IS-Net 29 DATA WEBHOUSE

Informatique de gestion et
systèmes d'information

Options d'Expressions SQL Analytiques

Centre de compétences

Hes·so
Haute Ecole Spécialisée
de Suisse Occidentale

LIBD@hegne.ch / Mars 2003

ISNet29_SQL_aggregation.doc

TABLE DES MATIERES

1 Introduction

Ce document introduit un concept important utilisé dans les *datawarehouses*[1]: les extensions SQL d'agrégations et d'analyses. Ce document est divisé en deux chapitres: le premier concerne les agrégations avec le langage SQL et le deuxième concerne les fonctions analytiques.

CHAPITRE 1: Agrégations avec le langage SQL

L'agrégation est un élément essentiel dans un *datawarehouse*. Afin d'améliorer les performances des requêtes SQL sur la base de données, Oracle fournit les extensions à la clause GROUP BY suivante:

- ROLLUP et CUBE
- Trois fonctions de groupe (GROUPING)
- Expression de type GROUPING SETS

Les extensions CUBE, ROLLUP, et GROUPING SETS permettent d'exécuter des requêtes et de générer des rapports plus facilement et plus rapidement. ROLLUP calcule les agrégations de type SUM, COUNT, MAX, MIN et AVG en augmentant le niveau d'agrégation, à partir du niveau le plus détaillé jusqu'au grand total. CUBE est une extension similaire à ROLLUP, permettant à une simple expression de calculer toutes les combinaisons possibles d'agrégations. CUBE est capable de générer l'information nécessaire pour les tableaux-croisés utilisés dans les rapports (en une seule requête).

Les extensions CUBE, ROLLUP, et GROUPING SETS permettent de spécifier exactement les groupes d'intérêt dans la clause GROUP BY. Ceci permet des analyses croisées efficaces sur de multiples dimensions sans utiliser l'extension CUBE. Parce que le calcul de cubes génère un processus lourd, remplacer ce dernier avec les GROUPINGS SETS peut augmenter les performances de manière significatives. CUBE, ROLLUP et GROUPING SETS produisent un lot unique de résultat qui est équivalent à l'utilisation de UNION ALL.
Pour améliorer les performances, CUBE, ROLLUP et GROUPING SETS peuvent être parallélisés. De ce fait le calcul d'agrégation est plus efficace pour les performances de la base de données.

Les trois fonctions de groupe GROUPING aident à identifier le groupe pour lequel appartient chaque ligne et permet de classer les lignes en sous-total et de filtrer le résultat.

2 Analyse au travers de multiples dimensions

Un concept clé dans les systèmes décisionnels est l'analyse multidimensionnelle: examine l'entreprise à partir de toutes les combinaisons dimensionnelles nécessaires. Notre modèle dimensionnel se compose des dimensions: CALENDRIER, SITE, ADRESSE_IP et SUBNET. Les événements ou entités associés avec un lot particulier de valeurs dimensionnelles font références aux faits, valeurs de la table de **fait** (dans notre cas il s'agit de la table PROXY).

[1] Entrepôt de données

Pour plus de renseignements, se référer au document "Introduction aux Entrepôts de données".

Le but des extensions SQL et fonctions analytiques, sont d'optimiser les performances liées aux requêtes de génération de rapports. Etudions de plus près ces caractéristiques.

3 L'extension ROLLUP de la clause GROUP BY

ROLLUP permet le calcule de différents niveaux de sous-totaux à travers un groupe spécifique de dimensions. Il calcule également un grand total. ROLLUP est très efficace et ne surcharge que très peu la requête.

ROLLUP fonctionne de la manière suivante: il crée des sous-totaux qui vont du niveau le plus détaillé jusqu'au grand total, suivant une liste de groupe spécifiée dans la clause ROLLUP. Celles-ci définit l'ordre de groupage des colonnes.

ROLLUP crée des sous-totaux à *n*+1 niveaux, ou *n* est le nombre de colonnes groupées. Par exemple, si dans une requête on spécifie le groupage des colonnes *annee*, *region* et *departement* (*n*=3), il y aura quatre niveaux d'agrégation.

3.1 Quand faut-il utiliser ROLLUP?

On peut utiliser ROLLUP lorsque des tâches incluent des sous-totaux.

- C'est particulièrement utile d'avoir des sous-totaux dans une dimension hiérarchique de type temporelle ou géographique. Par exemple une requête spécifiant ROLLUP(annee, mois, jour) ou ROLLUP(pays, canton, ville).
- C'est également indiqué dans l'emploi de tables de type *summary tables* (voir documentation sur les *materialized view*), ROLLUP peut simplifier et améliorer la maintenance de ces tables.

Requête simple, sans sous-totaux

```
SQL> select annee, region, departement, revenu
  2  from ventes order by annee, region;

ANNEE    REGION     DEPARTEMENT                REVENU
-------- ---------- -------------------- ----------
03.03.00 Centre     VoitureLocation           75000
03.03.00 Centre     VoitureVente              74000
03.03.00 Est        VoitureLocation           89000
03.03.00 Est        VoitureVente             115000
03.03.00 Ouest      VoitureLocation           87000
03.03.00 Ouest      VoitureVente              86000
03.03.01 Centre     VoitureLocation           82000
03.03.01 Centre     VoitureVentes             85000
03.03.01 Est        VoitureLocation          101000
03.03.01 Est        VoitureVentes            137000
03.03.01 Ouest      VoitureLocation           96000
03.03.01 Ouest      VoitureVentes             97000
```

Requête impliquant l'extension ROLLUP

```
SQL> SELECT annee, region, departement,
  2   SUM(revenu) AS Revenu FROM ventes
  3   GROUP BY ROLLUP (annee, region, departement) ;

ANNEE    REGION     DEPARTEMENT                REVENU
-------- ---------- -------------------- ----------
03.03.00 Centre     VoitureLocation            75000
03.03.00 Centre     VoitureVente               74000
03.03.00 Centre                               149000
03.03.00 Est        VoitureLocation            89000
03.03.00 Est        VoitureVente              115000
03.03.00 Est                                  204000
03.03.00 Ouest      VoitureLocation            87000
03.03.00 Ouest      VoitureVente               86000
03.03.00 Ouest                                173000
03.03.00                                      526000
03.03.01 Centre     VoitureLocation            82000
03.03.01 Centre     VoitureVentes              85000
03.03.01 Centre                               167000
03.03.01 Est        VoitureLocation           101000
03.03.01 Est        VoitureVentes             137000
03.03.01 Est                                  238000
03.03.01 Ouest      VoitureLocation            96000
03.03.01 Ouest      VoitureVentes              97000
03.03.01 Ouest                                193000
03.03.01                                      598000
                                             1124000
```

Idem, mais sans utiliser l'extension ROLLUP (beaucoup plus lourd!)

```
SELECT annee, region, departement, SUM(revenu) AS Revenu
 FROM ventes
 GROUP BY annee, region, departement
UNION ALL
 SELECT annee, region, to_char(null), SUM(revenu) AS
Revenu
 FROM ventes
 GROUP BY annee, region
UNION ALL
 SELECT annee, to_char(null), to_char(null), SUM(revenu)
AS Revenu
 FROM ventes
 GROUP BY annee
UNION ALL
 SELECT to_date(null), to_char(null), to_char(null),
SUM(revenu) AS Revenu
 FROM ventes ;
```

3.2 ROLLUP partiel (partial ROLLUP)

Il est également possible d'avoir seulement les sous-totaux pour certaines colonnes. Par exemple:

```
SQL> SELECT annee, region, departement,
  2   SUM(revenu) AS Revenu FROM ventes
  3   GROUP BY annee, ROLLUP(region, departement) ;

ANNEE    REGION     DEPARTEMENT                REVENU
-------- ---------- -------------------- ----------
03.03.00 Centre     VoitureLocation           75000
03.03.00 Centre     VoitureVente              74000
03.03.00 Centre                              149000
03.03.00 Est        VoitureLocation           89000
03.03.00 Est        VoitureVente             115000
03.03.00 Est                                 204000
03.03.00 Ouest      VoitureLocation           87000
03.03.00 Ouest      VoitureVente              86000
03.03.00 Ouest                               173000
03.03.00                                     526000
03.03.01 Centre     VoitureLocation           82000
03.03.01 Centre     VoitureVentes             85000
03.03.01 Centre                              167000
03.03.01 Est        VoitureLocation          101000
03.03.01 Est        VoitureVentes            137000
03.03.01 Est                                 238000
03.03.01 Ouest      VoitureLocation           96000
03.03.01 Ouest      VoitureVentes             97000
03.03.01 Ouest                               193000
03.03.01                                     598000
```

4 L'extension CUBE de la clause GROUP BY

L'extension CUBE utilise un lot spécifique de groupage de colonnes et crée des sous-totaux pour TOUTES les combinaisons possibles. En terme d'analyse multidimensionnelle, CUBE génère tout les sous-totaux qui peuvent être calculés pour les données d'un cube avec ces dimensions. Par exemple, l'expression CUBE(annee, region, departement), donnera un résultat incluant toutes les valeurs qui seraient générées par ROLLUP, avec en plus d'autres combinaisons (voir exemple plus loin).

4.1 Quand faut-il utiliser CUBE?

On peut utiliser l'extension CUBE dans toutes les situations ou l'on requiert des rapports de tableaux-croisés. Les données dont a besoin un tableau-croisé peuvent être générées avec la seule instruction SELECT et CUBE. Comme pour ROLLUP, CUBE est bénéfique dans le cas d'utilisation des *summary tables*. L'insertion de données dans les *summary tables* est d'autant plus rapide si la requête englobant l'extension CUBE est exécutée en parallèle.

