Un bref rappel des commandes SQL et SQL étendu sous Oracle

Une transaction est un ensemble de modifications de la base qui forme un tout indivisible. Il faut effectuer ces modifications entièrement ou pas du tout, sous peine de laisser la base dans un état incohérent.
Les Systèmes de Gestion de Bases de Données permettent aux utilisateurs de gérer leurs transactions. Ils peuvent à tout moment :

- Valider la transaction en cours par la commande **COMMIT**. Les modifications deviennent définitives et visibles à tous les utilisateurs.
- Annuler la transaction en cours par la commande **ROLLBACK**. Toutes les modifications depuis le début de la transaction sont alors défaites.

En cours de transaction, seul l'utilisateur ayant effectué les modifications les voit.

1. Création d'une table

La table est la structure de base contenant les données des utilisateurs. La syntaxe de la création d'une table est la suivante :

```
CREATE TABLE nom_table
(nom_col1 TYPE1,
nom_col2 TYPE2,
...)
```

Les types de données peuvent être :

- **NUMBER[(longueur,[précision]) :** Longueur précise le nombre maximum de chiffres significatifs stockés (par défaut 38). Précision donne le nombre maximum de chiffres après la virgule (par défaut 38), sa valeur peut être comprise entre -84 et 127. Une valeur négative signifie que le nombre est arrondi à gauche de la virgule.
- **CHAR(longueur) :** Ce type de données permet de stocker des chaînes de caractères de longueur fixe. Longueur doit être inférieur à 255, sa valeur par défaut est 1.
- **VARCHAR(longueur) :** Ce type de données permet de stocker des chaînes de caractères de longueur variable. Longueur doit être inférieur à 2000, il n'y a pas de valeur par défaut.
- **DATE :** Ce type de données permet de stocker des données constituées d'une date et d'une heure.

La commande **DROP TABLE** permet de supprimer une table, sa syntaxe est la suivante :

```
DROP TABLE nom_table ;
```

On a la possibilité de changer le nom d'une table par la commande RENAME, la syntaxe est la suivante :

```
RENAME ancien_nom TO nouveau_nom ;
```

2. Interrogation d'une base

La commande **SELECT** permet de récupérer des données contenues dans une ou plusieurs tables.
La commande **SELECT** la plus simple a la syntaxe suivante :

```
SELECT *
FROM nom_table ;
```

Par défaut toutes les lignes sont sélectionnées. On peut limiter la sélection à certaines colonnes, en indiquant une liste de noms de colonnes à la place de l'astérisque.

```
SELECT nom_col1, nom_col2, ...
FROM nom_table ;
```

La clause **DISTINCT** ajoutée derrière la commande **SELECT** permet d'éliminer les duplications

```
SELECT DISTINCT fonction
FROM emp;
```

La clause **WHERE** permet de spécifier quelles sont les lignes à sélectionner. Elle est suivie d'un
prédicat qui sera évalué pour chaque ligne de la table. Les lignes pour lesquelles le prédicat
est vrai seront sélectionnées.
La syntaxe est la suivante :

```
SELECT *
FROM nom_table
WHERE predicat ;
```

SELECT qui permet aussi d'effectuer des calculs sur l'ensemble des valeurs d'une colonne. Ces
calculs sur l'ensemble des valeurs d'une colonne se font au moyen de l'une des fonctions
suivantes :

- **AVG([DISTINCT | ALL] expression) :** Renvoie la moyenne des valeurs d' expression.
- **COUNT(* | [DISTINCT | ALL] expression) :** Renvoie le nombre de lignes du résultat de la requête. Si expression est présent, on ne compte que les lignes pour lesquelles cette expression n'est pas NULL.
- **MAX([DISTINCT | ALL] expression :** Renvoie la plus petite des valeurs d'expression.
- **MIN([DISTINCT | ALL] expression) :** Renvoie la plus grande des valeurs d'expression.
- **STDDEV([DISTINCT | ALL] expression) :** Renvoie l'écart-type des valeurs d'expression.
- **SUM([DISTINCT | ALL] expression) :** Renvoie la somme des valeurs.
- **VARIANCE([DISTINCT | ALL] expression) :** Renvoie la variance des valeurs d'expression.
- **DISTINCT :** Indique à la fonction de groupe de ne prendre en compte que des valeurs distinctes.
- **ALL :** Indique à la fonction de groupe de prendre en compte toutes les valeurs, c'est la valeur par défaut.

