Chapitre 10

Bases de données

La portion de code fournie par le Ministère, que nous allons étudier dans le chapitre suivant, est écrit en PL/SQL, un langage de programmation propriétaire d'Oracle, concepteur du système de gestion de bases données le plus utilisé par les entreprises. Nous allons donc faire un survol rapide des bases de données, du langage SQL d'interrogation des bases de données et, enfin, de PL/SQL, qui permet d'associer les requêtes SQL aux structures de contrôle des langages de programmation, ce qui nous introduira au code du ministère.

10.1 Les bases de données relationnelles

Bases de données

On appelle **base de données** toute façon permettant de stocker, puis de récupérer, des informations en rapport avec un thème ou une activité. Ces informations sont très structurées [1], et la base est localisée dans un même lieu et sur un même support. Ce dernier est généralement informatisé de nos jours.

Le *système de gestion de base de données* (SGBD en abrégé ; DBMS pour *DataBase Management System* en anglais) est une suite de programmes qui manipule la base de données et permet d'accéder aux données qui y sont stockées

Le terme anglais (*database*) pour base de données est apparu en 1964, pour désigner une collection d'informations partagées par différents utilisateurs d'un système d'informations militaire.

Les premières bases de données elles-mêmes sont apparues au début des années 1960.

1. On parle d'*entrepôt de données* lorsque les informations ne sont pas structurées.

En juin 1970, Edgar Frank CODD publie l'article *A Relational Model of Data for Large Shared Data Banks* (« Un référentiel de données relationnel pour de grandes banques de donnés partagées ») dans la revue *Communications of the ACM* (*Association for Computing Machinery*). Ce modèle relationnel, fondé sur la logique des prédicats du premier ordre, est rapidement reconnu comme un modèle théorique intéressant pour l'interrogation des bases de données et CODD reçoit le prix Turing (l'analogue du prix Nobel pour l'informatique théorique) en 1981.

Principe des bases de données relationnelles

Une **base de données relationnelle** est une base de données où l'information est organisée dans des tableaux à deux dimensions, appelés des *relations* ou **tables**. Une base de données consiste en une ou plusieurs tables. Les lignes de ces tables sont appelées des *uplets* ou **enregistrements**. Les colonnes sont appelées des **attributs**. Les composant des lignes sont les **champs**, déterminés par une ligne et une colonne.

Un attribut est une **clé primaire** lorsque la valeur de cet attribut est différent pour chaque ligne d'une table. Une clé primaire permet donc de repérer une seule ligne de la table.

10.2 SQL

Définition

SQL (sigle de *Structured Query Language*, en français « langage de requête structurée ») est un langage servant à exploiter des bases de données relationnelles. Il comprend une partie langage de *manipulation* des données, permettant de rechercher, d'ajouter, de modifier ou de supprimer des données dans les bases de données relationnelles.

Il comprend également une partie langage de *définition* des données, permettant de créer et de modifier l'organisation des données dans la base de données, une partie langage de *contrôle de transaction*, permettant de commencer et de terminer des transactions, et une partie langage de *contrôle des données*, permettant d'autoriser ou d'interdire l'accès à certaines données à certaines personnes. Ces trois dernières fonctionnalités ne nous intéresseront pas ici.

Historique

Le modèle relationnel de CODD inspire le langage *Structured English QUEry Language* (SEQUEL) (« langage d'interrogation structurée » en anglais), développé chez IBM en 1970 par Donald CHAMBERLIN et Raymond BOYCE, conçu pour manipuler et éditer des données stockées dans une base de données relationnelles à l'aide du système de gestion de base de données d'IBM, appelé *System R*. Le nom SEQUEL, déposé commercialement par l'avionneur *Hawker Siddeley* pour un système d'acquisition de données, est abandonné et contracté en SQL en 1975.

En 1979, *Relational Software, Inc.* (devenu plus tard *Oracle Corporation*) présente la première version commercialement disponible de SQL, rapidement imité par d'autres fournisseurs.

SQL est adopté comme recommandation par l'*Institut de normalisation américain* (ANSI) en 1986, puis comme norme internationale par l'ISO (*Organisation de Normalisation Internationale*) en 1987 sous le nom de *ISO/CEI 9075 - Technologies de l'information - Langages de base de données - SQL*.