Note: CUBE convient particulièrement bien dans les requêtes qui utilisent des colonnes provenant de plusieurs dimensions plutôt que de colonnes représentant différents niveaux d'une seule dimension.

Requête impliquant l'extension CUBE

```
SQL> SELECT annee, region, departement,
  2   SUM(revenu) AS Revenu FROM ventes
  3   GROUP BY CUBE (annee, region, departement) ;

-- résultats du query sur la page suivante
```

Résultat de la requête précédente (CUBE)

ANNEE	REGION	DEPARTEMENT	REVENU
03.03.00	Centre	VoitureLocation	75000
03.03.00	Centre	VoitureVente	74000
03.03.00	Centre		*149000*
03.03.00	Est	VoitureLocation	89000
03.03.00	Est	VoitureVente	115000
03.03.00	Est		*204000*
03.03.00	Ouest	VoitureLocation	87000
03.03.00	Ouest	VoitureVente	86000
03.03.00	Ouest		*173000*
03.03.00		VoitureLocation	251000
03.03.00		VoitureVente	275000
03.03.00			*526000*
03.03.01	Centre	VoitureLocation	82000
03.03.01	Centre	VoitureVentes	85000
03.03.01	Centre		*167000*
03.03.01	Est	VoitureLocation	101000
03.03.01	Est	VoitureVentes	137000
03.03.01	Est		*238000*
03.03.01	Ouest	VoitureLocation	96000
03.03.01	Ouest	VoitureVentes	97000
03.03.01	Ouest		*193000*
03.03.01		VoitureLocation	279000
03.03.01		VoitureVentes	319000
03.03.01			*598000*
	Centre	VoitureLocation	157000
	Centre	VoitureVente	74000
	Centre	VoitureVentes	85000
	Centre		*316000*
	Est	VoitureLocation	190000
	Est	VoitureVente	115000
	Est	VoitureVentes	137000
	Est		*442000*
	Ouest	VoitureLocation	183000
	Ouest	VoitureVente	86000
	Ouest	VoitureVentes	97000
	Ouest		*366000*
		VoitureLocation	530000
		VoitureVente	275000
		VoitureVentes	319000
			1124000

4.2 CUBE partiel

CUBE partiel (*partial* CUBE) ressemble au ROLLUP partiel (*partial* ROLLUP) dans le sens que l'on peut la limiter à certaines dimensions. Les sous-totaux de toutes les combinaisons possibles sont limitées aux dimensions qui se trouvent entre parenthèse. Par exemple:

```
SQL> SELECT annee, region, departement,
  2   SUM(revenu) AS Revenu FROM ventes
  3   GROUP BY annee, CUBE(region, departement) ;

ANNEE    REGION     DEPARTEMENT                REVENU
-------- ---------- -------------------- ----------
03.03.00 Centre     VoitureLocation             75000
03.03.00 Centre     VoitureVente                74000
03.03.00 Centre                                149000
03.03.00 Est        VoitureLocation             89000
03.03.00 Est        VoitureVente               115000
03.03.00 Est                                   204000
03.03.00 Ouest      VoitureLocation             87000
03.03.00 Ouest      VoitureVente                86000
03.03.00 Ouest                                 173000
03.03.00            VoitureLocation            251000
03.03.00            VoitureVente               275000
03.03.00                                       526000
03.03.01 Centre     VoitureLocation             82000
03.03.01 Centre     VoitureVentes               85000
03.03.01 Centre                                167000
03.03.01 Est        VoitureLocation            101000
03.03.01 Est        VoitureVentes              137000
03.03.01 Est                                   238000
03.03.01 Ouest      VoitureLocation             96000
03.03.01 Ouest      VoitureVentes               97000
03.03.01 Ouest                                 193000
03.03.01            VoitureLocation            279000
03.03.01            VoitureVentes              319000
03.03.01                                       598000
```

Il est possible de calculer les sous-totaux sans l'extension CUBE. Néanmoins ce mécanisme est très lourd, parce que dans ce cas, pour *n* dimension du cube, 2 à la puissance *n* SELECT devront être exécutés. Dans un exemple à trois dimensions, il faudra 8 SELECT reliés par des UNION ALL !

5 Fonctions de groupes (GROUPING Functions)

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" (espace blanc) créées par ROLLUP ou CUBE? Il existe un moyen et nous allons voir lequel.
Il existe à partir de la version 8i la fonction GROUPING, et dès la version 9i, les fonctions GROUPING_ID et GROUP_ID.

5.1 Fonction GROUPING

En utilisant un colonne comme argument, GROUPING retourne 1 quand il rencontre une valeur NULL créée par ROLLUP ou CUBE. En effet, si un NULL indique un sous-total, GROUPING retourne un 1. N'importe quelle autre valeur, incluant une valeur stockée NULL, retourne un 0.

Requête de type ROLLUP avec une fonction GROUPING

```
SELECT annee, region, departement, SUM(revenu) AS Revenu,
  GROUPING(annee) AS AN, GROUPING(region) AS RE,
  GROUPING(departement) AS DE
  FROM ventes
  GROUP BY ROLLUP (annee, region, departement) ;

ANNEE    REGION   DEPARTEMENT          REVENU     AN     RE     DE
-------- -------- ------------------------ ------ ------------
03.03.00 Centre   VoitureLocation       75000      0      0      0
03.03.00 Centre   VoitureVente          74000      0      0      0
03.03.00 Centre                        149000      0      0      1
03.03.00 Est      VoitureLocation       89000      0      0      0
03.03.00 Est      VoitureVente         115000      0      0      0
03.03.00 Est                           204000      0      0      1
03.03.00 Ouest    VoitureLocation       87000      0      0      0
03.03.00 Ouest    VoitureVente          86000      0      0      0
03.03.00 Ouest                         173000      0      0      1
03.03.00                               526000      0      1      1
03.03.01 Centre   VoitureLocation       82000      0      0      0
03.03.01 Centre   VoitureVentes         85000      0      0      0
03.03.01 Centre                        167000      0      0      1
03.03.01 Est      VoitureLocation      101000      0      0      0
03.03.01 Est      VoitureVentes        137000      0      0      0
03.03.01 Est                           238000      0      0      1
03.03.01 Ouest    VoitureLocation       96000      0      0      0
03.03.01 Ouest    VoitureVentes         97000      0      0      0
03.03.01 Ouest                         193000      0      0      1
03.03.01                               598000      0      1      1
                                      1124000      1      1      1
```

On peut augmenter la lisibilité en utilisant les fonctions GROUPING et DECODE comme dans l'exemple ci-dessous.

Lisibilité améliorée avec DECODE

```
SELECT DECODE(GROUPING (region), 1,'Total régions', region)
   AS Région,
 DECODE(GROUPING(departement), 1, 'Total départements',
   departement) AS Département,
 TO_CHAR(SUM(revenu), '9,999,999,999') Revenu
 FROM ventes
 GROUP BY CUBE (region, departement) ;

RÉGION         DÉPARTEMENT           REVENU
-------------- -------------------- -------------
Centre         VoitureLocation              157,000
Centre         VoitureVente                  74,000
Centre         VoitureVentes                 85,000
Centre         Total départements           316,000
Est            VoitureLocation              190,000
Est            VoitureVente                 115,000
Est            VoitureVentes                137,000
Est            Total départements           442,000
Ouest          VoitureLocation              183,000
Ouest          VoitureVente                  86,000
Ouest          VoitureVentes                 97,000
Ouest          Total départements           366,000
Total régions VoitureLocation               530,000
Total régions VoitureVente                  275,000
Total régions VoitureVentes                 319,000
Total régions Total départements          1,124,000
```

Explication de la ligne **SELECT DECODE(GROUPING (region), 1,'Total régions', region) AS Région.** La valeur de **region** est déterminée avec la fonction DECODE qui contient la fonction de groupes GROUPING. GROUPING retourne un 1 si la valeur de la ligne est un agrégat créé par ROLLUP ou CUBE, sinon elle retourne un 0. La fonction DECODE opère ensuite sur le résultat de la fonction de groupe GROUPING. Elle retourne le texte "Total régions" si elle reçoit un 1, et que la valeur de **region** (de la table) reçoit un 0. La valeur de la table sera une valeur telle que "Centre" ou un NULL. La seconde colonne "DEPARTEMENT" agit de la même manière.

5.2 Quand faut-il utiliser GROUPING?

La fonction GROUPING est très utile pour identifier les NULLs de la base de données, elle permet aussi de trier, classer les lignes de sous-totaux ainsi que le filtrage des résultats.
Il est également possible d'utiliser la clause HAVING avec une fonction de groupes GROUPING.

5.3 Fonction GROUPING_ID

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. Par exemple, une clause GROUP BY ayant quatre colonnes doit être analysée avec quatre fonctions GROUP BY. Quand il faut stocker le résultat de la requête dans des tables, comme avec les vues matérialisées (*materialized views*), pour les colonnes supplémentaires, il en résulte un gaspillage d'espace de stockage.

Pour résoudre ce problème, Oracle9i introduit la fonction GROUPING_ID. GROUPING_ID retourne un nombre unique qui permet de déterminer le niveau exact du GROUP BY. Pour chaque ligne, GROUPING_ID prend un "set" de 1 et 0.

Par exemple, si on utilise l'expression CUBE(a,b), les valeurs possibles seront:

Niveau d'agrégation	Vecteur de bit	GROUPING_ID
a, b	0 0	0
a	0 1	1
b	1 0	2
Grand Total	1 1	3

GROUPING_ID est particulièrement utile pour le *refresh* et *rewrite* des *materialized views* (exemple plus loin).

