Chapitre 3
La manipulation des données (LMD)

1. Introduction

Le langage de manipulation de données permet aux utilisateurs et aux développeurs d'accéder aux données de la base, de modifier leur contenu, d'insérer ou de supprimer des lignes.

Il s'appuie sur quatre ordres de base qui sont SELECT, INSERT, DELETE et UPDATE.

Ces quatre ordres ne sont pas toujours autorisés par l'administrateur de la base qui est le seul à pouvoir attribuer ou non les droits d'utilisation sur ces ordres.

Pour l'utilisateur lambda, il pourra indiquer que seul l'ordre SELECT est utilisable. Les ordres de modification de la base ne sont pas accessibles pour certains utilisateurs pour des raisons évidentes de sécurité.

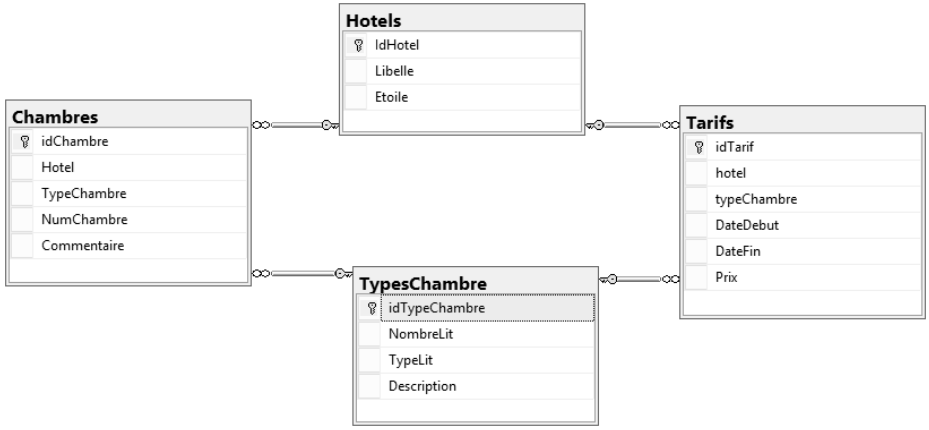
2. La sélection de données

L'ordre SELECT permet de réaliser des requêtes simples assez rapidement même sans connaissances approfondies en langage de programmation. C'est l'ordre de base qui permet d'indiquer au serveur que l'on désire extraire des données.

Il peut également être très puissant si l'on connaît toutes les fonctions et toutes les possibilités du langage. On peut réaliser des requêtes complexes, avec de nombreuses tables mais il faut toujours faire attention aux performances qui peuvent se dégrader très rapidement sur un ordre SQL mal construit ou n'utilisant pas les bons index dans les tables. Il faut être vigilant et utiliser les outils d'analyse de requête (cf. chapitre Approfondissement - Quelques notions de performances) avant d'exécuter une requête sur une base réelle avec des tables conséquentes.

Les principaux éléments d'une requête de sélection	
Clause	**Expression**
SELECT	Liste colonne(s) et ou éléments d'extraction
FROM	Table(s) source(s)
WHERE	Condition(s) ou restriction(s), optionnelle
GROUP BY	Regroupement(s), optionnelle
HAVING	Condition(s) ou restriction(s) sur le(s) regroupement(s), optionnelle
ORDER BY	Tri(s)

Les tables de base qui sont utilisées dans les sections suivantes sont celles-ci :



2.1 L'ordre de sélection de données SELECT

Le SELECT est l'ordre le plus important et le plus utilisé en SQL. Avec cet ordre, nous pouvons ramener des lignes d'une ou plusieurs tables mais également transformer des données par l'utilisation de fonction ou encore réaliser des calculs.

Nous allons décrire progressivement les possibilités de cet ordre dans les paragraphes suivants.

L'utilisation la plus courante consiste à sélectionner des lignes dans une table comme ceci :

```
SELECT NombreLit, Description FROM TypesChambre;
```

Dans cet exemple, nous avons sélectionné deux colonnes de la table TypesChambre.

L'ordre va donc nous ramener toutes les lignes de la table pour ces deux colonnes.