Donnons des exemples avec des tables ressemblant au code que nous étudierons dans le chapitre suivant pour les trois genres de manipulations : création, transaction et interrogation d'une base de données relationnelle.

10.2.1 Création d'une base de données

Création de la base

La création de la base se fait en général grâce à une commande du SGBD. Certaines implémentations de SQL, mais pas toutes, ont, pour créer la base de données de nom « ma_base », une requête analogue à la requête suivante :

```
CREATE DATABASE ma_base
```

Remarque.- La casse (minuscule ou majuscule) d'un caractère en SQL n'a pas d'importance. En général, on choisit d'utiliser les majuscules pour les mots clés de SQL et les minuscules pour les autres mots.

Création d'une table

La création d'une table sert à définir les colonnes et le type de données qui seront contenus dans chacune des colonnes (entier, chaîne de caractères, date, valeur binaire...).

Syntaxe.- La syntaxe générale pour créer une table est la suivante :

```
CREATE TABLE nom_de_la_table
(
colonne1 type_donnees,
colonne2 type_donnees,
colonne3 type_donnees,
colonne4 type_donnees )
```

Types.- Les types principaux sont : À modifier par Pierre

- INT pour les entiers ;
- VARCHAR(*taille*) pour les chaînes de caractères, où *taille* est une constante entière positive, désignant la longueur maximum de la chaîne de caractère.

Options.- Pour chaque colonne, il est possible de définir des options telles que :

- NOT NULL : empêche d'enregistrer une valeur nulle pour une colonne ;
- DEFAULT : attribue une valeur par défaut si aucune donnée n'est indiquée pour cette colonne lors de l'ajout d'une ligne dans la table ;
- PRIMARY KEY : indique que cette colonne est considérée comme clé primaire pour un index.

Exemple.- Par exemple :

```
CREATE TABLE candidats
(
identifiant INT PRIMARY KEY NOT NULL,
nom VARCHAR(100),
prenom VARCHAR(100),
ville VARCHAR(255),
age INT )
```

permet de créer une table rudimentaire de « candidats ».

10.2.2 Transactions : insertion et mise à jour

On appelle *transaction* tout travail unitaire effectué sur une base de données qui conduit à un ou plusieurs changements de la base de données.

Insertion de lignes dans une table

L'insertion de données dans une table s'effectue à l'aide de la commande INSERT INTO.

Insertion d'une ligne complète.- Pour insérer une ligne, en indiquant les valeurs pour chaque colonne existante dans l'ordre de celles-ci, on utilise la requête :

```
INSERT INTO table
VALUES (valeur1, valeur2 ...)
```

Lorsque le champ à remplir est de type VARCHAR ou TEXT, il faut placer le texte entre guillemets verticaux. En revanche, lorsque la colonne est un nombre de type tel que INT ou BIGINT, il n'y a pas besoin d'utiliser de guillemets, il suffit juste d'indiquer le nombre.

Insertion d'une ligne incomplète.- Il est possible de ne pas renseigner toutes les colonnes. Dans ce cas, il faut spécifier les noms colonnes que l'on veut compléter avant le mot clé VALUES :

```
INSERT INTO table
(nom_colonne_1, nom_colonne_2 ...)
VALUES (valeur1, valeur2 ...)
```

Dans ce deuxième cas, l'ordre des colonnes n'est pas important.

Valeur nulle.- Lorsqu'on insère une ligne incomplète, quelle est la valeur prise pour les colonnes non spécifiées ?

Nous avons vu que cela peut être une valeur par défaut spécifiée lors de la création de la table grâce au mot clé DEFAULT.

Lorsque ce n'est pas le cas, il vaut mieux ne pas utiliser l'une des valeurs compatible avec le type de la colonne. La plupart des SGBD ont une valeur spéciale, dite **valeur nulle**, qui ne fait partie d'aucun des types utilisés. Cette valeur nulle est notée NULL en SQL.

Insertion de plusieurs lignes à la fois.- On peut insérer plus sieurs lignes à la fois, en séparant les n-uplets de valeurs à insérer par des virgules.