5.4 Fonction GROUP_ID

Les extensions ROLLUP et CUBE offrent plus de puissance et de flexibilité, ils donnent aussi des résultats plus complexes incluant des groupes dupliqués. La fonction GROUP_ID donne une idée de la duplication de ces groupes. Dans le cas ou il y aurait plusieurs jeux de lignes pour un niveau donné, GROUP_ID assigne la valeur 0 à toutes les lignes du premier jeu de lignes. Tous les autres jeux de données dupliquées pour un groupe particulier sont mis à une valeur plus grande, en débutant avec 1.

Requête générant la duplication de groupes

```
SELECT region, departement, SUM(revenu),
  GROUPING_ID(region, departement) GROUPING_ID, GROUP_ID()
  FROM ventes
  GROUP BY GROUPING SETS (region, ROLLUP(region,
    departement)) ;

REGION DEPARTEMENT     SUM(REVENU) GROUPING_ID GROUP_ID()
------ -------------- ----------- ----------- ----------
Centre VoitureLocation      157000           0          0
Centre VoitureVente          74000           0          0
Centre VoitureVentes         85000           0          0
Est    VoitureLocation      190000           0          0
Est    VoitureVente         115000           0          0
Est    VoitureVentes        137000           0          0
-- suite voir page suivante
```

```
-- suite
Ouest  VoitureLocation      183000          0          0
Ouest  VoitureVente          86000          0          0
Ouest  VoitureVentes         97000          0          0
                           1124000          3          0
Centre                      316000          1          0
Est                         442000          1          0
Ouest                       366000          1          0
Centre                      316000          1          1
Est                         442000          1          1
Ouest                       366000          1          1
```

Cette requête génère les groupes suivants: **(**region, departement**)**, **(**region**)**, **(**region**)**, et **()**. On s'aperçoit que **(**region**)** est répété deux fois.

Pour ne pas avoir de groupe dupliqué, il suffit d'ajouter la clause HAVING GROUP_ID()=0 à la requête, ainsi les trois dernières lignes n'apparaîtrons pas.

Note: pour la syntaxe utilisée pour GROUPINGS SETS, voir "Expression GROUPINGS SETS".

6 Expression GROUPING SETS

Dans une clause GROUP BY avec l'extension 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.

Exemple de GROUPING SETS

```
SELECT region, annee, departement,
 TO_CHAR(SUM(revenu), '9,999,999,999') Revenu
 FROM ventes
 GROUP BY GROUPING SETS ((region, annee, departement),
 (region, departement), (annee, departement)) ;

REGION     ANNEE    DEPARTEMENT      REVENU
---------- -------- ---------------- -------------
Centre     03.03.00 VoitureLocation          75,000
Centre     03.03.01 VoitureLocation          82,000
Centre     03.03.00 VoitureVente             74,000
Centre     03.03.01 VoitureVentes            85,000
Est        03.03.00 VoitureLocation          89,000
Est        03.03.01 VoitureLocation         101,000
Est        03.03.00 VoitureVente            115,000
Est        03.03.01 VoitureVentes           137,000
Ouest      03.03.00 VoitureLocation          87,000
Ouest      03.03.01 VoitureLocation          96,000
Ouest      03.03.00 VoitureVente             86,000
Ouest      03.03.01 VoitureVentes            97,000
Centre              VoitureLocation         157,000
Centre              VoitureVente             74,000
Centre              VoitureVentes            85,000
-- suite page suivante
```

Suite de l'exemple GROUPING SETS

```
Est                  VoitureLocation          190,000
Est                  VoitureVente             115,000
Est                  VoitureVentes            137,000
Ouest                VoitureLocation          183,000
Ouest                VoitureVente              86,000
Ouest                VoitureVentes             97,000
           03.03.00  VoitureLocation          251,000
           03.03.00  VoitureVente             275,000
           03.03.01  VoitureLocation          279,000
           03.03.01  VoitureVentes            319,000
```

Si on avait utilisé l'extension CUBE dans la requête précédente, on aurait eu 8 (2*2*2) groupes, alors que seulement trois groupes était demandé.

GROUPING SETS permet de définir plusieurs groupes dans la même requête. GROUP BY calcule tous les groupes spécifiés et les associe entre eux avec un UNION ALL. Par exemple, l'expression suivante:

```
GROUP BY GROUPING SETS (region, annee, departement)
```

Est équivalente à:

```
GROUP BY region
UNION ALL
GROUP BY annee
UNION ALL
GROUP BY departement
```

Expression GROUPING SETS et équivalence avec GROUP BY

Expression GROUPING SETS	Expression GROUP BY (équivalence)
GROUP BY GROUPING SETS(a, b, c)	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY c
GROUP BY GROUPING SETS(a, b, (b, c))	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY b, c
GROUP BY GROUPING SETS((a, b, c))	GROUP BY a, b, c
GROUP BY GROUPING SETS(a, (b), ())	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY ()
GROUP BY GROUPING SETS(a, ROLLUP(b, c))	GROUP BY a UNION ALL GROUP BY ROLLUP(b, c)

En l'absence d'un *optimizer*, une requête basée sur un UNION aurait besoin de plusieurs accès (*scans*) à la table de base, la table de **fait**. En utilisant l'expression GROUPING SETS, tous les groupes sont disponibles dans le même bloc de requête, et il n'y aura qu'un seul accès à la table de **fait**.

7 Répartition de groupes concaténés

La répartition de groupes concaténés (*Concatenated Groupings*) offre une manière intéressante de générer des groupes. Ces groupes sont simplement spécifiés en listant les différents jeux de groupes, cubes, et rollups, en les séparant par une virgule.

Exemple d'un jeu de groupes concaténés

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

Cette expression SQL définit les groupes suivants:

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

La concaténation de jeux de groupes est utile pour les raisons suivantes:

- facilité d'écriture de la requête (il n'y a pas besoin d'énumérer manuellement tous les groupes)
- utilisation par les applications (les applications OLAP génèrent du code SQL qui utilise des jeux de groupes)

8 Cubes hiérarchiques

Une des utilisations des groupes concaténés est la génération d'agrégats utilisés dans les données de cubes hiérarchiques (*hierarchical cube*). Un cube hiérarchique est un jeu de données dont les données sont agrégées avec un ROLLUP pour chacune de ces dimensions, et ces agrégations sont combinées au travers des autres dimensions. Un cube hiérarchique n'est pas la même chose qu'une agrégation de cube pour toutes les colonnes principales des dimensions. Un cube hiérarchique est un jeu de groupes plus petit qui inclus un jeu d'agrégations utilisé par les requêtes du *business intelligence*[2]. On peut générer toutes les agrégations pour un cube hiérarchique en utilisant des ROLLUP concaténés.

Exemple d'un cube hiérarchique simple (trois dimensions avec trois niveaux hiérarchiques)

Temps: année, trimestre, mois

Produit: catégorie, marque, article

Géographie: région, canton, ville

Pour les besoins du *business intelligence*, il faut calculer et stocker les agrégats pour les différentes combinaisons des trois dimensions. On peut utiliser ROLLUP sur chaque dimension pour spécifier les bons agrégats. Une fois que l'on a les bons agrégats pour chaque dimension, on les concatène les uns avec les autres. Ceci produira un cube hiérarchique.

Exemple de GROUPING SETS concaténés

```
GROUP BY ROLLUP(annee, trimestre, mois),
  ROLLUP(categorie, marque, article),
  ROLLUP(region, mois, ville)
```

[2] ensemble des technologies permettant en fin de chaîne d'apporter une aide à la décision

Résultat de l'exemple précédent (génère quatre groupes pour chaque dimension)

```
ROLLUP par Temps ROLLUP par Produit         ROLLUP par Geogra
---------------- ------------------------ ------------------
annee,trim,mois  categorie,marque,article region,canton,ville
annee,trimestre  categorie,marque         region
annee            categorie                region
temps (tous)     produit (tous)           géographie (tous)
```

Le produit cartésien de l'exemple ci-dessus génère 64 (4x4x4) groupes d'agrégation pour les besoins du cube. Il y a en effet un avantage à utiliser trois ROLLUP au lieu d'utiliser 64 jeux de groupes. Il y a aussi moins de risque d'erreur de développement et de maintenance.

9 Combinaison avec les vues matérialisées (*Materialized Views*)

L'utilisation de vues matérialisées est une technique importante dans les datawarehouses pour maximiser les performances sur les requêtes. Dans Oracle9i, les extensions du GROUP BY, CUBE, ROLLUP et GROUPING SETS peuvent être utilisées dans la création et la maintenance des vues matérialisées. Cette option permet d'avoir plusieurs niveaux de groupes contenus dans une unique vue matérialisée. Le fait que l'on puisse remplacer plusieurs (peut-être des centaines) vues matérialisées de petites tailles, par un petit nombre de vues matérialisées de niveau multiple, la gestion de la base de données s'en trouve simplifiée. De plus, la rapidité de création et de maintenance des vues matérialisées est augmentée.

