIN DETAILS_COMMANDES
5  WHERE ANNEE = 2011 GROUP BY PAYS;
```

RN	PAYS	CA	NT
1	Pologne	10 464 648	1
2	Irlande	18 515 473	1
3	Norvège	19 121 805	1
4	Autriche	23 540 792	1
5	Finlande	26 656 539	1
6	Suisse	29 869 836	1
7	Belgique	30 031 948	2
8	Suède	32 004 959	2
9	Danemark	33 055 390	2
10	Portugal	36 860 875	2
11	Italie	42 563 171	2
12	Argentine	43 528 526	3
13	Canada	51 809 202	3
14	Venezuela	65 413 825	3
15	Mexique	72 323 122	3
16	Espagne	81 051 275	3
17	Royaume-Uni	112 585 949	4
18	Brésil	130 405 108	4
19	Allemagne	171 322 408	4
20	France	177 411 925	4
21	États-Unis	194 575 638	4

Dans l'exemple précédent le nombre d'enregistrements est 21 ; il ne peut pas être divisé par 4, ainsi le premier ensemble a 6 enregistrements et les suivants uniquement 5. Il est également possible d'utiliser la fonction « **NTILE** » pour filtrer les enregistrements et n'afficher que les enregistrements d'un seul ensemble. Par exemple n'afficher que les enregistrements qui font partie du troisième quart.



```
SQL> SELECT ROW_NUMBER() OVER( ORDER BY CA) RN, PAYS, CA
2  FROM ( SELECT ROW_NUMBER() OVER(
3           ORDER BY SUM(PRIX_UNITAIRE*QUANTITE)) RN,
4           PAYS, SUM(PRIX_UNITAIRE*QUANTITE) CA,
5           NTILE(4) OVER(
6           ORDER BY SUM(PRIX_UNITAIRE*QUANTITE)) NT
7  FROM CLIENTS NATURAL JOIN COMMANDES
8  NATURAL JOIN DETAILS_COMMANDES
9  WHERE ANNEE = 2011 GROUP BY PAYS ) SR WHERE NT = 3;
```

RN	PAYS	CA
-----		

1	Argentine	43	528	526
2	Canada	51	809	202
3	Venezuela	65	413	825
4	Mexique	72	323	122
5	Espagne	81	051	275

De la même manière vous pouvez retrouver dans l'exemple suivant l'affichage par pays le 1% des produits qui ont le poids le plus grand dans le chiffre d'affaire.



```
SQL> SELECT ROW_NUMBER() OVER( ORDER BY PAYS, CA DESC) RN,
2      PAYS, PRODUIT, CA, NT
3 FROM ( SELECT ROW_NUMBER() OVER(
4      ORDER BY SUM(DC.PRIX_UNITAIRE*DC.QUANTITE)) RN,
5      PAYS, NOM_PRODUIT PRODUIT,
6      SUM(DC.PRIX_UNITAIRE*DC.QUANTITE) CA,
7      NTILE(100) OVER ( PARTITION BY PAYS
8      ORDER BY SUM(DC.PRIX_UNITAIRE*DC.QUANTITE) DESC) NT
9      FROM CLIENTS NATURAL JOIN COMMANDES
10     NATURAL JOIN DETAILS_COMMANDES DC
11     JOIN PRODUITS USING(REF_PRODUIT)
12     WHERE ANNEE = 2011 GROUP BY PAYS, NOM_PRODUIT) WHERE NT = 1;
```

RN	PAYS	PRODUIT	CA	NT
1	Allemagne	Cake Mix	2 223 226	1
2	Allemagne	Fruit Cocktail	2 213 302	1
3	Argentine	Dried Apples	675 297	1
4	Argentine	Chocolate Biscuits Mix	648 238	1
5	Autriche	Crab Meat	410 989	1
6	Autriche	Perth Pasties	326 668	1
7	Belgique	Röd Kaviar	524 076	1
8	Belgique	Cake Mix	501 066	1
9	Brésil	Ravioli	1 813 829	1
10	Brésil	Chicken Soup	1 658 338	1
11	Canada	Fruit Cocktail	717 532	1
12	Canada	Pineapple	706 070	1
13	Danemark	Cake Mix	518 277	1
14	Danemark	Ravioli	482 567	1
15	Espagne	Ravioli	1 166 586	1
16	Espagne	Pineapple	1 102 903	1
17	États-Unis	Ravioli	2 760 241	1
...				

## WIDTH\_BUCKET

La fonction « **NTILE** » effectue un partitionnement des enregistrements en équilibrant le nombre des enregistrements dans chaque partition. La fonction « **WIDTH\_BUCKET** » découpe l'intervalle des valeurs en parts égales et renvoie pour chaque enregistrement l'appartenance à un des ces intervalles. La syntaxe de cette fonction est :

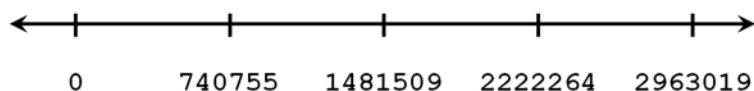
**WIDTH\_BUCKET** ( **expression**, **min**, **max**, **nombre\_partitions** )

**min** La borne inférieure pour l'expression.

**max** La borne supérieure pour l'expression.

**nombre\_partitions** Le nombre de parts pour le découpage de l'intervalle.

Voici la distribution dans les intervalles des valeurs pour l'exemple suivant :