Par exemple, la requête :

```
INSERT INTO candidats
VALUES
   (1, 'Pierre', 'Dupond', 'Paris', 20),
   (2, 'Sabrina', 'Durand', 'Nantes', 19),
   (3, 'Julien', 'Martin', 'Lyon', 21),
   (4, 'David', 'Bernard', 'Marseille', 18),
   (5, 'Marine', 'Leroy', 'Paris', 20)
```

conduit à la table suivante :

identifiant	prenom	nom	ville	age
1	Pierre	Dupond	Paris	20
2	Sabrina	Durand	Nantes	19
3	Julien	Martin	Lyon	21
4	David	Bernard	Marseille	18
5	Marine	Leroy	Paris	20

Mise à jour

On utilise la commande UPDATE pour effectuer une mise à jour, c'est-à-dire remplacer certaines valeurs d'une ligne par d'autres :

```
UPDATE table
SET nom_colonne_1 = nouvelleValeur
WHERE condition
```

Par exemple, la requête :

```
UPDATE candidats
SET prenom = 'Marie'
WHERE identifiant = 5
```

conduit à la table suivante :

identifiant	prenom	nom	ville	age
1	Pierre	Dupond	Paris	20
2	Sabrina	Durand	Nantes	19
3	Julien	Martin	Lyon	21
4	David	Bernard	Marseille	18
5	Marie	Leroy	Paris	20

Contrôle des transactions

Les commandes INSERT, UPDATE et DELETE (nous n'avons pas vu cette dernière) doivent être suivies d'une commande de contrôle de transaction, dont :

— COMMIT pour sauvegarder les changements effectués ;
— ROLLBACK pour annuler les changements effectués.

Une telle commande de contrôle ne suit pas nécessairement *immédiatement* une transaction : elle s'effectue en général après avoir effectué que quelques vérifications.

10.2.3 Projection

L'utilisation la plus courante de SQL consiste à lire des données issues de la base de données. Cela s'effectue grâce à la commande SELECT, qui retourne des enregistrements dans un tableau de résultat. En général on ne retourne pas l'intégralité des données mais une partie seulement, représentant l'information désirée.

Projection d'une colonne

Obtenir une colonne, et non pas l'ensemble des colonnes, s'effectue grâce à la requête suivante :

```
SELECT nom_du_champ
FROM nom_table
```

Par exemple, si l'on veut avoir la liste des villes d'où proviennent les candidats, on obtient cette information grâce à la requête suivante :

```
SELECT ville
FROM candidats
```

dont le résultat est :

```
 ville
-------
 Paris
 Nantes
 Lyon
 Marseille
 Paris
```

Remarque.- Sous quelle forme le résultat d'une requête est-il obtenu ?

Dans ce qui est appelé le *mode interactif* (d'une implémentation) de SQL, on obtiendra effectivement quelque chose qui ressemble à ce que nous avons indiqué, avec plus ou moins d'ornements.

Suppression des doublons

Dans la *vue* obtenue par projection dans l'exemple précédent, on obtient deux fois la ville 'Paris'. Cela n'est pas particulièrement gênant dans ce cas, mais cela peut le devenir. On peut éviter les *doublons* en ajoutant un modificateur à la requête :

SELECT DISTINCT nom_du_champ
FROM nom_table

Par exemple, si l'on veut avoir la liste des villes d'où proviennent les candidats, sans redondance, il suffit d'effectuer la requête suivante :

```
SELECT DISTINCT ville
FROM candidats
```

dont le résultat est :

```
 ville
-------
 Paris
 Nantes
 Lyon
 Marseille
```

Tri de la vue

Dans la mesure où on s'intéresse au résultat mais pas à la base de données, ni même à la table, dans son intégralité, il n'y a aucune raison de laisser apparaître les noms des villes dans l'ordre de la table initiale. Il vaut mieux trier le résultat obtenu.

Pour cela on fait suivre la requête précédente de :

ORDER BY nom_du_champ

Par défaut les résultats seront classés par ordre ascendant. Toutefois il est possible d'inverser l"ordre en utilisant le suffixe DESC après le nom de la colonne. On peut aussi utiliser le suffixe ASC mais cela n'est pas utile puisque l'ordre ascendant est celui par défaut.