Si on avait voulu toutes les colonnes et toutes les lignes de la table, l'ordre aurait été celui-ci :

```
SELECT * FROM TypesChambre;
```

idTypeChambre	NombreLit	TypeLit	Description
1	1	lit simple	1 lit simple avec douche
2	2	lit simple	2 lits simples avec douche
3	2	lit simple	2 lits simples avec douche et WC séparés
4	1	lit double	1 lit double avec douche
5	1	lit double	1 lit double avec douche et WC séparés
6	1	lit double	1 lit double avec bain et WC séparés
7	1	lit XL	1 lit double large avec bain et WC séparés

L'étoile est pratique lorsque l'on ne connaît pas le nom des colonnes, mais le résultat est rarement lisible avec des tables contenant un nombre important de colonnes. Pour connaître le nom des colonnes, faites un DESC de la table auparavant (DESC <nom table>).

La syntaxe simple est donc :

```
SELECT <colonne 1>, <colonne 2> ... | * FROM <table1>, <table2> ...
```

Si certaines colonnes ont le même nom mais appartiennent à des tables différentes, il faudra ajouter le nom de la table devant la colonne afin que le système sache quelle colonne prendre.

Il n'est pas obligatoire de mettre le nom des tables devant chaque colonne, mais pour une question de lisibilité et de maintenance, il est préférable de les mettre sur des sélections complexes.

```
SELECT Hotels.Libelle
, Hotels.Etoile
, Chambres.NumChambre
, TypesChambre.Description
FROM Chambres, Hotels, TypesChambre;
```

Nous verrons dans la section L'utilisation des alias de ce chapitre que pour alléger la lecture il est également possible de donner un alias à chaque table. Cet alias est souvent simple et permet de retrouver facilement la table concernée, par exemple CH pour Chambres ou TYP pour TypesChambre.

2.2 Les options DISTINCT et ALL

Par défaut, lors de l'exécution d'un SELECT, toutes les lignes sont ramenées (l'option ALL est automatique). Si l'on veut supprimer les doublons, il faut ajouter l'ordre DISTINCT.

L'ordre DISTINCT s'applique à toutes les colonnes présentes.

Exemple

```
SELECT NombreLit, TypeLit, Description FROM TypesChambre;
```

et :

```
SELECT DISTINCT NombreLit, TypeLit, Description FROM TypesChambre;
```

Les deux SELECT ci-dessus ont le même résultat car il y a un doublon sur les trois premières lignes. Ces deux premières colonnes sont identiques mais pas la troisième.

NombreLit	TypeLit	Description
1	**lit double**	1 lit double avec bain et WC séparés
1	**lit double**	1 lit double avec douche
1	**lit double**	1 lit double avec douche et WC séparés
1	lit simple	1 lit simple avec douche
1	lit XL	1 lit double large avec bain et WC séparés

En revanche, si on réduit la sélection à deux colonnes comme :

```
SELECT NombreLit, TypeLit FROM TypesChambre;
```

on obtient :

NombreLit	TypeLit
1	lit simple
2	lit simple
2	lit simple
1	lit double
1	lit double

Si on ajoute un ordre DISTINCT, une des deux lignes contenant '2' et 'lit simple' sera supprimée.

```
SELECT DISTINCT NombreLit, TypeLit FROM TypesChambre;
```

Remarque

La clause DISTINCT ne peut pas être utilisée avec des opérateurs de regroupement (voir le GROUP BY). En effet, les opérateurs de type COUNT ou SUM éliminent automatiquement les doublons.

NombreLit	TypeLit
1	lit double
1	lit simple
1	lit XL
2	lit simple

2.3 Les tris

Lorsque l'on ramène des colonnes d'une ou de plusieurs tables avec un SELECT, il est souvent intéressant d'obtenir un résultat trié sur certaines colonnes.

Pour cela, on utilisera la clause ORDER BY en fin de requête. On peut trier sur n'importe quelles colonnes d'une table, il faut pour cela que les colonnes fassent partie de la sélection, mais elles ne sont pas obligatoirement affichées.

La clause ne peut être utilisée qu'une seule fois dans une requête et doit toujours être la dernière clause de la requête.

Le tri par défaut est ascendant, noté ASC (du plus petit au plus grand). Il est possible d'indiquer que l'on désire réaliser le tri en descendant en notant DESC.