```
SQL> SELECT PAYS, SUM(QUANTITE) Q,
2      ROUND(MAX(SUM(QUANTITE))OVER() / 4) B1,
3      ROUND(MAX(SUM(QUANTITE))OVER() / 2) B2,
4      ROUND(MAX(SUM(QUANTITE))OVER()*3 / 4) B3,
5      NTILE(4) OVER(ORDER BY SUM(QUANTITE)) NT,
6      WIDTH_BUCKET(SUM(QUANTITE), 0, MAX(SUM(QUANTITE)) OVER ()+1, 4) WB
7 FROM CLIENTS NATURAL JOIN COMMANDES NATURAL JOIN DETAILS_COMMANDES
8 WHERE ANNEE = 2011 GROUP BY PAYS;
```

PAYS		Q		B1		B2		B3	NT	WB
Pologne	159 140	<b>740 755</b>	1 481 509	2 222 264	1	1				
Irlande	282 713	740 755	1 481 509	2 222 264	1	1				
Norvège	291 405	740 755	1 481 509	2 222 264	1	1				
Autriche	362 205	740 755	1 481 509	2 222 264	1	1				
Finlande	407 997	740 755	1 481 509	2 222 264	1	1				
Suisse	457 257	740 755	1 481 509	2 222 264	1	1				
Belgique	457 380	740 755	1 481 509	2 222 264	2	1				
Suède	488 813	740 755	1 481 509	2 222 264	2	1				
Danemark	506 217	740 755	1 481 509	2 222 264	2	1				
Portugal	564 737	740 755	1 481 509	2 222 264	2	1				
Italie	649 492	740 755	1 481 509	2 222 264	2	1				
Argentine	663 302	740 755	1 481 509	2 222 264	3	1				
Canada	<b>794 266</b>	740 755	1 481 509	2 222 264	3	<b>2</b>				
Venezuela	998 992	740 755	1 481 509	2 222 264	3	2				
Mexique	1 101 722	740 755	1 481 509	2 222 264	3	2				
Espagne	1 238 242	740 755	1 481 509	2 222 264	3	2				
Royaume-Uni	<b>1 725 326</b>	740 755	1 481 509	2 222 264	4	<b>3</b>				
Brésil	1 992 712	740 755	1 481 509	2 222 264	4	3				
Allemagne	<b>2 621 324</b>	740 755	1 481 509	2 222 264	4	<b>4</b>				
France	2 713 106	740 755	1 481 509	2 222 264	4	4				
États-Unis	2 963 018	740 755	1 481 509	2 222 264	4	4				

Dans l'exemple précédent, vous pouvez voir le décalage entre la fonction « **NTILE** » et la fonction « **WIDTH\_BUCKET** » pour la répartition en bouquets de valeurs. Pour la borne supérieure de la fonction « **WIDTH\_BUCKET** » on rajoute une unité pour que la plus grande valeur fasse partie du dernier bouquet de valeurs.

Comme pour la fonction « **NTILE** », vous pouvez utiliser une sous-requête pour filtrer les enregistrements suivant les bouquets de valeurs. Les bornes utilisées peuvent être définies par des fonctions analytiques ; ainsi vous pouvez utiliser des valeurs contextuelles.

Voici un exemple qui permet d'afficher uniquement les produits qui se trouvent dans les trois derniers bouquets en tenant compte des quantités vendues par pays. Pour chaque pays il calcule une centaine de bouquets de valeurs fonctions de la plus grande et la plus petite quantité vendue par produit.