Par exemple, si l'on veut avoir la liste des villes d'où proviennent les candidats, sans redondance et classées par odre alphabétique, on peut effectuer la requête suivante :

```
SELECT DISTINCT ville
FROM candidats
ORDER BY ville
```

dont le résultat est :

```
 ville
--------
 Lyon
 Marseille
 Nantes
 Paris
```

Remarque.- Les puristes auront remarqué que pour qu'il puisse y avoir tri, les types de données doivent être des ensembles munis d'un ordre total (naturel). Ceci est bien le cas.

Cela ne pourrait pas se faire si on utilisait, par exemple, un type COULEUR (bien qu'on pourrait utiliser la longueur d'onde correspondante).

Sélection

On peut ne vouloir obtenir que les candidats habitant Paris, par exemple. Pour cela on filtre grâce à une requête utilisant le mot clé WHERE :

```
SELECT nom_colonne
FROM nom_table
WHERE condition
```

Par exemple, si l'on veut avoir la liste des noms des candidats habitant Paris, on peut utiliser la requête suivante :

```
SELECT nom
FROM candidats
WHERE ville = 'Paris'
```

dont le résultat est :

```
 nom
------
 Dupond
 Leroy
```

La requête suivante aurait certainement été plus appropriée :

```
SELECT DISTINCT nom
FROM candidats
WHERE ville = 'Paris'
ORDER BY nom
```

mais elle aurait donné le même résultat pour notre exemple très simple.

Opérateurs de comparaison primitifs

Nous avons utilisé l'opérateur d'égalité pour effectuer le filtrage ci-dessus. De façon plus générale, il existe d'autres opérateurs de comparaisons, dont la liste ci-jointe présente les plus couramment utilisés :

Opérateur	Description
=	Égal
<>	Pas égal
!=	Pas égal
>	Strictement supérieur à
<	Strictement inférieur à
>=	Supérieur ou égal à
<=	Inférieur ou égal à
IN	Liste de plusieurs valeurs possibles
IS NULL	La valeur est nulle
IS NOT NULL	La valeur n'est pas nulle

L'utilisation des sept premiers cas devrait être claire. Détaillons les deux derniers cas.

Sélection parmi une suite de valeurs

Pour chercher toutes les lignes dont le champ « nom_colonne » est égal à `valeur1` ou `valeur2` ou `valeur3`, il est possible d'utiliser la syntaxe suivante :

```
SELECT nom_colonne
FROM table
WHERE nom_colonne IN (valeur1, valeur2, valeur3)
```

Il n'y a pas de limite dans le nombre de valeurs que l'on peut proposer.

Sélection des valeurs nulles

L'opérateur IS permet de filtrer les lignes qui contiennent la valeur NULL. Cet opérateur est indispensable car la valeur NULL est une valeur inconnue et ne peut pas par conséquent être filtrée par les opérateurs de comparaison (égal, inférieur, supérieur ou différent).

Pour filtrer les lignes où les champs d'une colonne sont à NULL, on utilise la syntaxe suivante :

```
SELECT nom_colonne
FROM table
WHERE nom_colonne IS NULL
```

À l'inverse, pour filtrer les lignes dont le champ spécifié n'est pas NULL, on utilise la syntaxe suivante :

```
SELECT nom_colonne
FROM table
WHERE nom_colonne IS NOT NULL
```

Combinaison de plusieurs opérateurs de comparaison

On peut combiner les conditions primitives (celles utilisant les opérateurs de comparaison primitifs) grâce aux connecteurs logiques NOT, AND et OR, en utilisant des couples de parenthèses le cas échéant.

Par exemple, si l'on veut avoir la liste des noms des candidats habitant Paris ou dont le prénom est David, on peut effectuer la requête suivante :

```
SELECT nom
FROM candidats
WHERE (ville = 'Paris') OR (prenom = 'David')
```

dont le résultat est :

nom
Dupond
Bernard
Leroy

Projection de plusieurs colonnes

La vue peut comprendre plusieurs colonnes. Pour cela on utilise une requête :

```
SELECT nom_colonne1, nom_colonne2
FROM nom_table
```

dans laquelle les noms des colonnes sont indiquées dans l'ordre désiré d'affichage, séparés par des virgules. Le symbole spécial « * » remplace la liste de toutes les colonnes, dans l'ordre.