Le résultat d'une requête est constitué d'un ensemble de colonnes. Par défaut, l'en-tête de chaque colonne est l'expression fournie dans la liste de sélection.
Lorsque les résultats de la requête s'affichent, l'en-tête par défaut de chaque colonne est le nom qui lui a été attribué lors de sa création. Vous pouvez spécifier un autre en-tête de colonne, ou alias, de l'une des trois façons suivantes :

```
SELECT nom_colonne AS alias
SELECT nom_colonne alias
SELECT alias = nom_colonne
```

Quelques exemples:

Exemple : Quels sont les employés dont la commission est supérieure au salaire?

```
SELECT nom, salaire, comm
FROM emp
WHERE comm > salaire;
```

Exemple : Quels sont les employés gagnant entre 20000 et 25000?

```
SELECT nom, salaire
FROM emp
WHERE salaire BETWEEN 20000 AND 25000;
```

Exemple : Quels sont les employés commerciaux ou ingénieurs?

```
SELECT num, nom, fonction, salaire
FROM emp
WHERE fonction IN ('commercial','ingénieur');
```

Exemple : Quels sont les employés dont le nom commence par M?

```
SELECT nom
FROM emp
WHERE nom LIKE 'M%';
```

Exemple : Quels sont les employés du département 30 ayant un salaire supérieur à 25000?

```
SELECT nom
FROM emp
WHERE n_dept = 30
AND salaire > 25000;
```

Exemple : Quels sont les employés directeurs, ou commerciaux et travaillant dans le département 10?

```
SELECT nom, fonction, salaire, n_dept
FROM emp
WHERE fonction = 'directeur'
OR (fonction = 'commercial' AND n_dept = 10);
```

Exemple : Quels sont les employés dont la commission a la valeur NULL?

```
SELECT nom
FROM emp
WHERE comm IS NULL;
```

Exemple : Salaire de chaque employé.

```
SELECT nom, salaire "SALAIRE MENSUEL"
FROM emp;
```

3. Classement du résultat d'une interrogation

Les critères de classement sont spécifiés dans une clause **ORDER BY** dont la syntaxe est la suivante :

```
ORDER BY {nom_col1 | num_col1 [DESC] [, nom_col2 | num_col2 [DESC],...]}
```

Le classement se fait d'abord selon la première colonne spécifiée dans l'**ORDER BY** puis les lignes ayant la même valeur dans la première colonne sont classées selon la deuxième colonne de l'**ORDER BY**, etc... Pour chaque colonne, le classement peut être ascendant (par défaut) ou descendant (**DESC**).

Exemple : Donner tous les employés classés par fonction, et pour chaque fonction classés par salaire décroissant

```
SELECT nom, fonction, salaire
FROM emp
```

```
ORDER BY fonction, salaire DESC;
```
Dans un classement les valeurs **NULL** sont toujours en tête quel que soit l'ordre du classement (ascendant ou descendant).

4. Calcul sur des groupes

Il est possible de subdiviser la table en groupes, chaque groupe étant l'ensemble des lignes ayant une valeur commune. C'est la clause **GROUP BY** qui permet de découper la table en plusieurs groupes :

```
GROUP BY expr_1, expr_2, ...
```

Exemple : Total des salaires pour chaque département

```
SELECT SUM(salaire), n_dept
FROM emp
GROUP BY n_dept;
```

Remarque : Dans la liste des colonnes résultat d'un **SELECT** comportant une fonction de groupe, ne peuvent figurer que des caractéristiques de groupe, c'est-à-dire : soit des fonctions de groupe, soit des expressions figurant dans le **GROUP BY**.

Exemple : Donner la liste des salaires moyens par fonction pour les groupes ayant plus de deux employés.

```
SELECT fonction,COUNT(*),AVG(salaire)
FROM emp
GROUP BY fonction
HAVING COUNT(*) > 2;
```

Remarque : Un **SELECT** de groupe peut contenir à la fois une clause **WHERE** et une clause **HAVING**. La clause **WHERE** sera d'abord appliquée pour sélectionner les lignes, puis les groupes seront constitués à partir des lignes sélectionnées, et les fonctions de groupe seront évaluées.

Exemple : Donner le nombre d'ingénieurs ou de commerciaux des départements ayant au
moins deux employés de ces catégories.

```
SELECT n_dept, COUNT(*)
FROM emp
WHERE fonction in ('ingenieur','commercial')
GROUP BY n_dept
HAVING COUNT(*) >= 2;
```

Une clause **HAVING** peut comporter une sous-interrogation.