```
SQL> SELECT PAYS, PRODUIT, Q, NT, WB
2 FROM ( SELECT ROW_NUMBER() OVER(ORDER BY SUM(DC.QUANTITE)) RN, PAYS,
3      NOM_PRODUIT PRODUIT, SUM(DC.QUANTITE) Q,
4      NTILE(100)OVER(PARTITION BY PAYS ORDER BY SUM(DC.QUANTITE))NT,
5      WIDTH_BUCKET(SUM(DC.QUANTITE),MIN(SUM(DC.QUANTITE))
6      OVER (PARTITION BY PAYS)-1, MAX(SUM(DC.QUANTITE))
7      OVER (PARTITION BY PAYS)+1, 100) WB
```

```

8      FROM CLIENTS NATURAL JOIN COMMANDES
9      NATURAL JOIN DETAILS_COMMANDES DC
10     JOIN PRODUITS USING(REF_PRODUIT)
11     WHERE ANNEE = 2011 GROUP BY PAYS, NOM_PRODUIT)
12 WHERE WB >= 98 ORDER BY PAYS, Q;

```

PAYS	PRODUIT	Q	NT	WB
-----	-----	-----	-----	-----
Allemagne	Chang	27 401	100	100
Argentine	Dried Apples	8 671	100	100
Autriche	Mascarpone Fabioli	4 868	100	100
Belgique	Röd Kaviar	5 950	100	100
Brésil	Les Comptoirs - Olive Oil	19 501	98	98
Brésil	Outback Lager	19 603	99	100
Brésil	Beer	19 663	100	100
Canada	Northwoods Cranberry Sauce	9 199	100	100
Danemark	Tourtière	5 964	100	100
Espagne	Outback Lager	12 724	99	98
Espagne	Beer	12 859	100	100
États-Unis	Mishi Kobe Niku	30 003	99	98
États-Unis	Röd Kaviar	30 268	100	100
Finlande	Konbu	5 695	100	100
France	Mustard	27 265	100	100
Irlande	Dried Pears	4 167	100	100
Italie	Outback Lager	7 380	99	99
Italie	Syrup	7 426	100	100
...				

## Fonctions de fenêtre

Les fonctions de fenêtres peuvent être utilisées dans les calculs cumulatifs, de mouvements, et d'agréations centrées. Elles retournent une valeur pour chaque ligne de la table, laquelle dépend des autres lignes dans la fenêtre correspondante.

Les fonctions qui peuvent être utilisées comme des fonctions de fenêtre sont : « **SUM** », « **AVG** », « **MAX** », « **MIN** », « **COUNT** », « **STDDEV** », « **VARIANCE** », « **FIRST\_VALUE** », « **LAST\_VALUE** ».

### FIRST\_VALUE

La fonction « **FIRST\_VALUE** » autorise la sélection des premières lignes de la fenêtre. Ces lignes sont particulièrement utiles parce qu'elles sont souvent utilisées comme lignes de base dans les calculs. La syntaxe de la fonction est :

```

{ FIRST_VALUE | LAST_VALUE } ( expression )
    [ { RESPECT | IGNORE } NULLS ] OVER (...)

```



```

SQL> SET NULL "-----"
SQL> WITH VENTES AS (
2   SELECT CL.PAYS,CO.DATE_COMMANDE,SUM(CO.PORT) SP
3   FROM CLIENTS CL JOIN COMMANDES CO
4   ON ( CL.CODE_CLIENT = CO.CODE_CLIENT
5   AND CO.ANNEE = 2011 AND CO.MOIS = 5
6   AND CL.PAYS IN ('Suisse'))
7   GROUP BY CL.PAYS,CO.DATE_COMMANDE)

```