Syntaxe de l'ORDER BY

```
ORDER BY <colonne 1> [ASC|DESC], <colonne 2> [ASC|DESC]...
```

Exemple

Tri sur la DateDebut de la table Tarifs en ascendant et tri sur le prix de la table en descendant.

```
SELECT hotel
, typeChambre
, DateDebut
, prix
FROM Tarifs
ORDER BY DateDebut, prix DESC;
```

Hotel	typeChambre	DateDebut	prix
1	7	01/04/2021	103,49
4	7	01/04/2021	103,49
4	6	01/04/2021	91,99
1	6	01/04/2021	91,99
1	5	01/04/2021	80,49
4	3	01/04/2021	80,49

On constate que le tri se fait sur la colonne prix du plus grand au plus petit (DESC).

Il existe également la possibilité d'indiquer l'ordre de la colonne dans le SELECT à la place de son nom, à condition que la colonne soit présente dans la sélection.

Exemple sans le nom des colonnes

```
SELECT hotel
, typeChambre
, DateDebut
, prix
FROM Tarifs
ORDER BY 3, 4 DESC;
```

Certes, cette notation fonctionne et permet de ne pas ressaisir le nom des colonnes, mais n'aide vraiment pas à la lisibilité de la requête. Avec des sélections multiples, l'ordre devient vite illisible pour celui qui ne l'a pas écrit.

Remarque

Attention : le tri peut être source de baisse de performance de la requête. En effet, si la requête ramène des millions de lignes, le système va récupérer toutes les lignes dans un premier temps, puis trier l'ensemble et ensuite seulement commencer à restituer les éléments dans l'ordre. Le système va utiliser de l'espace disque pour stocker les éléments à trier puis de la mémoire pour réaliser le tri.

Le tri est déconseillé dans le cas où il n'y a pas de clause WHERE ou que celle-ci est peu restrictive et que la table contient des millions d'enregistrements.

Chapitre 6
Fonctions

1. Fonctions et procédures

Les fonctions et procédures stockées sont des outils permettent d'implémenter des traitements fonctionnels dans la base de données, au plus proche des objets et des données. L'utilisation de cet outil est souvent motivée par la volonté de rapprocher le traitement des données, plutôt que de devoir extraire les données dans une application, d'y appliquer le traitement pour finalement écrire le résultat dans cette même base de données. Il s'agit donc d'un choix d'architecture logicielle qui dépend de ce que l'on souhaite faire des données, notamment leur destination : le choix d'utiliser ou non des fonctions et procédures stockées dépend en partie du besoin d'écrire les résultats ou de communiquer les données et les résultats à des systèmes externes à l'application.

Les fonctions et procédures stockées peuvent utiliser différents langages : intégrés à PostgreSQL comme SQL ou PL/pgSQL ou utilisant des processus externes, comme Python. De nombreuses extensions permettent d'intégrer d'autres langages comme Java, R ou PHP.

Si les fonctions stockées sont utilisables depuis longtemps dans PostgreSQL, la notion de procédure stockée apparaît avec la version 12 : elle apporte la possibilité de valider les modifications de données (COMMIT) pendant l'exécution de la procédure.

1.1 Fonctions

Les fonctions sont créées avec la commande `CREATE FUNCTION`, dont le synopsis est le suivant :

```
CREATE [ OR REPLACE ] FUNCTION
   nom( [ [ mode_arg ] [ nom_arg ] type [ { DEFAULT | = }
expr ] [, ...] ] )
   [ RETURNS type
     | RETURNS TABLE ( nom_colonne type [, ...] ) ]
 { LANGUAGE nom_langage
   | IMMUTABLE | STABLE | VOLATILE
   | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
   | SECURITY INVOKER | SECURITY DEFINER
   | COST execution_cost
   | ROWS result_rows
 } ...
```

Le nom de la fonction peut être qualifié avec un schéma, comme pour une table ou une vue. La clause `OR REPLACE` permet de remplacer une fonction existante.