Exemple : Quel est le département ayant le plus d'employés?

```
SELECT n_dept,COUNT(*)
FROM emp
GROUP BY n_dept
HAVING COUNT(*) = (SELECT MAX(COUNT(*))
FROM emp
GROUP BY n_dept);
```

5. Expressions et fonctions simples

Une expression est un ensemble de variables (contenu d'une colonne), de constantes et de fonctions combinées au moyen d'opérateurs. Les fonctions prennent une valeur dépendant de leurs arguments qui peuvent être eux-mêmes des expressions.
Les expressions peuvent figurer :

- en tant que colonne résultat d'un **SELECT**,
- dans une clause **WHERE**,
- dans une clause **ORDER BY**.

Il existe trois types d'expressions correspondant chacun à un type de données de SQL : arithmétique, chaîne de caractère, date. A chaque type correspondent des opérateurs et des
fonctions spécifiques.

a. Expressions et fonctions arithmétiques

Une expression arithmétique peut contenir des noms de colonnes, des constantes, des fonctions arithmétiques combinés au moyen des opérateurs arithmétiques.

· **Opérateurs arithmétiques**

Les opérateurs arithmétiques présents dans SQL sont les suivants :

- + addition ou + unaire
- - soustraction ou - unaire
- * multiplication
- / division

Exemple : Donner pour chaque commercial son revenu (salaire + commission).

```
SELECT nom, salaire+comm
FROM emp
WHERE fonction = 'commercial';
```

· **Fonctions arithmétiques**

- **ABS(nb) :** Renvoie la valeur absolue de nb.
- **CEIL(nb) :** Renvoie le plus petit entier supérieur ou égal à nb.
- **COS(n) :** Renvoie le cosinus de n, n étant un angle exprimé en radians.
- **COSH(n) :** Renvoie le cosinus hyperbolique de n.
- **EXP(n):** Renvoie e puissance n.
- **FLOOR(nb) :**Renvoie le plus grand entier inférieur ou égal à nb.
- **LN(n) :** Renvoie le logarithme népérien de n qui doit être un entier strictement positif.
- **LOG(m,n) :** Renvoie le logarithme en base m de n. m doit être un entier strictement supérieur à 1, et n un entier strictement positif.
- **MOD(m,n) :** Renvoie le reste de la division entière de m par n, si n vaut 0 alors renvoie m. Attention, utilisée avec au moins un de ses arguments négatifs, cette fonction donne des résultats qui peuvent être différents d'un modulo classique. Cette fonction ne donne pas toujours un résultat dont le signe du diviseur.
- **POWER(m,n) :** Renvoie m puissance n, m et n peuvent être des nombres quelconques entiers ou réels mais si m est négatif n doit être un entier.
- **ROUND(n[,m]) :** Si m est positif, renvoie n arrondi (et non pas tronqué) à m chiffres après la virgule. Si m est négatif, renvoie n arrondi à m chiffres avant la virgule. m doit être un entier et il vaut 0 par défaut.
- **SIGN(nb) :** Renvoie -1 si nb est négatif, 0 si nb est nul, 1 si nb est positif.
- **SIN(n) :** Renvoie le sinus de n, n étant un angle exprimé en radians.
- **SINH(n) :** Renvoie le sinus hyperbolique de n.
- **SQRT(nb) :** Renvoie la racine carrée de nb qui doit être un entier positif ou nul.

- **TAN(n) :** Renvoie la tangente de n, n étant un angle exprimé en radians.
- **TANH(n) :** Renvoie la tangente hyperbolique de n.
- **TRUNC(n[,m]) :** Si m est positif, renvoie n arrondi tronqué à m chiffres après la virgule. Si m est négatif, renvoie n tronqué à m chiffres avant la virgule. m doit être un entier et il vaut 0 par défaut.