```

8  SELECT PAYS,JOUR,SP,
9         FIRST_VALUE( SP) OVER
10        ( ORDER BY JOUR RANGE INTERVAL '1' DAY PRECEDING) FV1,
11        FIRST_VALUE( SP) OVER ( ORDER BY JOUR
12                                RANGE 1 PRECEDING )           FV2,
13        FIRST_VALUE( SP) OVER ( ORDER BY JOUR)               FV3,
14        FIRST_VALUE( SP) IGNORE NULLS OVER ( ORDER BY JOUR) FV4
15  FROM VENTES V
16        PARTITION BY (PAYS) RIGHT OUTER JOIN
17        ( SELECT JOUR FROM DIM_TEMPS
18          WHERE ANNEE = 2011 AND MOIS_N = 5) T
19        ON ( V.DATE_COMMANDE = T.JOUR)
20  ORDER BY PAYS,JOUR;

```

PAYS	JOUR	SP	FV1	FV2	FV3	FV4
-----	-----	-----	-----	-----	-----	-----
Suisse	01/05/2011	-----	-----	-----	-----	-----
Suisse	02/05/2011	73,60	-----	-----	-----	73,60
Suisse	03/05/2011	87,60	73,60	73,60	-----	73,60
Suisse	04/05/2011	-----	87,60	87,60	-----	73,60
Suisse	05/05/2011	61,40	-----	-----	-----	73,60
Suisse	06/05/2011	78,50	61,40	61,40	-----	73,60
Suisse	07/05/2011	65,10	78,50	78,50	-----	73,60
Suisse	08/05/2011	122,00	65,10	65,10	-----	73,60
Suisse	09/05/2011	179,70	122,00	122,00	-----	73,60
...						

Dans l'exemple précédent, les fonctions « **FIRST\_VALUE** » sont calculées sans utilisation de partitionnement ; ainsi l'ensemble des enregistrements est une seule partition. La requête utilise une jointure externe avec la clause « **PARTITION BY** » qui garantit un enregistrement par jour. La colonne FV1 qui utilise une fenêtre logique d'une journée et la colonne FV2 qui utilise une fenêtre d'un enregistrement précédent retournent la même valeur. Les deux colonnes FV3 et FV4 affichent le premier enregistrement de la partition, mais FV4 ne tient pas compte des valeurs NULL.



```

SQL> WITH VENTES AS (
2  SELECT CL.PAYS,CO.DATE_COMMANDE,SUM(CO.PORT) SP
3        FROM CLIENTS CL JOIN COMMANDES CO
4        ON ( CL.CODE_CLIENT = CO.CODE_CLIENT
5              AND CO.ANNEE = 2011 AND CO.MOIS = 5
6              AND CL.PAYS IN ('France','Suisse'))
7  GROUP BY CL.PAYS,CO.DATE_COMMANDE)
8  SELECT PAYS,JOUR,SP,
9         FIRST_VALUE( SP) OVER
10        ( PARTITION BY PAYS
11          ORDER BY JOUR RANGE INTERVAL '1' DAY PRECEDING) FV1,
12        FIRST_VALUE( SP) OVER ( PARTITION BY PAYS
13                                ORDER BY JOUR RANGE 1 PRECEDING )           FV2,
14        FIRST_VALUE( SP) OVER ( PARTITION BY PAYS
15                                ORDER BY JOUR)                               FV3,
16        FIRST_VALUE( SP) IGNORE NULLS OVER (
17                                                PARTITION BY PAYS ORDER BY JOUR) FV4
18  FROM VENTES V
19        PARTITION BY (PAYS) RIGHT OUTER JOIN
20        ( SELECT JOUR FROM DIM_TEMPS
21          WHERE ANNEE = 2011 AND MOIS_N = 5) T
22        ON ( V.DATE_COMMANDE = T.JOUR)
23  ORDER BY PAYS,JOUR;

```

PAYS	JOUR	SP	FV1	FV2	FV3	FV4
-----	-----	-----	-----	-----	-----	-----
France	01/05/2011	473,20	473,20	473,20	473,20	473,20
France	02/05/2011	491,70	473,20	473,20	473,20	473,20
France	03/05/2011	371,90	491,70	491,70	473,20	473,20
France	04/05/2011	354,30	371,90	371,90	473,20	473,20
France	05/05/2011	428,30	354,30	354,30	473,20	473,20
...						
France	31/05/2011	361,80	343,80	343,80	473,20	473,20
Suisse	01/05/2011	-----	-----	-----	-----	-----
Suisse	02/05/2011	73,60	-----	-----	-----	73,60
Suisse	03/05/2011	87,60	73,60	73,60	-----	73,60
Suisse	04/05/2011	-----	87,60	87,60	-----	73,60
Suisse	05/05/2011	61,40	-----	-----	-----	73,60
Suisse	06/05/2011	78,50	61,40	61,40	-----	73,60
Suisse	07/05/2011	65,10	78,50	78,50	-----	73,60
Suisse	08/05/2011	122,00	65,10	65,10	-----	73,60
Suisse	09/05/2011	179,70	122,00	122,00	-----	73,60
Suisse	10/05/2011	130,90	179,70	179,70	-----	73,60
Suisse	11/05/2011	159,50	130,90	130,90	-----	73,60
Suisse	12/05/2011	52,80	159,50	159,50	-----	73,60
Suisse	13/05/2011	71,00	52,80	52,80	-----	73,60
...						

## LAST\_VALUE

La fonction « **LAST\_VALUE** » autorise la sélection des dernières lignes de la fenêtre. Ces lignes sont particulièrement utiles parce qu'elles sont souvent utilisées comme lignes de base dans les calculs.