Les arguments de la fonction sont composés de quatre termes : le mode, entrée ou sortie, le nom, qui est optionnel, le type de données qui est obligatoire, et la valeur par défaut qui peut utiliser une valeur ou une expression :

– Le mode désigne l'utilisation de l'argument en entrée (IN), en sortie (OUT) ou les deux (INOUT). Par défaut, un argument est utilisé en entrée. Le mode VARIADIC permet de créer un argument ayant un nombre variable de valeurs. La variable implicite est alors un tableau de données.

– Le nom est utilisé dans la fonction comme une variable locale implicite. Lorsque les arguments n'ont pas de nom, ils sont automatiquement nommés avec une séquence commençant par **$** : $1, $2, $3, etc. Lorsqu'ils sont nommés, il est possible de les appeler par leur nom à l'appel de la fonction.

– Le type de données est l'un de ceux utilisés dans PostgreSQL, comme dans les tables, et permet de définir le prototype de la fonction.

– L'expression par défaut alimente l'argument avec une valeur pour le résultat d'une expression, comme par exemple une fonction retournant l'heure courante. Ceci permet d'appeler la fonction sans indiquer de valeur en entrée pour les arguments, et ainsi de rendre la fonction polymorphe.

La fonction retourne une donnée typée, un tuple ou un ensemble de tuples, et donc une relation.

Par défaut, une fonction est VOLATILE, c'est-à-dire qu'elle peut modifier des données, et que le résultat peut être différent pour une même donnée en entrée. Le mode STABLE indique que même si des données peuvent être lues dans des tables, pour une requête dans laquelle la fonction est appelée, le résultat est le même pour une même donnée en entrée. Le mode IMMUTABLE indique que la fonction ne lit ni ne modifie de données dans les tables, et que, pour une requête dans laquelle la fonction est appelée, le résultat est le même pour une même donnée en entrée.

Par défaut, la fonction est exécutée même si les données en entrée sont nulles, avec la clause `CALLED ON NULL INPUT` : le code de la fonction doit alors tenir compte du fait que les données peuvent être nulles. La clause `RETURNS NULL ON NULL INPUT` ou la clause `STRICT` indiquent que si une donnée, parmi les arguments en entrée, est nulle, alors la fonction n'est pas appelée et retourne `NULL`.

Une fonction est par défaut exécutée avec les droits du rôle qui l'appelle, avec la clause `SECURITY INVOKER`. Au contraire, la clause `SECURITY DEFINER` permet d'exécuter la fonction avec les droits du rôle propriétaire de la fonction, quel que soit le rôle appelant la fonction.

Enfin, les clauses `COST` et `ROWS` sont utiles pour définir le coût d'appel d'une fonction, ce qui ne sert que lorsque la fonction retourne un ensemble de valeurs.

1.2 Langage SQL

Le langage SQL est utilisable dans le cadre d'une fonction, ce qui permet d'encapsuler une requête SQL, par exemple une expression couramment utilisée : concaténation de chaîne, traitement de nombres, formatage d'objet, et tant d'autres possibilités.

L'exemple suivant permet de concaténer des chaînes de caractères, en intercalant un caractère particulier :

```
CREATE OR REPLACE FUNCTION concerts.concat_artiste( text, text )
 RETURNS text
 LANGUAGE SQL
 AS $$
 SELECT $1 || ' / ' || $2
 $$ ;
```

La fonction peut être utilisée dans une requête `SELECT` :

```
SELECT concerts.concat_artiste('Noah', 'Maurane') ;
┌─────────────────┐
│ concat_artiste  │
├─────────────────┤
│ Noah / Maurane  │
└─────────────────┘
```

La fonction peut alors être appelée dans une requête créant un concert. L'exemple suivant utilise des paramètres nommés, et utilise ces noms dans le code de la fonction :

```
CREATE OR REPLACE FUNCTION concerts.concat_artiste
( art_nom1 text, art_nom2 text )
 RETURNS text
 LANGUAGE SQL
 AS $$
 SELECT art_nom1 || ' / ' || art_nom2
 $$ ;
```

De cette façon, le code de la fonction est plus lisible. Avec la même fonction, il est possible de définir la chaîne de caractères servant de séparation, tout en ayant une valeur par défaut :

```
CREATE OR REPLACE FUNCTION concerts.concat_artiste
   ( art_nom1 text, art_nom2 text, sep text default ' / ' )
 RETURNS text
 LANGUAGE SQL
 AS $$
 SELECT art_nom1 || sep || art_nom2
 $$ ;
```