Par exemple, si l'on veut avoir la liste des prénoms et des noms des candidats habitant Paris ou Lyon, classés suivant le nom, il suffit d'effectuer la requête suivante :

```
SELECT nom, prenom
FROM candidats
WHERE (ville = 'Paris') OR (ville = 'Lyon')
ORDER BY nom
```

dont le résultat est :

nom	prenom
Dupond	Pierre
Leroy	Marie
Martin	Julien

Utilisation d'une fonction d'agrégation statistique

Les fonctions d'agrégations sont utilisées pour obtenir des statistiques portant sur enregistrements. Les fonctions principales sont :

- AVG() calculant la moyenne d'un ensemble de valeurs (numériques) ;
- COUNT() comptant le nombre de lignes concernées ;
- MAX() donnant la plus haute valeur (numérique) ;
- MIN() donnant la plus petite valeur (numérique) ;
- SUM() calculant la somme sur plusieurs lignes d'une valeur (numérique).

Par exemple :

```
SELECT AVG(age)
FROM candidats
```

donnera l'âge moyen des candidats, ici 19 (ou peut-être 20) et non 19,6 car l'âge est un entier.

Sous-requête

Une *sous-requête* (aussi appelée « requête imbriquée » ou « requête en cascade ») consiste à exécuter une requête à l'intérieur d'une autre requête.

L'exemple ci-dessous est une exemple typique de sous-requête qui retourne un seul résultat à la requête principale :

```
SELECT *
FROM table
WHERE nom_colonne = (
                     SELECT valeur
                     FROM table2
                     LIMIT 1
                    )
```

Une requête imbriquée peut également retourner une colonne entière. La requête externe peut alors utiliser la commande IN pour filtrer les lignes qui possèdent une des valeurs retournées par la requête interne. L'exemple ci-dessous met en évidence un tel cas de figure :

```
SELECT *
FROM table
WHERE nom_colonne IN (
      SELECT colonne
      FROM table2
      WHERE cle_etrangere = 36
      )
```

Existence d'une ligne vérifiant une condition

La commande EXISTS s'utilise dans une clause conditionnelle pour savoir s'il y a présence d'au moins une ligne lors de l'utilisation d'une sous-requête.

L'utilisation basique de la commande EXISTS consiste à vérifier si une sous-requête retourne un résultat ou non, en utilisant EXISTS dans la clause conditionnelle. La requête externe s'exécutera uniquement si la requête interne retourne au moins un résultat :

```
SELECT nom_colonne1
FROM table1
WHERE EXISTS (
     SELECT nom_colonne2
     FROM table2
     WHERE nom_colonne3 = 10
     )
```

On peut de même utiliser NOT EXISTS.

Variantes *Oracle* de SQL

Chaque implémentation de SQL apporte son lot de variantes. Nous avons déjà vu CREATE DATABASE. Parmi les variantes dues à *Oracle*, citons :

— OWNER qui désigne celui qui est propriétaire de la table ;
— `all_catalog` qui permet de manipuler toutes les tables accessibles par l'utilisateur en cours.

10.3 PL/SQL

Motivation

Les requêtes SQL correspondent à des instructions primitives. On peut utiliser SQL de façon interactive ou mixer les requêtes et les structures de contrôle d'un langage de programmation afin d'obtenir toute la puissance des algorithmes.

PL/SQL (sigle de *Procedural Language / Structured Query Language*) est un langage de programmation propriétaire, créé par *Oracle*, utilisé dans le cadre de bases de données relationnelles.

Sa syntaxe générale s'inspire du langage de programmation Ada. Un programme est constitué de procédures et de fonctions. Des variables permettent l'échange d'information entre les requêtes SQL et le reste du programme

Syntaxe

Nous avons rencontré la syntaxe du langage Ada dans le chapitre précédent. Voyons quelques variantes, rencontrées dans la portion de code communiquée par le ministère.