```
SQL> SELECT PAYS, MOIS, SUM(DC.QUANTITE) Q,
2          LAST_VALUE( SUM(DC.QUANTITE)) OVER
3          ( PARTITION BY PAYS ORDER BY MOIS ROWS
4            BETWEEN CURRENT ROW AND 1 FOLLOWING ) LV1,
5          LAST_VALUE( SUM(DC.QUANTITE)) OVER
6          ( PARTITION BY PAYS ORDER BY MOIS ) LV2,
7          LAST_VALUE( SUM(DC.QUANTITE)) OVER
8          ( PARTITION BY PAYS ) LV3
9 FROM CLIENTS NATURAL JOIN COMMANDES
10 NATURAL JOIN DETAILS_COMMANDES DC
11 JOIN PRODUITS USING(REF_PRODUIT)
12 WHERE ANNEE = 2010 AND PAYS IN ('France','Suisse')
13 GROUP BY PAYS, MOIS;
```

PAYS	MOIS	Q	LV1	LV2	LV3
-----	-----	-----	-----	-----	-----
France	1	369 797	266 531	369 797	188 311
France	2	266 531	292 601	266 531	188 311
France	3	292 601	320 200	292 601	188 311
...					
France	10	383 735	355 441	383 735	188 311
France	11	355 441	188 311	355 441	188 311
France	12	188 311	188 311	188 311	188 311
Suisse	1	48 029	70 379	48 029	49 359
Suisse	2	70 379	26 949	70 379	49 359
Suisse	3	26 949	55 450	26 949	49 359

```
...
Suisse      11      54 646      49 359      54 646      49 359
Suisse      12      49 359      49 359      49 359      49 359
```


Il faut être attentif parceque si, pour la fonction « **LAST\_VALUE** », vous définissez un ordre de tri, il faut impérativement définir la fenêtre ; sinon la valeur de la fonction est identique à celle de l'enregistrement en cours. Si vous n'initialisez pas l'ordre de tri alors la dernière valeur de la partition est retournée.

## NTH\_VALUE


**11g**

A partir de la version Oracle 11g il est possible de retrouver n'importe quelle occurrence à l'aide la fonction « **NTH\_VALUE** ». La syntaxe de la fonction est :

**NTH\_VALUE (expression, occurrence) [ {RESPECT | IGNORE} NULLS ] OVER ( ... )**