Exemple de vue matérialisée avec l'extension ROLLUP

```
CREATE MATERIALIZED VIEW mv AS
SELECT prod_ligne, prod_famille, prod_id, region, canton,
       ville, SUM(ventes) AS Tot_ventes,
   GROUPING_ID(prod_ligne, prod_famille, prod_id, region,
               canton, ville)
FROM ventes
GROUP BY prod_ligne, ROLLUP(prod_famille, prod_id), region,
   ROLLUP(canton, ville);
```

L'expression SQL ci-dessus calcule 9 sous-totaux a travers la hiérarchie des dimensions Produit et Geographie et les stocke dans une seule vue matérialisée. Les sous-totaux sont:

- prod_ligne, prod_famille, prod_id, region, canton, ville
- prod_ligne, prod_famille, prod_id, region, canton
- prod_ligne, prod_famille, prod_id, region
- prod_ligne, prod_famille, region, canton, ville
- prod_ligne, prod_famille, region, canton
- prod_ligne, prod_famille, region
- prod_ligne, region, canton, ville

- prod_ligne, region, ville
- prod_ligne, region

Il est évident que le fait de créer une seule vue matérialisée à la place d'en créer 9 autres contenant un niveau unique d'agrégation (GROUP BY sans extension), est particulièrement profitable pour la création et de la maintenance. Avec une situation ou l'on aurait plusieurs vues matérialisées contenant un niveau unique d'agrégation, il y aurait d'autant plus d'accès à la table de **fait**. Un autre avantage vient du fait que l'*optimizer* détecte si certain jeu de groupes peuvent être ré-utilisés afin de ne pas devoir à nouveau trier les données et à nouveau les calculer, de cette manière cela maximise les performances.

CHAPITRE 2: Fonctions analytiques

Oracle a amélioré le langage d'analyse SQL en introduisant une famille de fonctions SQL analytiques. Ces fonctions permettent de calculer (définitions, voir ci-dessous):

- Rankings et percentiles
- Moving window calculations
- Lag/lead analyse
- First/last analyse
- Statistiques de régression linéaire

Une autre amélioration du code SQL, est l'inclusion de l'expression CASE. Cette expression fournit le "si – alors" (*if-then*) très utiles dans certaines situations.

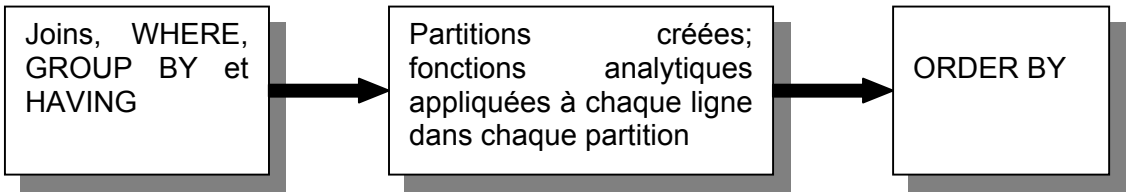
De plus les fonctions analytiques peuvent être parallélisées, ce qui augmente très nettement les performances.

Bref descriptif des différents types de fonctions analytiques

Type	Utiliser pour
Ranking	Calcul de rangs(*ranks*), percentiles[3], et n-tiles de valeurs dans un jeu de résultat. Supporte les fonctions: RANK, DENSE_RANK, PERCENT_RANK, CUME_DIST, et NTILE. Répond aux questions du style "montrer les 10 premiers et 10 derniers vendeurs par région" ou "montrer pour chaque région, les vendeurs qui ont fait plus de 25% des ventes".
Windowing	Calcul cumulatif et déplacement d'agrégats. Fonctionne avec les fonctions: SUM, AVG, MIN, MAX, COUNT, VARIANCE, STDDEV, FIRST_VALUE, LAST_VALUE, et les nouvelles fonctions de statistique. Répond aux questions du style "Quel est le *moving average* [4]de la 13ème semaine?" ou "quelle est la somme cumulée des ventes pour chaque région?".
Reporting	Calcul de comparaisons de valeurs pour différents niveaux d'agrégation, par exemple, les parts de marché. Fonctionne avec les fonctions: SUM, AVG, MIN, MAX, COUNT (avec/sans DISTINCT), VARIANCE, STDDEV, RATIO_TO_REPORT, et les nouvelles fonctions de statistique.
LAG/LEAD	Trouve une valeur dans une ligne, un nombre spécifique de lignes à partir de la ligne courante.
FIRST/LAST	(9i) Première ou dernière valeur dans un groupe ordonné.
Linear Regression	Calcul de ligne de régression et autres statistiques (pente, interception, et autres).
Inverse Percentile	(9i) La valeur dans un jeu de données correspondant à un percentile précis.
Hypothetical Rank and Distribution	(9i) Le rang (*rank*) ou le percentile qu'une ligne aurait, si elle avait été insérée dans le jeu de données spécifique.

[3] Centième partie d'une suite ordonnée de données statistiques. Synonyme de centile.
[4] technique d'analyse permettant de créer des graphes indiquant les fluctuations de prix

Ordre des processus durant l'exécution des fonctions analytiques



Concepts essentiels des fonctions analytiques:

- Ordre de traitement

  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. (voir schéma ci-dessus).

- Partitions de jeu de résultat

  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'ont aucuns liens avec les tables partitionnées d'Oracle**. Dans ce chapitre, le terme de partition fait uniquement référence aux fonctions analytiques. Les partitions sont créées seulement après que les groupes aient été défini 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 quelles colonnes ou expressions. 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.

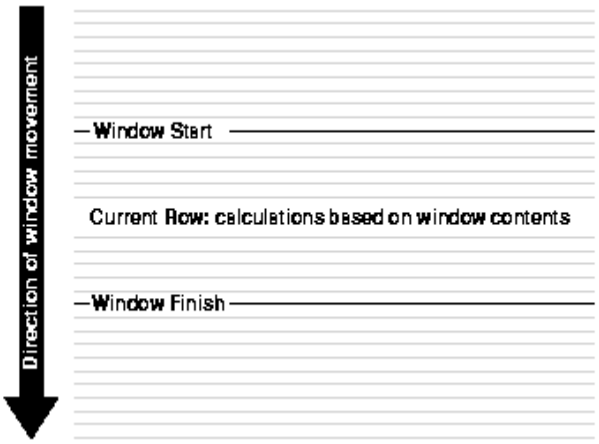
- Fenêtre (*Window*)

  Pour chaque ligne dans une partition, on peut définir une fenêtre (plage) non fixe de données (*sliding window*). Cette fenêtre détermine **la rangée** de 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 logique tel que le temps (*time*). La fenêtre possède une ligne de commencement et de fin. Selon de sa définition, une fenêtre définie pour une fonction de somme cumulative a sa ligne de commencement fixée sur la première ligne de la partition, et sa ligne de fin peut "glisser" depuis le point de départ jusqu'à la dernière ligne de la partition. Par contre, une fenêtre définit pour un *moving average*, a un point de départ et de fin, ce qui maintient une rangée constante physique ou logique.

  Une fenêtre peut être aussi grande que toutes les lignes d'une partition ou grande comme une portion de fenêtre, de la taille d'une ligne. Avec une fonction utilisant une fenêtre (*window*), la ligne courante est inclue dans le calcul, dès lors, il faut le spécifier (*n*-1) quand on traite avec *n* articles.

- Ligne courante

  Chaque calcul entreprit avec une fonction analytique est basée sur une ligne courante pour une partition. La ligne courante sert de point de référence pour déterminer le commencement et la fin de la fenêtre. Par exemple, un *centered moving average* [5]pourrait être défini avec une fenêtre qui contiendrait la ligne courante, les six lignes précédentes, ainsi que les six suivantes. Ceci créerait une fenêtre dite coulissante (*sliding window*) de 13 lignes (voir schéma ci-dessous).



Schéma, documentation officielle d'Oracle

10 Fonctions de classement (*Ranking Functions*)

Une fonction de classement calcule le rang d'un enregistrement par rapport aux autres enregistrements, basé sur les valeurs du jeu de mesures. Les différents types de fonctions de classement sont:

- RANK et DENSE_RANK
- CUME_DIST et PERCENT_RANK
- NTILE
- ROW_NUMBER

10.1 RANK et DENSE_RANK

RANK et DENSE_RANK permettent de classer (ranger) des articles dans un groupe. Par exemple, trouver le top des trois produits les plus vendu dans le canton de Neuchâtel l'année passée.

Syntaxe de RANK et DENSE_RANK

```
RANK () OVER ( [query_partition_clause] order_by_clause )
DENSE_RANK () OVER ( [query_partition_clause]
                     order_by_clause )
```

[5] *Moving average* est à la base de l'anlyse technique, cela permet en autre d'aplanir les fluctuations aléatoires des marché financiers.

La différence entre RANK et DENSE_RANK se trouve dans le fait que la fonction DENSE_RANK ne laisse aucun trous (intervalle) dans une série de rangée lorsque celles-ci sont très proches. Par exemple, si on utilise DENSE_RANK pour classer deux personnes dans une compétition qui sont très proches au niveau du temps, on dira qu'ils occupent les deux la même place (ex. les deux seront classés 3ème), la personne suivante occupera la 4ème place, malgré qu'elle soit la 5ème personne. RANK agira de même, sauf que la personne suivante sera classée en 5ème position.

Voici un exemple d'application de la fonction RANK qui affiche les 3 premières ventes pour la vente de voiture et par région de la table VENTES.