Commentaires.- Le double tiret `--` annonce un *commentaire de fin de ligne* en PL/SQL, c'est-à-dire que ce qui suit (jusqu'à la fin de la ligne) n'est pas pris en compte par le compilateur.

Structure d'un bloc.- Un programme est structuré en *blocs*, imbriqués les uns dans les autres. La structure d'un bloc est :

```
DECLARE
-- définitions de variables
BEGIN
-- Les instructions à exécuter
EXCEPTION
-- La récupération des erreurs
END;
```

où seuls BEGIN et END sont obligatoires. Les blocs, comme les instructions, se terminent par un point virgule « ; ».

Identificateurs.- Les identificateurs Oracle ont 30 caractères au plus, commencent par une lettre, peuvent contenir des lettres, des chiffres et les caractères spéciaux « _ », « $ » et « # ». Les lettres sont les 26 lettres de l'alphabet latin pur, pour lesquelles on ne fait pas la différence entre minuscule et majuscule.

Types.- Les types habituels sont les types SQL2 ou *Oracle* : `integer`, `varchar`... à compléter par

Pierre

De façon spécifique, il existe des types composites adaptés à la récupération des colonnes et des lignes des tables SQL :

— le type de la colonne `name` de la table `table` est `table.name.%TYPE` ;
— le type de toutes les colonnes d'une ligne de la table `table` est `table.%ROWTYPE`.

Déclaration d'une variable.- Suivant Ada, toute variable doit être déclarée avant d'être utilisée, en suivant la syntaxe :

```
identificateur [CONSTANT] type [:= valeur];
```

par exemple :

```
age integer;
nom varchar(30);
```

Cette syntaxe ne permet pas les déclarations multiples telles que :

```
i, j integer;
```

Affectation.- Il existe deux façons de donner une valeur à une variable :

— soit en utilisant l'opérateur d'affectation « `:=` » d'Ada ;
— soit en utilisant la directive INTO de la requête SELECT de SQL.

Par exemple :

```
age := 20;
```

```
SELECT nom INTO name
FROM candidats
WHERE identifiant = 4;
```

affecte la valeur 'Bernard' à la variable `name`.

On peut utiliser plusieurs variables :

```
SELECT expr1, expr2... INTO var1, var2...
```

où `expr` est le nom d'une colonne, une constante ou un agrégat.

La condition WHERE doit sélectionner une ligne et une seule. Dans le cas contraire, on tombe sur l'une des deux exceptions suivantes :

— si le SELECT renvoie plus d'une ligne, une exception TOO_MANY_ROWS est levée ;
— si le SELECT ne renvoie aucune ligne, une exception NO_DATA_FOUND est levée.

Branchements.- À côté des trois genres habituels de branchement, à savoir :

— le test :

```
IF condition THEN
   -- instructions
END IF;
```

— l'alternative :

```
IF condition THEN
   -- instructions1
ELSE
   -- instructions2
END IF;
```

— et les alternatives imbriquées :

```
IF condition1 THEN
   -- instructions1
ELSEIF condition2 THEN
   -- instructions2
ELSEIF ...
   ...
ELSE
   -- instructionsN
END IF;
```

on peut aussi utiliser la fonction `DECODE()` à un nombre variable d'arguments :

`DECODE(`*expression*`,` *search1*`,` *result1* `[,` *search2*`,` *result2*`]... [,` *defaultResult*`])`

où :

expression est la valeur à comparer ;
les *searchi* sont les valeurs comparées à *expression* ;
resulti est la valeur renvoyée si la valeur de *expression* est égale à *searchi*.

On compare ainsi *expression* à un certain nombre de valeurs *search*. Si *expression* est égale à une des valeurs *searchi* alors la fonction renvoie la valeur *resulti* correspondante. Si *expression* n'est égale à aucune de ces valeurs *searchi* alors la fonction renvoie la valeur *defaultResult*.

Boucle « tant que ».- Sa syntaxe est :

```
WHILE condition LOOP
   -- instructions
END LOOP;
```

Boucle « pour ».- Sa syntaxe est :

```
FOR compteur IN [REVERSE] inf..sup LOOP
   -- instructions
END LOOP;
```

Par exemple :

```
FOR i IN 1..100 LOOP
   somme := somme + i;
END LOOP;
```

Boucle « exit ».- La boucle :

```
LOOP
   -- instructions
END LOOP;
```

boucle *a priori* indéfiniment.