b. Autres fonctions analytiques

- **GREATEST(expr1, expr2,...) :** Renvoie la plus grande des valeurs expr1, expr2,.... Toutes les expressions sont converties au format de expr1 avant comparaison.
- **LEAST(expr1, expr2,...) :** envoie la plus petite des valeurs expr1, expr2,.... Toutes les expressions sont converties au format de expr1 avant comparaison.
- **NVL(expr_1, expr_2) :** Prend la valeur expr_1, sauf si expr_1 est NULL auquel cas NVL prend la valeur expr_2. 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 fonction **NVL** permet de remplacer une valeur **NULL** par une valeur significative.
- **DECODE(crit, val_1, res_1 [, val_2, res_2 ...], def) :** Cette fonction permet de choisir une valeur parmi une liste d'expressions, en fonction de la valeur prise par une expression servant de critère de sélection. Le résultat récupéré est : **res_1** si l'expression **crit** a la valeur **val_1**, **res_2** si l'expression **crit** a la valeur **val_2**, **def** (la valeur par défaut) si l'expression **crit** n'est égale à aucune des expressions **val_1, val_2,...,.** Les expressions résultats **res_1, res_2, ..., def** peuvent être de types différents : caractère et numérique, ou caractère et date (le résultat est du type de la première expression rencontré dans le **DECODE**).

  Exemple : Donner la liste des employés avec pour chacun d'eux sa catégorie (président = 1, directeur = 2, autre = 3).

```
SELECT nom, DECODE(fonction,'president',1,'directeur',2,3)
FROM emp;
```

  Exemple : Donner la liste des employés en les identifiant par leur fonction dans le département 10 et par leur nom dans les autres départements.

```
SELECT DECODE (n_dept,10,fonction,nom)
FROM emp;
```

- **RANK () :** Elle 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. La syntaxe de cette fonction est la suivante :

```
RANK () OVER ([partioning] ordering)
```

  Exemple :

```
SELECT emp_lname, salary, state,
RANK () OVER (ORDER BY salary DESC) "Rank"
FROM employee WHERE state IN ('NY','UT')
```

- **Dense_RANK () :** élimine les trous (vides) dans la séquence de classement.

- **ROW_NUMBER () :** elle attribue un nombre unique (séquentielle, à partir de 1, définit par ORDER BY) pour chaque ligne dans la partition. La syntaxe de cette fonction est la suivante :
  ROW_NUMBER()OVER([partioning] ordering)
- **NTILE (N) :** La fonction **NTILE** ordonne les lignes en **N** packets, **N** étant un entier passé en paramètre. La clause **ORDER BY** est donc obligatoire. **NTILE** renvoie pour chaque ligne le numéro du paquet auquel cette ligne appartient.

6. Les entrepôts de données

Un entrepôt de données est une base de données :

- Consolidant les données de bases de données opérationnelles dont les schémas sont généralement différents,
- Utilisée en consultation et seulement mise à jour périodiquement (par exemple tous les jours),
- Organisée pour permettre le traitement de requêtes analytiques plutôt que transactionnelles (OLAP par rapport à OLTP).

Pour extraire des informations d'un entrepôt de données il y'a deux approches principales

Exprimer les requêtes en SQL, on parle alors de ROLAP (relational on-line analytical processing). C'est à cette approche qu'on s'intéresse dans le cadre de ce TP.

Voir l'entrepôt comme une base de données multidimensionnelle, dont le modèle est un cube a n-dimensions. Il y a une dimension du cube par attribut dimensionnel et les éléments du cube sont les valeurs des attributs dépendants. On parle alors de MOLAP (multidimensional on-line analytical processing).

7. Commandes SQL pour les agrégations dans les ED (Approche ROLAP)

a. Utilisation de ROLLUP

L'opération **ROLLUP** calcule les sous-totaux de lignes agrégées. Elle ajoute des lignes de sous-total dans les jeux de résultats des requêtes comportant des clauses **GROUP BY**. Elle génère un jeu de résultats indiquant les agrégats pour une hiérarchie de valeurs de colonnes sélectionnées. Utilisez l'opération **ROLLUP** si vous souhaitez qu'un jeu de résultats indique les totaux et les sous-totaux.

Cette syntaxe SQL...	définit les jeux suivants...

`GROUP BY ROLLUP (A, B, C);`	`(A, B, C)`
	`(A, B)`
	`(A)`
	`( )`

Cette requête ROLLUP...	équivaut à cette requête sans ROLLUP...