```
SQL> SELECT PAYS, NOM,
2      ROW_NUMBER() OVER ( PARTITION BY PAYS ORDER BY NOM) N,
3      NTH_VALUE(NOM,2) OVER (PARTITION BY PAYS ORDER BY NOM
4                          ROWS BETWEEN UNBOUNDED PRECEDING AND
5                          UNBOUNDED FOLLOWING ) NOM2,
6      NTH_VALUE(NOM,3) OVER (PARTITION BY PAYS ORDER BY NOM
7                          ROWS BETWEEN UNBOUNDED PRECEDING AND
8                          UNBOUNDED FOLLOWING ) NOM3,
9      NTH_VALUE(NOM,4) OVER (PARTITION BY PAYS ORDER BY NOM
10                         ROWS BETWEEN UNBOUNDED PRECEDING AND
11                         UNBOUNDED FOLLOWING ) NOM4,
12     NTH_VALUE(NOM,5) OVER (PARTITION BY PAYS ORDER BY NOM
13                         ROWS BETWEEN UNBOUNDED PRECEDING AND
14                         UNBOUNDED FOLLOWING ) NOM5
15 FROM EMPLOYES WHERE PAYS IS NOT NULL;
```

PAYS	NOM	N	NOM2	NOM3	NOM4	NOM5
Allemagne	Charles	1	King	Kremser	Marielle	Mennetrier
Allemagne	King	2	King	Kremser	Marielle	Mennetrier
Allemagne	Kremser	3	King	Kremser	Marielle	Mennetrier
Allemagne	Marielle	4	King	Kremser	Marielle	Mennetrier
Allemagne	Mennetrier	5	King	Kremser	Marielle	Mennetrier
Allemagne	Mure	6	King	Kremser	Marielle	Mennetrier
Allemagne	Thimoleon	7	King	Kremser	Marielle	Mennetrier
Argentine	Burst	1	Cleret	Damas	Marchand	Montesinos
Argentine	Cleret	2	Cleret	Damas	Marchand	Montesinos
Argentine	Damas	3	Cleret	Damas	Marchand	Montesinos
Argentine	Marchand	4	Cleret	Damas	Marchand	Montesinos
Argentine	Montesinos	5	Cleret	Damas	Marchand	Montesinos
Autriche	Blard	1	Cazade	Guillossou	Idesheim	-----
Autriche	Cazade	2	Cazade	Guillossou	Idesheim	-----
Autriche	Guillossou	3	Cazade	Guillossou	Idesheim	-----
Autriche	Idesheim	4	Cazade	Guillossou	Idesheim	-----
Belgique	Bazart	1	Canu	Rivat	Thomas	-----
Belgique	Canu	2	Canu	Rivat	Thomas	-----
Belgique	Rivat	3	Canu	Rivat	Thomas	-----
Belgique	Thomas	4	Canu	Rivat	Thomas	-----
Brésil	Jenny	1	Poidatz	Viry	-----	-----
Brésil	Poidatz	2	Poidatz	Viry	-----	-----
Brésil	Viry	3	Poidatz	Viry	-----	-----

## LAG

Les fonctions « **LAG** » et « **LEAD** » sont utiles pour comparer des valeurs quand les positions relatives des lignes peuvent être connues de manière fiable. Elles fonctionnent en spécifiant le nombre de lignes qui séparent la ligne cible à partir de la ligne courante. Il faut faire attention car les fonctions analytiques « **LAG** » et « **LEAD** » ne supportent pas l'utilisation des fenêtres logiques ou positionnelles.

La fonction « **LAG** » fournit un accès à une ligne à un espacement donné avant la position courante.

**LAG** ( **EXPRESSION** [, **OFFSET**] [, **DEFAULT**] ) [ { **RESPECT** | **IGNORE** } **NULLS** ] **OVER** ( ... )

<b>EXPRESSION</b>	Une expression impliquant des colonnes de références ou d'agrégats.
<b>OFFSET</b>	Le nombre de lignes qui séparent la ligne cible à partir de la ligne courante, la valeur par défaut étant 1.
<b>DEFAULT</b>	Un paramètre optionnel ; c'est la valeur retournée si offset tombe en dehors de la limite de la table ou de la partition ; la valeur par défaut est NULL.

Les deux colonnes LAG1 et LAG2 ont les mêmes valeurs sauf pour la valeur par défaut qui dans le cas de LAG1 est -1 comme spécifié et pour LAG2 elle est NULL.



```
SQL>SELECT ROW_NUMBER()OVER(ORDER BY PAYS,MOIS) RN,PAYS,MOIS M,SUM(PORT) P,
2      LAG(SUM(PORT),1,-1) OVER( PARTITION BY PAYS ORDER BY MOIS) LAG1,
3      LAG(SUM(PORT)) OVER ( PARTITION BY PAYS ORDER BY MOIS) LAG2,
4      LAG(SUM(PORT),6,-1) OVER( PARTITION BY PAYS ORDER BY MOIS) LAG3
5  FROM CLIENTS NATURAL JOIN COMMANDES
6  WHERE ANNEE = 2010 GROUP BY PAYS, MOIS;
```

RN	PAYS	M	P	LAG1	LAG2	LAG3
-----						
1	Allemagne	1	6 654,2	-1,0	-----	-1,0
2	Allemagne	2	6 524,8	6 654,2	6 654,2	-1,0
3	Allemagne	3	7 122,7	6 524,8	6 524,8	-1,0
4	Allemagne	4	5 972,6	7 122,7	7 122,7	-1,0
5	Allemagne	5	5 519,4	5 972,6	5 972,6	-1,0
6	Allemagne	6	7 882,6	5 519,4	5 519,4	-1,0
7	Allemagne	7	5 465,8	7 882,6	7 882,6	6 654,2
8	Allemagne	8	6 156,9	5 465,8	5 465,8	6 524,8
9	Allemagne	9	5 854,5	6 156,9	6 156,9	7 122,7
10	Allemagne	10	5 037,7	5 854,5	5 854,5	5 972,6
11	Allemagne	11	5 580,5	5 037,7	5 037,7	5 519,4
12	Allemagne	12	6 060,4	5 580,5	5 580,5	7 882,6
13	Argentine	1	1 814,0	-1,0	-----	-1,0
14	Argentine	2	1 340,5	1 814,0	1 814,0	-1,0
15	Argentine	3	1 887,9	1 340,5	1 340,5	-1,0
16	Argentine	4	2 352,3	1 887,9	1 887,9	-1,0
17	Argentine	5	1 167,9	2 352,3	2 352,3	-1,0
18	Argentine	6	2 040,1	1 167,9	1 167,9	-1,0
19	Argentine	7	1 687,5	2 040,1	2 040,1	1 814,0
20	Argentine	8	1 431,8	1 687,5	1 687,5	1 340,5
...						

## LEAD

La fonction « **LEAD** » fournit un accès à une ligne à un espacement donné après la position courante.





**LEAD( EXPRESSION[,OFFSET][,DEFAULT])[{RESPECT|IGNORE}NULLS]OVER(...)**