La création de cette fonction fait qu'il existe deux fonctions `concat_artiste()`, avec deux prototypes différents, ce qui rend ambigüe l'utilisation de la première forme, qu'il faut supprimer :

```
SELECT concerts.concat_artiste('Noah', 'Maurane' ) ;
ERROR:  function tickets.concat_artiste(unknown, unknown) is not unique
LINE 1: select tickets.concat_artiste('Noah', 'Maurane' )
               ^
HINT:  Could not choose a best candidate function. You might need to
add explicit type casts.
```

La suppression de la première version de la fonction est possible par la commande `DROP FUNCTION` :

```
DROP FUNCTION concerts.concat_artiste( text, text) ;
```

L'appel de la fonction restante est alors possible de deux façons différentes :

```
SELECT concerts.concat_artiste('Noah', 'Maurane');
┌─────────────────┐
│ concat_artiste  │
├─────────────────┤
│ Noah / Maurane  │
└─────────────────┘

SELECT tickets.concat_artiste('Noah', 'Maurane', ' ; ');
┌─────────────────┐
│ concat_artiste  │
├─────────────────┤
│ Noah ; Maurane  │
└─────────────────┘
```

La fonction peut être déclarée `STRICT`, dans ce cas, car c'est de toute façon le comportement de la contaténation de chaînes de caractères : lorsqu'une chaîne de caractères en entrée est nulle, alors la donnée retournée est nulle. On économise dans ce cas l'appel de la fonction.

De plus, la fonction peut être déclarée `IMMUTABLE`, car elle ne modifie ni ne lit de données dans une table, et retourne toujours le même résultat pour un même ensemble de données en entrée.

La déclaration de la fonction est alors :

```
CREATE OR REPLACE FUNCTION concerts.concat_artiste
   ( art_nom1 text, art_nom2 text, sep text default ' / ' )
 RETURNS text
 LANGUAGE SQL
 STRICT IMMUTABLE
 AS $$
 SELECT art_nom1 || sep || art_nom2
 $$ ;
```

Les paramètres étant nommés, il est possible de les appeler par leurs noms à l'exécution de la fonction, et ainsi de les utiliser dans n'importe quel ordre. La valeur est affectée avec l'opérateur :=, comme dans les deux exemples suivants :

```
SELECT concerts.concat_artiste
   ( art_nom1 := 'Noah', art_nom2 := 'Maurane');
┌────────────────┐
│ concat_artiste │
├────────────────┤
│ Noah / Maurane │
└────────────────┘

SELECT concerts.concat_artiste
   ( art_nom2 := 'Noah', art_nom1 := 'Maurane');
┌────────────────┐
│ concat_artiste │
├────────────────┤
│ Maurane / Noah │
└────────────────┘
```

Dans ce cas, les valeurs sont inversées dans le résultat.

2. Langage PL/pgSQL

Le langage PL/pgSQL est un module optionnel de PostgreSQL, livré avec PostgreSQL, et pré-installé en tant qu'extension dans les bases de données. Il s'agit d'un langage interprété, utilisé dans une fonction ou une procédure, permettant de manipuler les données à travers le langage SQL. En effet, ce dernier est directement utilisable dans le code PL/pgSQL.

L'exemple suivant réécrit l'exemple de la fonction SQL précédente, avec le langage PL/pgSQL :

```
CREATE OR REPLACE FUNCTION tickets.concat_artiste
 ( art_nom1 text, art_nom2 text
  , sep text default ' / ' )
 RETURNS text
 LANGUAGE plpgsql
 STRICT IMMUTABLE
AS $$
DECLARE
 v_ret text ;
BEGIN
 v_ret := art_nom1 || sep || art_nom2 ;

 return v_ret ;
END
$$ ;
```

Le code est composé de deux blocs :

– le bloc `DECLARE` permet de déclarer des variables internes à la fonction,

– le bloc `BEGIN` débute le code exécutable, qui se termine par l'instruction `END`.

Le code de la fonction affecte le résultat de la concaténation des chaînes de caractères dans la variable `v_ret`, puis cette variable est utilisée par l'instruction `return` à la fin de la fonction.