Pour avoir le Top N Ranking

```
SELECT * FROM
  (SELECT region, TO_CHAR(SUM(revenu), '9,999,999,999')
    Revenu,
     RANK() OVER (ORDER BY SUM(revenu) DESC) AS REGION_RANG
   FROM ventes
   WHERE departement='VoitureVentes'
   GROUP BY region)
WHERE REGION_RANG <=3;

REGION     REVENU          REGION_RANG
---------- -------------- -----------
Est               137,000           1
Ouest              97,000           2
Centre             85,000           3
```

11 CUME_DIST

La fonction CUME_DIST (définie comme l'inverse de *percentile* dans certains livres de statistiques) calcule la position d'une valeur relative spécifique par rapport à un jeu de valeurs. L'ordre peut être ascendant (par défaut) ou descendant. La rangée de valeurs pour CUME_DIST est plus grande que 0 jusqu'à 1. Pour calculer le CUME_DIST d'une valeur x dans un jeu S de taille N, utiliser la formule suivante:

```
CUME_DIST(x) = nombre de valeurs dans S venant avant et
               incluant x, dans l'ordre spécifié par
               ORDER / N
```

Syntaxe de CUME_DIST

```
CUME_DIST ( ) OVER ( [query_partition_clause]
                     order_by_clause )
```

L'exemple suivant montre une répartition cumulative des ventes par région pour chaque année

```
SELECT annee, region,
   TO_CHAR(SUM(revenu), '9,999,999,999') Revenu,
   CUME_DIST() OVER ( PARTITION BY annee ORDER BY
                      SUM(revenu))
   AS CUME_DIST_Revenu
   FROM ventes
   WHERE ventes.annee IN (TO_DATE('03-03-2001','DD-MM-YYYY'),
                          TO_DATE('03-03-2000','DD-MM-YYYY'))
   GROUP BY annee, region ;

ANNEE    REGION     REVENU          CUME_DIST_REVENU
-------- ---------- --------------- ----------------
03.03.00 Centre             149,000        ,333333333
03.03.00 Ouest              173,000        ,666666667
03.03.00 Est                204,000                 1
03.03.01 Centre             167,000        ,333333333
03.03.01 Ouest              193,000        ,666666667
03.03.01 Est                238,000                 1
```

12 PERCENT_RANK

PERCENT_RANK est similaire à CUME_DIST, mais elle utilise une valeur de rangée (*rank*) plutôt que l'addition de lignes pour son compteur. Donc, elle retourne le rang en % de la valeur cumulative pour un groupe de valeurs. Cette fonction est disponible dans la plupart des tableurs du marché. Le PERCENT_RANK d'une ligne est calculé ainsi:

```
(rang d'une ligne dans sa partition -1) / (nombre de lignes
dans la partition -1)
```

PERCENT_RANK retourne des valeurs dans une rangée de zéro à un. La ligne(s) avec une rangée de 1 aura un PERCENT_RANK de zéro.

Syntaxe de PERCENT_RANK

```
PERCENT_RANK ( ) OVER ( [query_partition_clause]
                        order_by_clause )
```

13 NTILE

NTILE permet de facilement calculer des *tertiles*, *quartiles*, *deciles* et autres statistiques récapitulatives. Cette fonction divise une partition ordonnée en un nombre spécifique de groupes appelés **buckets,** et elle attribue un numéro (*bucket number*) à chaque ligne de la partition. NTILE permet un calcul 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. Par exemple, si on a 100 lignes dans une partition et que l'on demande la fonction NTILE avec quatre **buckets**, 25 lignes se feront attribuer la valeur 1, 25 lignes auront la valeur 2, et ainsi de suite.

Si le nombre de lignes dans la partition ne se divise pas de manière égale, alors le nombre de lignes attribué pour chaque **bucket** différera de un tout au plus. Les lignes qui sont en plus, seront distribuées de façon à avoir une ligne pour chaque **bucket** partant depuis le plus petit nombre (*bucket number*). Par exemple, s'il y a 103 lignes dans une partition avec une fonction NTILE(5), les premières 21 lignes seront dans le premier **bucket**, les prochaines 21 dans le second **bucket**, les prochaines 21 dans le troisième, les 20 prochaines dans le quatrième **bucket** et les dernières 20 dans le cinquième **bucket**.

Syntaxe de la fonction NTILE

```
NTILE ( expr ) OVER ( [query_partition_clause]
                      order_by_clause )
```

Le N dans NTILE (N) peut être une constante (par exemple, 5) ou une expression.

L'exemple suivant attribue pour chaque total des ventes/année dans 2 buckets

```
SELECT annee AS ANNEE,
   TO_CHAR(SUM(revenu), '9,999,999,999') Revenu,
   NTILE(2) OVER (ORDER BY SUM(revenu)) AS TILE2
   FROM ventes
   GROUP BY annee ;

ANNEE    REVENU              TILE2
-------- -------------- ----------
03.03.00        526,000          1
03.03.01        598,000          1
03.03.02        598,006          2
03.03.03        598,012          2
```

14 ROW_NUMBER

La fonction ROW_NUMBER attribue un nombre unique (séquentielle, à partir de 1, définit par ORDER BY) pour chaque ligne dans la partition.

Syntaxe de la fonction ROW_NUMBER

```
ROW_NUMBER ( ) OVER ( [query_partition_clause]
                      order_by_clause )
```

Exemple de la fonction ROW_NUMBER

```
SELECT region, annee,
   TO_CHAR(SUM(revenu), '9,999,999,999') Revenu,
   ROW_NUMBER() OVER (ORDER BY TRUNC(SUM(revenu), -3) DESC)
     AS ROW_NUMBER
   FROM ventes
   GROUP BY annee, region ;

REGION     ANNEE    REVENU          ROW_NUMBER
---------- -------- -------------- ----------
Est        03.03.01         238,000          1
Est        03.03.02         238,002          2
Est        03.03.03         238,004          3
Est        03.03.00         204,000          4
Ouest      03.03.01         193,000          5
Ouest      03.03.02         193,002          6
Ouest      03.03.03         193,004          7
Ouest      03.03.00         173,000          8
Centre     03.03.01         167,000          9
Centre     03.03.02         167,002         10
Centre     03.03.03         167,004         11
Centre     03.03.00         149,000         12
```

15 Fonctions d'agrégations de fenêtres (*Windowing Aggregate Functions*)

Les fonctions de fenêtres peuvent être utilisées dans les calculs cumulatifs, de mouvements (*moving*), et d'agrégations centrées (*centered aggregates*). Elles retournent une valeur pour chaque ligne de la table, laquelle dépend des autres lignes dans la fenêtre correspondante. Ces fonctions incluent les fonctions *moving sum*[6], *moving average*[7], *moving min/max*[8], *cumulative sum*[9], en plus des autres fonctions de statistiques. Elles peuvent être utilisées seulement avec les clauses SELECT et ORDER BY. Deux autres fonctions sont disponibles: FIRST_VALUE, laquelle retourne la première valeur de la fenêtre; et LAST_VALUE, laquelle retourne la dernière valeur de la fenêtre. Ces fonctions fournissent un accès à plus d'une ligne d'une table sans auto-jointure.

6

7

8

9 somme cumulative

Syntaxe des fonctions de fenêtre

```
{SUM|AVG|MAX|MIN|COUNT|STDDEV|VARIANCE|FIRST_VALUE|LAST_VALUE}
  ({value expression1 | *}) OVER
    ([PARTITION BY value expression2[,...])
      ORDER BY value expression3 [collate clause>]
                [ASC| DESC] [NULLS FIRST | NULLS LAST] [,...]
{ ROWS | RANGE }
{ BETWEEN
  { UNBOUNDED PRECEDING
  | CURRENT ROW
  | value_expr { PRECEDING | FOLLOWING }
  }
  AND
  { UNBOUNDED FOLLOWING
  | CURRENT ROW
  | value_expr { PRECEDING | FOLLOWING }
  }
| { UNBOUNDED PRECEDING
  | CURRENT ROW
  | value_expr PRECEDING
  }
}
```

15.1 Traitement des NULLs comme entrée pour les fonctions de fenêtres

La sémantique des NULLs dans les fonctions de fenêtres, correspondent à la sémantique des NULLs dans les fonctions SQL d'agrégation. D'autres sémantiques peuvent être obtenues avec les fonctions définies par les utilisateurs (*user-defined*), ou par les expressions DECODE ou CASE dans les fonctions de fenêtres.

15.2 Fonctions de fenêtres avec offset[10] logique (logical offset)

Un *offset* logique peut être spécifié avec des constantes tels que RANGE 10 PRECEDING, ou une expression qui évalue par rapport à une constante, ou par rapport à un intervalle comme RANGE INTERVAL N DAY/MONTH/YEAR PRECEDING ou une expression qui évalue par rapport à un intervalle. Avec un *offset* logique, il ne peut y avoir qu'une seule expression dans l'expression ORDER BY d'une fonction, avec un type compatible NUMERIC si *l'offset* est numérique, ou DATE si un intervalle est spécifié. Exemple d'une fonction cumulative d'agrégat:

```
SELECT      region,      annee,      TO_CHAR(SUM(revenu),
'9,999,999,999')  Revenu,  TO_CHAR(SUM(SUM(revenu))  OVER
(PARTITION  BY  region  ORDER  BY  region,  annee  ROWS
UNBOUNDED PRECEDING), '9,999,999,999') AS CUM_VENTES
   FROM ventes
   GROUP BY region, annee
   ORDER BY region, annee ;
```

[10] excentrage

Résultat de la fonction cumulative d'agrégat

```
REGION     ANNEE    REVENU          CUM_VENTES
---------- -------- -------------- --------------
Centre     03.03.00         149,000        149,000
Centre     03.03.01         167,000        316,000
Centre     03.03.02         167,002        483,002
Centre     03.03.03         167,004        650,006
Est        03.03.00         204,000        204,000
Est        03.03.01         238,000        442,000
Est        03.03.02         238,002        680,002
Est        03.03.03         238,004        918,006
Ouest      03.03.00         173,000        173,000
Ouest      03.03.01         193,000        366,000
Ouest      03.03.02         193,002        559,002
Ouest      03.03.03         193,004        752,006
```

Dans cet exemple, la fonction analytique SUM défini, pour chaque ligne, une fenêtre qui commence au début de la partition (UNBOUNDED PRECEDING) et termine, par défaut, à la ligne courante.