```
SQL> SELECT ROW_NUMBER()OVER(ORDER BY PAYS,MOIS)RN,PAYS,MOIS M,SUM(PORT) P,
2      LEAD(SUM(PORT),1,-1) OVER( PARTITION BY PAYS ORDER BY MOIS) LEAD1,
3      LEAD(SUM(PORT),6,-1) OVER( PARTITION BY PAYS ORDER BY MOIS) LEAD2
4  FROM CLIENTS NATURAL JOIN COMMANDES
5  WHERE ANNEE = 2010 GROUP BY PAYS, MOIS;
```

	RN	PAYS	M	P	LEAD1	LEAD2
1	1	Allemagne	1	6 654,2	6 524,8	5 465,8
2	2	Allemagne	2	6 524,8	7 122,7	6 156,9
3	3	Allemagne	3	7 122,7	5 972,6	5 854,5
4	4	Allemagne	4	5 972,6	5 519,4	5 037,7
5	5	Allemagne	5	5 519,4	7 882,6	5 580,5
6	6	Allemagne	6	7 882,6	5 465,8	6 060,4
7	7	Allemagne	7	5 465,8	6 156,9	-1,0
8	8	Allemagne	8	6 156,9	5 854,5	-1,0
9	9	Allemagne	9	5 854,5	5 037,7	-1,0
10	10	Allemagne	10	5 037,7	5 580,5	-1,0
11	11	Allemagne	11	5 580,5	6 060,4	-1,0
12	12	Allemagne	12	6 060,4	-1,0	-1,0
13	1	Argentine	1	1 814,0	1 340,5	1 687,5
14	2	Argentine	2	1 340,5	1 887,9	1 431,8
15	3	Argentine	3	1 887,9	2 352,3	1 549,8
16	4	Argentine	4	2 352,3	1 167,9	1 346,2

...

```
SQL> WITH VENTES AS (
2  SELECT CL.PAYS,CO.DATE_COMMANDE,SUM(CO.PORT) SP
3      FROM CLIENTS CL JOIN COMMANDES CO
4      ON ( CL.CODE_CLIENT = CO.CODE_CLIENT
5          AND CO.DATE_COMMANDE BETWEEN'10/05/2011' AND '20/05/2011'
6          AND CL.PAYS IN ('Autriche','Suisse'))
7  GROUP BY CL.PAYS,CO.DATE_COMMANDE)
8  SELECT PAYS,JOUR,SP,
9      LAG (SP) OVER ( PARTITION BY PAYS ORDER BY JOUR) LAG1,
10     LAG (SP) IGNORE NULLS
11     OVER ( PARTITION BY PAYS ORDER BY JOUR) LAG2,
12     LEAD(SP) OVER ( PARTITION BY PAYS ORDER BY JOUR) LEAD1,
13     LEAD(SP) IGNORE NULLS
14     OVER ( PARTITION BY PAYS ORDER BY JOUR) LEAD2
15  FROM VENTES V PARTITION BY (PAYS) RIGHT OUTER JOIN
16     ( SELECT JOUR FROM DIM_TEMPS
17       WHERE JOUR BETWEEN '10/05/2011' AND '20/05/2011') T
18     ON ( V.DATE_COMMANDE = T.JOUR);
```

	PAYS	JOUR	SP	LAG1	LAG2	LEAD1	LEAD2
Autriche	10/05/2011	-----	-----	-----	-----	50,20	50,20
Autriche	11/05/2011	50,20	-----	-----	-----	-----	81,60
Autriche	12/05/2011	-----	50,20	50,20	-----	-----	81,60
Autriche	13/05/2011	-----	-----	50,20	-----	-----	81,60
Autriche	14/05/2011	-----	-----	50,20	-----	-----	81,60
Autriche	15/05/2011	-----	-----	50,20	81,60	-----	81,60
Autriche	16/05/2011	81,60	-----	50,20	52,30	52,30	-----
Autriche	17/05/2011	52,30	81,60	81,60	66,30	66,30	-----

Autriche	18/05/2011	66,30	52,30	52,30	64,20	64,20
Autriche	19/05/2011	64,20	66,30	66,30	52,00	52,00
Autriche	20/05/2011	52,00	64,20	64,20	-----	-----
Suisse	10/05/2011	130,90	-----	-----	159,50	159,50
Suisse	11/05/2011	159,50	130,90	130,90	52,80	52,80
...						

## LISTAGG

11g

La fonction « **LISTAGG** » permet de retourner une chaîne de caractères qui est composée par les arguments concatènes dans l'ordre de tri défini dans la parenthèse qui suit les mots clé « **WITHIN GROUP** ». La syntaxe de la fonction est :

**LISTAGG ( expression ) WITHIN GROUP ( ORDER BY ... ) OVER ( ... )**

Pour l'exemple suivant il faut d'abord convertir la page de code du champ NOM car le retour de la fonction « **LISTAGG** » est une chaîne de caractères en page de code par défaut.