En fait elle peut prendre fin, plus exactement la première fois que l'on rencontre, lors de l'exécution, une instruction :

```
EXIT [WHEN condition];
```

avec la condition `condition` vérifiée (si celle-ci, facultative, apparaît).

Les curseurs.- Toutes les requêtes SQL sont associées dans PL/SQL à un **curseur**, c'est-à-dire une zone mémoire.

Un curseur peut être *implicite* (c'est-à-dire non déclaré par l'utilisateur) ou explicite. Les curseurs implicites sont ceux que nous avons vu lors de l'affectation spécifique à PL/SQL. Nous avons vu qu'ils doivent sélectionner une ligne unique. Les curseurs explicites permettent de sélectionner plusieurs lignes.

Les curseurs implicites se désignent tous par 'SQL' si besoin est.

Les curseurs ont des attributs :

— %ROWCOUNT est le nombre de lignes traitées par le curseur ;
— %FOUND est vrai si au moins une ligne a été traitée par la requête ;
— %NOTFOUND est vrai si aucune ligne n'a été traitée par la requête ;
— %ISOPEN est vrai si le curseur (explicite) est ouvert.

Par exemple `blabla.%ROWCOUNT` est le nombre de lignes du curseur `blabla`.

Un curseur explicite est déclaré par le mot clé CURSOR, suivi d'un identifiant, du mot clé `IS`, puis d'une requête SQL et se se termine par un point virgule.

Puisqu'un curseur explicite peut renvoyer plusieurs lignes, il sera en général utilisé dans une boucle « Pour » :

```
FOR c IN blabla LOOP
   -- instructions
END LOOP;
```

où `blabla` est le nom du curseur et `c` est du type `blabla%ROWTYPE`.

Les exceptions.- Il existe deux genres d'exceptions en PL/SQL : celles prédéfinies par *Oracle* et celles définies par le programmeur. Les exceptions prédéfinies sont NO_DATA_FOUND, TOO_MANY_ROWS, VALUE_ERROR (erreur arithmétique), ZERO_DIVIDE...

Les exceptions prédéfinies sont utilisées de la façon suivante :

```
BEGIN ...
EXCEPTION
   WHEN NO_DATA_FOUND THEN
...
   WHEN TOO_MANY_ROWS THEN
...
   WHEN OTHERS THEN -- optionnel
...
END;
```

Sous-programmes.- Un *bloc anonyme* PL/SQL est un bloc « DECLARE – BEGIN – END ». Le plus souvent, on crée une procédure ou une fonction nommée pour réutiliser le code.

La définition d'une procédure est :

```
PROCEDURE nom (<liste params>) IS
   -- déclaration des variables
BEGIN
   -- code de la procédure
END;
```

les variables étant déclarées entre IS et BEGIN.

La définition d'une fonction est :

```
FUNCTION nom (<liste params>)
RETURN <type retour> IS
   -- déclaration des variables
BEGIN
   -- code de la fonction
END;
```

On indique le type de passage des paramètres que l'on veut voir appliquer en mettant l'un des mots clés suivant entre l'identificateur et le type :

- IN pour le passage par valeur ;
- IN OUT pour le passage par référence ;
- OUT pour le passage par référence mais pour un paramètre dont la valeur n'est pas utilisée en entrée.

Remplacement d'une valeur nulle par une valeur.- La fonction `NVL()`, pour *Null VaLue*, permet de remplacer une occurrence de la valeur NULL par une autre valeur (non nulle). Sa syntaxe est :

```
NVL(exp, replace_with)
```

où :

`exp` est l'expression à tester ;
`replace_with` la valeur renvoyée si `exp` est NULL.
Cette fonction ne peut être placée que dans ce qui est renvoyé par un `SELECT`.

Génération aléatoire. La fonction `DBMS_RANDOM(a, b)`, à deux arguments, permet d'obtenir un nombre aléatoire compris entre a et b.