*15.3 Exemple de fonction d'agrégat "glissant" (*Moving Aggregate Function*)*

```
SELECT region, annee,
   TO_CHAR(SUM(revenu), '9,999,999,999') Revenu,
   TO_CHAR(AVG(SUM(revenu)) OVER (ORDER BY region, annee
   ROWS 2 PRECEDING), '9,999,999,999') AS MOVING_3_YEAR_AVG
   FROM ventes
   WHERE region='Centre'
   GROUP BY region, annee
   ORDER BY region, annee ;

REGION     ANNEE    REVENU          MOVING_3_YEAR_
---------- -------- -------------- --------------
Centre     03.03.00         149,000        149,000
Centre     03.03.01         167,000        158,000
Centre     03.03.02         167,002        161,001
Centre     03.03.03         167,004        167,002
```

Cet exemple de fenêtre basée sur le temps, montre, pour une région, la moyenne mobile (*moving average*) de ventes pour l'année courante et les trois années précédentes.

16 FIRST_VALUE et LAST_VALUE

Les fonctions FIRST_VALUE et LAST_VALUE autorisent la sélection des premières et dernières lignes de la fenêtre. Ces lignes sont particulièrement utiles parce qu'elles sont souvent utilisées comme lignes de base dans les calculs. Par exemple, avec une partition contenant les données de ventes triées par jour, on pourrait demander "De combien étaient les ventes journalières comparées au premier jour de vente (FIRST_VALUE) de la saison?". Autre exemple, on désire savoir, pour un jeu de lignes, l'ordre des ventes croissantes, "Quelle était la taille en pourcentage pour chaque vente de la région, comparée à la plus grande vente (LAST_VALUE) de la région?".

17 Fonctions de rapport d'agrégat

Après qu'une requête ait été traitée, les valeurs agrégées, comme le nombre de lignes traitées ou la valeur moyenne pour une colonne peut facilement être calculée dans une partition et être mis à disposition d'autres fonctions de rapport. Les fonctions de rapport d'agrégat retournent la même valeur d'agrégation pour chaque ligne dans une partition. Leur comportement vis-à-vis des NULLs est le même que pour les fonctions SQL d'agrégation.

Syntaxe de fonction de rapport d'agrégat

```
{SUM | AVG | MAX | MIN | COUNT | STDDEV | VARIANCE}
   ([ALL | DISTINCT] {value expression1 | *})
       OVER ([PARTITION BY value expression2[,...]])
```

De plus, les conditions suivantes s'appliquent:

- Un astérisque (*) est seulement autorisé dans COUNT(*)
- DISTINCT est supporté seulement si la fonction d'agrégation le permet
- *valeur expression1* et *valeur expressio2* peuvent être n'importe quelle expression impliquant des colonnes de références ou d'agrégats.
- La clause PARTITION BY défini les groupes sur lesquels les fonctions de fenêtre (*windowing functions*) seraient calculées. Si la clause PARTITION BY est absente, alors la fonction est calculée sur tout le jeu de résultat.

Les fonctions de rapport peuvent apparaître seulement dans la clause SELECT ou ORDER BY. Le bénéfice majeur des fonctions de rapport est leurs capacités à passer plusieurs données dans un block unique de requête et d'augmenter les performances de la requête. Les requêtes telles que "Compter le nombre de vendeurs avec des ventes de plus de 10% des ventes de la ville" ne requièrent pas de jointures entre les blocks séparés.

Exemple de question: "Pour chaque département, trouver la région dans laquelle il y a eu le maximum de ventes"

```
SELECT departement, region, revenu FROM
   (SELECT departement, region, SUM(revenu) AS Revenu,
    MAX(SUM(revenu)) OVER (PARTITION BY departement) AS
MAX_REG_VENTES
    FROM ventes
    GROUP BY departement, region)
    WHERE revenu=MAX_REG_VENTES;

-- Résultat de la sous-requête (inner)
DEPARTEMENT          REGION         REVENU MAX_REG_VENTES
-------------------- ---------- ---------- --------------
VoitureLocation      Centre         321003         392003
VoitureLocation      Est            392003         392003
VoitureLocation      Ouest          375003         392003
VoitureVente         Centre          74000         115000
VoitureVente         Est            115000         115000
VoitureVente         Ouest           86000         115000
VoitureVentes        Centre         255003         411003
VoitureVentes        Est            411003         411003
VoitureVentes        Ouest          291003         411003

-- Résultat de la requête en entier
DEPARTEMENT          REGION         REVENU
-------------------- ---------- ----------
VoitureLocation      Est            392003
VoitureVente         Est            115000
VoitureVentes        Est            411003
```

18 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.

Syntaxe

```
RATIO_TO_REPORT ( expr ) OVER ( [query_partition_clause] )
```

Selon la syntaxe, ceci s'applique pour:

- *expr* peut être n'importe quelle expression impliquant des colonnes de références ou d'agrégats.
- 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.

Exemple de ventes par département en utilisant RATIO_TO_REPORT

```
SELECT departement,
   TO_CHAR(SUM(revenu), '9,999,999,999') AS Revenu,
   TO_CHAR(SUM(SUM(revenu)) OVER (), '9,999,999,999') AS
TOTAL_VENTES,
   TO_CHAR(RATIO_TO_REPORT(SUM(revenu)) OVER (), '9.999')
   AS RATIO_TO_REPORT
   FROM ventes
   WHERE annee=TO_DATE('03/03/2002','DD/MM/YYYY')
   GROUP BY departement ;

DEPARTEMENT     REVENU     TOTAL_VENTES  RATIO_TO_REPORT
--------------- ---------- ------------- ---------------
VoitureLocation    279,003      598,006   .467
VoitureVentes      319,003      598,006   .533
```

19 Fonctions LAG/LEAD

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 (*target*) à partir de la ligne courante. Etant donné que les fonctions fournissent un accès à plus d'une ligne d'une table en même temps sans faire d'auto jointure (*self join*), elles peuvent améliorer la vitesse de traitement. La fonction LAG fournit un accès à une ligne à un *offset* donné avant la position courante, et la fonction LEAD fournit un accès à une ligne à un *offset* après la position courante.

Syntaxe pour LAG/LEAD

```
{LAG | LEAD} ( value_expr [, offset] [, default] )
   OVER ( [query_partition_clause] order_by_clause )
```

offset est un paramètre optionnel et par défaut c'est 1. *default* est un paramètre optionnel est c'est la valeur retournée si *offset* tombe en dehors de la limite de la table ou de la partition.

Exemple LAG/LEAD

```
SELECT annee, TO_CHAR(SUM(revenu), '9,999,999,999') AS
  Revenu, TO_CHAR(LAG(SUM(revenu),1) OVER (ORDER BY
    annee), '9,999,999,999') AS LAG1,
  TO_CHAR(LEAD(SUM(revenu),1) OVER (ORDER BY annee),
    '9,999,999,999') AS LEAD1
FROM ventes
WHERE annee>=TO_DATE('03/03/2000','DD/MM/YYYY')
AND annee<=TO_DATE('03/03/2003','DD/MM/YYYY')
GROUP BY annee ;

ANNEE    REVENU          LAG1            LEAD1
-------- --------------- --------------- ---------------
03.03.00         526,000                         598,000
03.03.01         598,000         526,000         598,006
03.03.02         598,006         598,000         598,012
03.03.03         598,012         598,006
```

20 Fonctions FIRST/LAST

Les fonctions d'agrégat FIRST/LAST permettent de retourner le résultat d'un agrégat appliqué sur un jeu de lignes qui range/classe (*rank*) comme premier ou dernier dans un ordre spécifique donné. FIRST/LAST permet d'ordonner sur une colonne A mais en retournant un résultat d'un agrégat appliqué sur une colonne B. C'est un plus car cela évite une auto jointure ou une sous-requête, ce qui améliore les performances. Ces fonctions commencent avec une fonction subsidiaire, laquelle est une fonction d'agrégat habituelle (MIN, MAX, SUM, AVG, COUNT, VARIANCE, STDDEV) qui produit la valeur de retour. La fonction subsidiaire est exécutée sur le jeu de lignes (1 ou plusieurs lignes) qui range/classe comme premier ou dernier, respectant l'ordre de spécification pour retourner une valeur unique.

Pour spécifier un ordre utilisé aux travers de chaque groupe, les fonctions FIRST/LAST ajoutent une nouvelle clause commençant avec le mot KEEP.

Syntaxe FIRST/LAST

```
aggregate_function KEEP
( DENSE_RANK LAST ORDER BY
  expr [ DESC | ASC ] [NULLS { FIRST | LAST }]
  [, expr [ DESC | ASC ] [NULLS { FIRST | LAST }]]...
)
[OVER query_partitioning_clause]
```

La clause ORDER BY peut prendre de nombreuses d'expressions.

Exemple FIRST/LAST

L'exemple qui suit permet de comparer un prix minimum et une liste de prix de produits. Pour chaque produit, il retourne:

- Liste des prix du produit avec le prix le plus bas (LP_OF_LO_MINP)
- Le prix minimum le plus bas (LO_MINP)
- Liste des prix du produit avec le prix le plus haut (LP_OF_HI_MINP)
- Le prix maximum le plus haut (HI_MINP)

```
SELECT prod_sous_cat, MIN(prod_liste_prix)
 KEEP (DENSE_RANK FIRST ORDER BY (prod_min_prix))
 AS LP_OF_LO_MINP,
MIN(prod_min_prix) AS LO_MINP,
MAX(prod_liste_prix) KEEP (DENSE_RANK LAST ORDER BY
   (prod_min_prix)) AS LP_OF_HI_MINP,
MAX(prod_min_prix) AS HI_MINP
FROM produits GROUP BY prod_sous_cat;
```

PROD_SOUS_CAT	LP_OF_LO_MINP	LO_MINP	LP_OF_HI_MINP	HI_MINP
baskets	140	100	160	120
bottes	100,5	82,5	120,5	102,5
ceintures	39	22	59	42
chaussettes	8,5	4,5	10,5	6,5
gants	28	15,8	48	35,8
jeans	110	79,9	130	99,9
jupes	55	35,5	75	55,5
pulls	88,4	66,5	108,4	86,5
shorts	72,5	29,8	72,5	49,8
vestes	330	245	350	265

21 FIRST/LAST comme rapport d'agrégats

Il est possible d'utiliser FIRST/LAST comme fonctions de rapport d'agrégats. Par exemple: trouver la liste des prix de produits particuliers et les comparer à la liste des prix de produits se trouvant dans une catégorie se situant en-dessous de la leurs avec le prix minimum le plus haut et le plus bas.