`SELECT A, B, C,SUM( D )`
`FROM T1`
`GROUP BY ROLLUP (A, B, C);`
	`SELECT * FROM (`
	`( SELECT A, B, C, SUM( D )`
	`GROUP BY A, B, C )`
	`UNION ALL`
	`( SELECT A, B, NULL, SUM( D )`
	`GROUP BY A, B )`
	`UNION ALL`
	`( SELECT A, NULL, NULL, SUM( D )`
	`GROUP BY A )`
	`UNION ALL`
	`( SELECT NULL, NULL, NULL, SUM( D ) )`
	`)`

b. Utilisation de GROUPING SETS

`GROUPING SETS` permet de calculer des groupes sur différents ensembles de colonnes de regroupement dans la même requête. Tandis que `CUBE` et `ROLLUP` ajoutent un ensemble prédéfini de sous-totaux au jeu de résultats, `GROUPING SETS` permet de spécifier les sous-totaux à ajouter. Il est plus complexe à utiliser que `ROLLUP` ou `CUBE` mais offre un contrôle plus précis sur le jeu de résultats. Cette opération est utile si vous souhaitez ne renvoyer qu'une partie du jeu de résultats multidimensionnel d'un cube. `GROUPING SETS` permet de sélectionner les variables à comparer plutôt que de renvoyer une comparaison de toutes les variables comme `CUBE` ou un sous-ensemble hiérarchique comme `ROLLUP`.

Cette syntaxe SQL...	définit les jeux suivants...
`GROUP BY GROUPING SETS ((A, B), (A, C), (C));`	(A, B) (A, C) (C)

Cette requête GROUPING SETS...	équivaut à cette requête sans GROUPING SETS...
`SELECT A, B, C,SUM( D ) FROM T1 GROUP BY GROUPING SETS (A, B, C);`	`SELECT * FROM ( ( SELECT A, NULL, NULL, SUM( D ) FROM t GROUP BY A ) UNION ALL ( SELECT NULL, B, NULL, SUM( D ) FROM t GROUP BY B ) UNION ALL ( SELECT NULL, NULL, C, SUM( D ) FROM t GROUP BY C ) )`

Utilisation de GROUPING SETS concaténés

Cette syntaxe SQL...	définit les jeux suivants...
`GROUP BY GROUPING SETS (A, B)(Y, Z);`	(A, Y) (A, Z) (B, Y) (B, Z)

c. Utilisation de CUBE

L'opérateur `CUBE` renvoie un jeu de résultats avec des informations supplémentaires sur les dimensions. Vous pouvez analyser plus précisément les colonnes de données, qui sont appelées dimensions. `CUBE` fournit, sous la forme d'un tableau croisé, un rapport sur toutes les combinaisons de dimensions possibles, puis génère un jeu de résultats présentant les agrégats de toutes les combinaisons de valeurs dans les colonnes sélectionnées.

Cette syntaxe SQL...	définit les jeux suivants...
`GROUP BY CUBE (A, B, C);`	(A, B, C) (A, B) (A, C) (B, C) (A) (B) (C) ( )

Cette requête CUBE...	équivaut à cette requête sans CUBE...

`SELECT A, B, C,SUM( D )`
`FROM T1`
`GROUP BY CUBE (A, B, C);`
	`SELECT A, B, C,`
	`SUM( D )`
	`FROM t`
	`GROUP BY GROUPING SETS(`
	`( A, B, C ),`
	`( A, B ), ( A ), ( B, C ),`
	`( B ), ( A, C ), ( C ),`
	`() )`

GROUPING Function: Indique si une colonne dans un jeu de résultats de l'opération **ROLLUP,CUBE ou GROUPIN SETS** a la valeur **NULL** parce qu'elle fait partie d'une ligne de sous-total, ou en raison des données sous-jacentes.
Elle peut figurer comme suit :

```
SELECT ... [GROUPING(dimension_column)...] ...
GROUP BY ... {CUBE | ROLLUP| GROUPING SETS} (dimension_column)
```

- **GROUPING_ID Function :** La fonction **GROUPING_ID** offre la possibilité d'économiser de l'espace lors de l'identification des lignes super-agrégées. A l'instar de la fonction **GROUPING**, cette fonction identifie ces lignes et renvoie le niveau **GROUP BY** comme bit-vector. **GROUPING_ID** prend toutes les valeurs 1 et 0 qui ont été créées lors de l'utilisation de la fonction **GROUPING** et génère à partir de celles-ci un bit-vector par agrégat. Le bit-vector est traité comme un nombre binaire et la valeur est retournée par la fonction **GROUPING_ID**. Lorsque, par exemple, un agrégat est exécuté avec l'expression **CUBE(a,b)**, les valeurs possibles sont les suivantes

Aggregation Level	Bit Vector	GROUPING_ID
a, b	0 0	0
a	0 1	1
b	1 0	2
Grand Total	1 1	3