```
SQL> SELECT M.NOM MANAGER, LISTAGG(
2      CONVERT( E.NOM||'; ', 'WE8MSWIN1252','AL16UTF16'))
3      WITHIN GROUP (ORDER BY E.NOM) EMPLOYES
4 FROM EMPLOYES E JOIN EMPLOYES M ON ( E.REND_COMPTE = M.NO_EMPLOYE )
5 GROUP BY M.NOM;
```

MANAGER	EMPLOYES
Belin	Aubert; Blard; Cazade; Charles; Cremel; Guillosoy; Idesh eim; King; Kremser; Lefebvre; Maillard; Marielle; Mennetr ier; Messelier; Mure; Pequignot; Seigne; Thimoleon; Tward owski; Weiss;
Brasseur	Belin; Chambaud; Leger; Ragon;
Buchanan	Bodard; Brunet; Coutou; Gardeil; Gerard; Herve; Jacquot; Jeandel; Perny; Rollet; Silberreiss; Urbaniak;
Chambaud	Alvarez; Arrambide; Clement; Di Clemente; Dodsworth; Dohr ; Griner; Letertre; Lombard; Malejac; Mangeard; Piroddi; Suyama;
Fuller	Buchanan; Splingart;
Leger	Aubry; Barre; Berlioz; Bettan; Cheutin; Davolio; Delorgue ; Destenay; Espeche; Falatik; Lamarre; Pagani; Peacock; P etit; Valot; Zonca;
...	

La fonction peut être utilisée avec une requête qui n'est pas agrégée à l'aide de l'extension analytique « **OVER PARTITION BY** ».

```
SQL> SELECT PAYS, NOM, SALAIRE,
2      LISTAGG(CONVERT( NOM||'; ', 'WE8MSWIN1252','AL16UTF16'))
3      WITHIN GROUP (ORDER BY NOM)
4      OVER ( PARTITION BY PAYS) EMPLOYES
5 FROM EMPLOYES WHERE PAYS IN ('France','Autriche','Suisse');
6
```

PAYS	NOM	SALAIRE	EMPLOYES
Autriche	Blard	5 900	Blard; Cazade; Guillosoy;

			Idesheim;
Autriche	Cazade	7 200	Blard; Cazade; Guilloso; Idesheim;
Autriche	Guilloso	5 200	Blard; Cazade; Guilloso; Idesheim;
Autriche	Idesheim	7 300	Blard; Cazade; Guilloso; Idesheim;
France	Frederic	8 100	Frederic; Gregoire; Regner; Teixeira;
France	Gregoire	8 000	Frederic; Gregoire; Regner; Teixeira;
France	Regner	9 300	Frederic; Gregoire; Regner; Teixeira;
France	Teixeira	6 800	Frederic; Gregoire; Regner; Teixeira;
Suisse	Aubert	5 100	Aubert; Lefebvre; Maillard; Pequignot; Seigne;
Suisse	Lefebvre	9 200	Aubert; Lefebvre; Maillard; Pequignot; Seigne;
Suisse	Maillard	6 600	Aubert; Lefebvre; Maillard; Pequignot; Seigne;
Suisse	Pequignot	6 200	Aubert; Lefebvre; Maillard; Pequignot; Seigne;
Suisse	Seigne	9 600	Aubert; Lefebvre; Maillard; Pequignot; Seigne;