Avec les fonctions FIRST et LAST utilisées comme rapport d'agrégats, il est par exemple facile d'avoir la réponse à la question "Que représente ce salaire en % par rapport au plus haut salaire?"

22 Fonctions de régression linéaire

Les fonctions de régressions supportent l'ajustement d'une ligne de régression de la méthode des moindres carrés (*ordinary-least-squares*) pour un jeu de nombres pairs. On peut les utiliser comme fonctions d'agrégat, ou comme fenêtre (*windowing*), ou comme fonctions de rapport (*reporting*).

Ces fonctions sont:

- REGR_COUNT
- REGR_AVGX
- REGR_AVGY
- REGR_SLOPE
- REGR_INTERCEPT
- REGR_R2
- REGR_SXX
- REGR_SYY
- REGR_SXY

Oracle applique la fonction à un jeu de pairs (*e1, e2*) après avoir éliminé toutes les pairs dans lesquelles il y avait un NULL pour *e1* ou *e2*. *e1* est interprétée comme une valeur de la variable dépendante (une "valeur y"), et *e2* est interprétée comme une valeur de la variable indépendante (une "valeur x"). Les deux expressions doivent être des nombres.

Les fonctions de régressions sont toutes calculées simultanément durant un seul passage à travers les données. Elles sont fréquemment combinées avec les fonctions COVAR_POP, COVAR_SAMP, et CORR.

22.1 REGR_COUNT

REGR_CCOUNT retourne le nombre de pairs non nulles qui vont avec la ligne de régression. Si appliquée à un jeu vide (ou s'il n'y pas de pairs (*e1, e2*) ou *e1* ou soit *e2* est nul), la fonction retourne 0.

22.2 REGR_AVGY et REGR_AVGX

REGR_AVGY et REGR_AVGX calculent les moyennes de la variable dépendante et de la variable indépendante de la ligne de régression. REGR_AVGY calcule la moyenne de son premier argument (*e1*) après avoir éliminé les pairs (*e1, e2*) ou *e1* ou *e2* est nul. De façon similaire, REGR_AVGX calcule la moyenne de son second argument (*e2*) après élimination des nuls. Les deux fonctions retournent NULL si elles sont appliquées pour un jeu vide.

22.3 REGR_SLOPE et REGR_INTERCEPT

La fonction REGR_SLOPE calcule la pente (*slope*) de la ligne de régression qui convient aux pairs non nuls (*e1, e2*).

La fonction REGR_INTERCEPT calcule l'interception de *y* de la ligne de régression. REGR_INTERCEPT retourne NULL chaque fois que les moyennes de la pente ou de régression sont NULL.

22.4 REGR_R2

La fonction REGR_R2 calcule le coefficient de corrélation (R-carré) pour la ligne de régression.
REGR_R2 retourne des valeurs entre 0 et 1 quand la ligne de régression est définie (la pente de la ligne n'est pas nulle), et qu'elle retourne NULL autrement. Plus la valeur est proche de 1, plus la ligne de régression convient à la donnée.

22.5 REGR_SXX, REGR_SYY, et REGR_SXY

Les fonctions REGR_SXX, REGR_SYY, et REGR_SXY sont utilisées dans divers calculs statistiques de diagnostique pour l'analyse de régression. Après avoir éliminé les pairs (*e1, e2*) ou e1, ou e2 ne sont pas nul, ces fonctions font les calculs suivants:

- REGR_SXX: REGR_COUNT(e1, e2) * VAR_POP(e2)
- REGR_SYY: REGR_COUNT(e1, e2) * VAR_POP(e1)
- REGR_SXY: REGR_COUNT(e1, e2) * COVAR_POP(e1, e2)

23 Fonctions de percentile inverse (*Inverse Percentile*)

En utilisant la fonction CUME_DIST, il est possible de trouver la distribution cumulative (percentile) d'un jeu de données. Toutefois, l'opération inverse (trouver quelle valeur calcule pour un certain percentil) n'est ni facile à faire, ni calculé de manière efficace. Pour surmonter cette difficulté, Oracle a introduit les fonctions PERCENTILE_CONT et PERCENTILE_DISC. Ces dernières peuvent être utilisées comme des fonctions de rapport de fenêtre (*window reporting*) ou comme des fonctions d'agrégation.

Ces fonctions ont besoin d'un ordre de tri et un paramètre qui prend une valeur pour le percentil entre 0 et 1. L'ordre de tri est traité par l'emploi d'une clause ORDER BY avec une expression. Lorsqu'elle est utilisée comme une fonction d'agrégation simple, elle retourne une seule valeur pour chaque jeu ordonné.

PERCENTILE_CONT est une fonction continue[11] (*continuous function*) calculée par interpolation. PERCENTILE_DISC est une fonction en escalier (*step function*) qui suppose des valeurs discrètes. Comme les autres agrégats, ces deux fonctions opèrent sur un groupe de lignes dans une requête groupée, mais avec les différences suivantes:

- Elles requièrent un paramètre entre 0 et 1 (inclusivement). Une valeur de paramètre en dehors de la rangée donnera une erreur. Ce paramètre doit être spécifié comme une expression qui est évaluée par rapport à une constante.
- Elles requièrent un ordre de tri. L'ordre de tri est une clause ORDER BY avec une seule expression. Les expressions multiples ne sont pas autorisées.

[11] fonction dont la variable est égale à x et qui reçoit un changement f(x) plus petit que sa valeur

Syntaxe normale d'agrégat

```
[PERCENTILE_CONT | PERCENTILE_DISC]( constant expression )
    WITHIN GROUP ( ORDER BY single order by expression
[ASC|DESC] [NULLS FIRST| NULLS LAST])
```

Exemple simple de percentile inverse

La fonction PERCENTILE_DISC(x) est calculée en scannant vers le haut les valeurs de CUME_DIST dans chaque groupe jusqu'à ce qu'il trouve la première valeur qui est plus grande ou égale à x, ou x est la valeur de percentile. Dans l'exemple suivant, pour la valeur de PERCENTILE_DISC(0.5), le résultat est 2500:

```
SELECT PERCENTILE_DISC(0.5) WITHIN GROUP
 (ORDER BY cli_credit_limite) AS perc_disc,
 PERCENTILE_CONT(0.5) WITHIN GROUP
 (ORDER BY cli_credit_limite) AS perc_cont
FROM clients
WHERE cli_ville IN ('Neuchâtel', 'Peseux', 'Yverdon') ;

PERC_DISC  PERC_CONT
---------- ----------
      2500       2650
```

Le résultat de PERCENTILE_CONT est calculé par interpolation linéaire entre les lignes après les avoir ordonnées. Pour calculer PERCENT_CONT(x), il faut en premier calculer le nombre de ligne = RN = (1+(x*(*n*-1))), ou *n* est le nombre de lignes dans le groupe et x est la valeur de percentile. Le résultat final de la fonction d'agrégat est calculé par interpolation linéaire entre les valeurs à partir des lignes aux nombres de lignes CRN = CEIL(RN) et FRN = FLOOR(RN).

Le résultat final sera: PERCENTILE_CONT(X) = si(CRN = FRN = RN), alors (valeur de l'expression pour RN) autre (CRN - RN) * (valeur de l'expression pour FRN) + (RN - FRN) * (valeur de l'expression pour CRN).

Dans notre exemple, nous avons: PERCENTILE_CONT(0.5). Ici *n* vaut 8. Le nombre de ligne RN = (1 + (0.5*(n-1)))=(1 + (0.5*(8-1))) = 4.5. PERCENTILE_CONT(0.5) = (5 - 4.5) * (valeur de ligne 4) + (4.5 - 4) * (valeur de ligne 5). Les résultats sont:

```
SELECT PERCENTILE_DISC(0.5) WITHIN GROUP
 (ORDER BY cli_credit_limite) AS perc_disc,
 PERCENTILE_CONT(0.5) WITHIN GROUP
 (ORDER BY cli_credit_limite) AS perc_cont
FROM clients
WHERE cli_ville IN ('Neuchâtel', 'Peseux', 'Yverdon') ;

PERC_DISC  PERC_CONT
---------- ----------
      2500       2650
```

Les fonctions d'agrégat de percentile inverse peuvent apparaître dans la clause HAVING comme pour les autres fonctions d'agrégat.

Jeu de données utilisé pour les fonctions PERCENTILE_CONT et PERCENTILE_DISC

```
NUMERO LOG_ZONE         LOG_ADRESSE        LOG_PRIX
------- --------------- ----------------- ---------
     51 Centre           22 Moulin            200000
     52 Centre           2 Château            180000
     53 Centre           8 Seyon              190000
```

Pour trouver la valeur moyenne et la médiane pour chaque zone, on peut utiliser cette requête:

```
SELECT logements.log_zone, TO_CHAR(AVG(logements.log_prix),
'9,999,999,999') Moyenne,
  PERCENTILE_DISC (0.5) WITHIN GROUP
```

Dans l'exemple ci-dessus, pour calculer la médiane du prix des logements, la donnée est ordonnée sur le prix du logement au travers de chaque zone et la médiane du prix est calculée en utilisant les méthodes "discrète" et "d'interpolation. Les résultats montrent comment PERCENTILE_DISC retourne les valeurs actuelles de la table, tandis que PERCENTILE_CONT calcule les nouvelles valeurs interpolées.

Un autre exemple plus parlant de fonction de rapport (*reporting*) avec un percentile inverse. Pour trouver la valeur médiane pour chaque zone et afficher les résultats comme une fonction de rapport, on peut utiliser la requête suivante:

```
SELECT logements.log_zone, TO_CHAR(logements.log_prix,
 '9,999,999,999') Prix,
  PERCENTILE_DISC (0.5) WITHIN GROUP
    (ORDER BY logements.log_prix DESC)
       OVER (PARTITION BY log_zone) AS PERCENTIL_DISC,
  PERCENTILE_CONT (0.5) WITHIN GROUP
    (ORDER BY logements.log_prix DESC)
       OVER (PARTITION BY log_zone) AS PERCENTIL_CONT
  FROM logements;

LOG_ZONE   PRIX          PERCENTIL_DISC PERCENTIL_CONT
---------- ------------ -------------- --------------
Centre           180,000         200000         200000
Centre           189,000         200000         200000
Centre           190,000         200000         200000
Centre           200,000         200000         200000
Centre           245,000         200000         200000
Centre           310,000         200000         200000
Centre           315,000         200000         200000
Urbaine          190,000         300000         290000
Urbaine          195,000         300000         290000
Urbaine          280,000         300000         290000
Urbaine          300,000         300000         290000
Urbaine          340,000         300000         290000
Urbaine          360,000         300000         290000
```

24 Rang Hypothétique et fonctions de répartition (*Hypothetical Rank and Distribution Functions*)

Ces fonctions fournissent des informations utiles pour le genre de question tel que "Qu'est-ce qui se passe si…". Par exemple "si on engage un salarié à 50'000 CHF/année, quel serait le rang (niveau) de son salaire par rapport aux autres employés?".

Syntaxe

Voici un exemple de requête qui trouve les rangs hypothétiques et de distributions pour une maison d'une valeur de 300'000 CHF.

```
SELECT logements.log_zone,
  RANK (300000) WITHIN GROUP
    (ORDER BY logements.log_prix DESC) AS RANK,
  DENSE_RANK (300000) WITHIN GROUP
    (ORDER BY logements.log_prix DESC) AS DENSE_RANK,
  PERCENT_RANK (300000) WITHIN GROUP
    (ORDER BY logements.log_prix DESC) AS PERCENT_RANK,
  CUME_DIST (3000000) WITHIN GROUP
    (ORDER BY logements.log_prix DESC) AS CUM_DIST
  FROM logements GROUP BY logements.log_zone ;

LOG_ZONE          RANK DENSE_RANK PERCENT_RANK   CUM_DIST
---------- ---------- ---------- ------------ ----------
Centre             3          3   ,285714286        ,125
Urbaine            3          3   ,333333333 ,142857143
```

A l'opposé des fonctions de percentile inversé, les fonctions de rang hypothétique ne peuvent pas être utilisées comme fonctions de rapport (*report*).

25 Fonction WIDTH_BUCKET

La fonction WIDTH_BUCKET permet de construire des histogrammes de largeur égales, dans lequel le graphique est divisé en intervalles de même taille. (Comparé avec la fonction NTILE, qui elle, crée des histogrammes de hauteurs égales.) Par exemple, un *bucket* peut être défini entre un intervalle de 10.00 et 19.999… pour indiquer que 10 est inclus dans l'intervalle et 20 est exclu. On peut l'écrire de cette façon: [10, 20).

Pour une expression donnée, la fonction WIDTH_BUCKET retourne un nombre (*bucket number*) dont le résultat de cette expression sera assigné après qu'il ait été évalué.

Syntaxe

Le premier paramètre est l'expression pour laquelle l'histogramme est créé. Le second et le troisième paramètre de l'expression indiquent les points de fin de la rangée pour le premier paramètre. Le quatrième paramètre indique le nombre de *buckets*.

```
WIDTH_BUCKET   (expression,   minval   expression,   maxval
expression, num buckets)
```

L'exemple qui suit, crée un histogramme de dix *buckets* sur la limite de crédit (colonne cli_credit_limite) pour les clients correspondant au critère de la clause WHERE, et retourne le *bucket number* ("BanquePlus") pour chaque client. Les clients dont la limite de crédit dépasse la valeur maximale ont une valeur de *bucket* de 11.

Exemple

```
SELECT cli_nom, cli_credit_limite,
  WIDTH_BUCKET(cli_credit_limite, 1500, 5000, 10)
    AS "BanquePlus"
  FROM clients WHERE CLI_VILLE IN
    ('Neuchâtel','Peseux','Bienne','Yverdon','Fenin')
  ORDER BY "BanquePlus" ;

CLI_NOM              CLI_CREDIT_LIMITE BanquePlus
-------------------- ----------------- ----------
Steiger                           1500          1
Termine                           2000          2
Dunod                             2000          2
Dubois                            2000          2
Tobler                            1999          2
Auvernac                          1900          2
Koller                            2500          3
Steulet                           2500          3
Dupond                            2500          3
Lini                              2500          3
Mathez                            2800          4
Leuenberg                         2800          4
Cramatte                          3500          6
Mavet                             3800          7
Durant                            4000          8
Loiret                            4800         10
Blanc                             5000         11
Joliet                            5100         11
Ayek                              5200         11
Chêne                             5000         11
```

26 Fonctions d'agrégats défini par l'utilisateur (*User-Defined Aggregate Functions*)

Oracle offre un mécanisme pour créer ses propres fonctions, appelée *User-Defined Aggregate Functions*. Ces fonctions sont écrites en langage de programmation tels que PL/SQL, Java, et C. Ces dernières peuvent être utilisées comme fonctions d'analyse ou d'agrégats dans les vues matérialisées.

Les avantages de ces fonctions sont:

- Fonctions complexes pouvant être programmées dans un langage entièrement procédurale.
- Meilleure extension/adaptation (*scalability*) par rapport aux autres techniques pour le traitement en parallèle des fonctions *User-Defined*.
- Des données de type objet peuvent être traitées.

Avant de développer ses propres fonctions, il est important de s'assurer que cela ne peut pas être fait avec des expressions SQL régulières, il est aussi possible d'élaborer des calcules complexes en utilisant l'expression CASE (voir ci-dessous).

27 Expressions CASE

Les expressions CASE sont similaires aux expressions DECODE, mais elles offrent plus de flexibilité et de puissance. Elles sont aussi plus simples à lire que les expressions DECODE. Elles sont souvent utilisées pour créer des catégories (par exemple, 0-15, 16-29, 30-39, …).

Syntaxe

```
expr WHEN comparison_expr THEN return_expr [, WHEN
comparison_expr THEN return_expr]…
```

Syntaxe pour des recherches

```
WHEN condition THEN return_expr [, WHEN condition THEN
return_expr]…
```

Exemple d'expression CASE

L'expression suivante trouve la limite moyenne de crédit dans la table clients, en utilisant 2000 CHF comme la limite inférieure possible:

```
SELECT AVG(CASE WHEN c.cli_credit_limite > 2000 THEN
           c.cli_credit_limite
   ELSE 2000 END) "Crédit moyen" from clients c;

Crédit moyen
------------
        3200
```

Cette expression est beaucoup plus rapide et moins gourmande en ressource que si elle est avait été écrite avec du code PL/SQL par exemple, en effet, chaque ligne n'a pas besoin d'être évaluée par une fonction spécifique qui contrôle si la limite est bien plus grande que 2000…

Autre exemple avec CASE

```
SELECT cli_nom,
   CASE WHEN cli_credit_limite BETWEEN 0 AND 1999 THEN 1 ELSE
     0 END AS "Basse",
   CASE WHEN cli_credit_limite BETWEEN 2000 AND 3999 THEN 1
     ELSE 0 END AS "Moyenne",
   CASE WHEN cli_credit_limite BETWEEN 4000 AND 8000 THEN 1
     ELSE 0 END AS "Haute"
   FROM clients;
```

Résultat de l'expression CASE

CLI_NOM	Basse	Moyenne	Haute
Termine	0	1	0
Dunod	0	1	0
Dubois	0	1	0
Koller	0	1	0
Steulet	0	1	0
Mathez	0	1	0
Leuenberg	0	1	0
Cramatte	0	1	0
Blanc	0	0	1
Chêne	0	0	1
Tobler	1	0	0
Dupond	0	1	0
Durant	0	0	1
Lini	0	1	0
Steiger	1	0	0
Mavet	0	1	0
Loiret	0	0	1
Joliet	0	0	1
Auvernac	1	0	0
Ayek	0	0	1

28 Conclusion

Grâce à l'ajout d'expressions analytiques au langage SQL, l'intégration du *Business intelligence*[12] dans les datawarehouses s'en trouve simplifié. Néanmoins, la création de rapport d'analyse fait uniquement avec le langage SQL requière notamment de très bonnes connaissances "métiers" pour pouvoir générer un bon code SQL. Ces expressions sont très puissantes, bien que pas toujours facile d'écriture. En fait, se seront surtout les applications qui profiteront de la puissance de ces fonctions SQL.

Références

- Documentation officielle d'Oracle - *Data Warehousing Guide*, chapitre *18 SQL for Aggregation in Data Warehouses* et *19 SQL for Analysis in Data Warehouses*.
- Sur le site technet.oracle.com: *Analytic SQL Features in Oracle9i*.

[12] ensemble des technologies permettant en fin de chaîne d'apporter une aide à la décision.