SORBONNE UNIVERSITÉ

Implémentation d'une Base de Données

Le langage SQL

Cyril GALLUT
UMR 7205
Institut de Systématique, Évolution, Biodiversité.

Sophie PASEK
UMR 7205
Institut de Systématique, Évolution, Biodiversité.

Bernard BILLOUD
UMR 8227
Laboratoire de Biologie Intégrative des Modèles Marins.

15 – 18 mars 2021

Plan du cours

" I'd have you lot up in front of the University authorities
first thing in the morning, if it wasn't for the fact
that you are the University authorities…"
Terry PRATCHETT

Introduction

Le langage SQL est utilisé pour construire et manipuler les bases de données relationnelles. Il est basé sur l'algèbre relationnelle. C'est un langage déclaratif, contrairement à la plupart des langages qui sont procéduraux. Il existe plusieurs standards qui sont plus ou moins disponibles dans les différents SGBDR.

— SQL-86

— SQL-89 ou SQL-1

— SQL-92 ou SQL2

— SQL-99 ou SQL3

— SQL :2003

— SQL :2008

— SQL :2011

Le SGBDR utilisé ici est PostgreSQL, c'est un SGBDR avancé et robuste, disponible sur de nombreuses plateformes et développé sous forme de logiciel libre. Comme la plupart des SGBD, PostgreSQL fonctionne avec une architecture client/serveur. La partie serveur est généralement installé sur une machine serveur accessible localement ou par le réseau. Le serveur postgres peut gérer de nombreuses bases en même temps et il gère également l'accès concurrentiel des utilisateurs. Il existe plusieurs clients qui permettent de consulter une base sous postgres :

— psql, en ligne de commande,

— pgAdmin, avec une interface graphique,

— phpPgAdmin, est un client web,

— ...

Le client peut tourner sur la même machine que le serveur ou sur une machine distante et il peut évidemment tourner sur un système différent.

Structure lexicale

Une commande SQL ou requête est composée d'une séquence de mots, séparés par des blancs, terminé par un point-virgule. Les mots peuvent être : un *mot clef*, un *identifieur*, un *littéral* (une constante) ou un *caractère spécial*. De plus une commande SQL peut contenir des *commentaires*, un commentaire commence par deux tirets (`--`) et se poursuit jusqu'à la fin de la ligne.

Les *mots clefs* sont tous les mots qui ont une signification dans le langage SQL, par exemple `SELECT`, `UPDATE`, `DELETE` etc.. Les mots clefs peuvent s'écrire indifféremment en minuscules ou en majuscules, par convention et pour faciliter la lecture on les met en majuscules.

Les *identifieurs* identifient les objets de la base : tables, colonnes, contraintes etc.. Les identifieurs sont toujours considérés comme écrits en minuscules sauf s'ils sont entourés de guillemets (`"`) : `ReGioNs` équivaut à `regions` mais `"Regions"` est un objet différent.

Les *constantes* peuvent être de type chaîne ou numérique. Une constante de type chaîne en SQL est une séquence arbitraire de caractères entourée par des apostrophes (`'`). Une constante numérique n'est pas entouré d'apostrophes : `42`, `5.01`, `1.925e-3`.

Les *caractères spéciaux* (`$ () [] , ; * . --`) ont une signification spéciale qui sera vue au fur et à mesure. Par exemple le `;` délimite la fin d'une commande. Il existe également un ensemble de caractères qui servent d'opérateurs :
`( + - * / < > = ~ ! @ # % ^ & | ` ?`).
Par exemple : - pour la soustraction.

Types de données

Les données peuvent êtres de différentes natures :

Entiers : `0, 10, 3, 527, 19` etc.,

Réels : `3.67, 3/7, 3.14` etc.,

Chaînes de caractères : `'toto', 'titi', 'Gros minet'` etc.,

Date : `'2009-09-28'`,

Heure : `'10h05'`,

Booléens : `Vrai, Faux`.

Les types numériques disponibles dans PostgreSQL sont regroupés dans le tableau 1. Le type `serial` s'auto-incrémente pour chaque nouvel enregistrement, cela permet de construire une clef primaire automatiquement. Les types chaînes disponibles dans PostgreSQL sont regroupés dans le tableau 2. Pour les types dates et heures voir le tableau 3.

TABLEAU 1 – Types numériques disponibles dans PosgreSQL.

Nom	Taille de stockage	Description	Étendue
smallint	2 octets	entier de faible étendue	de -32768 à +32767
int, integer	4 octets	entier habituel	de -2147483648 à +2147483647
bigint	8 octets	grand entier	de -9223372036854775808 à 9223372036854775807
decimal, numeric	variable	précision indiquée par l'utilisateur, valeur exacte	pas de limite
real	4 octets	précision variable, valeur inexacte	précision de 6 décimales
double precision, float	8 octets	précision variable, valeur inexacte	précision de 15 décimales
serial	4 octets	entier à incrémentation automatique	de 1 à 2147483647
bigserial	8 octets	entier de grande taille à incrémentation automatique	de 1 à 9223372036854775807

TABLEAU 2 – Types chaînes disponibles dans PosgreSQL.

Nom	Description
character varying(n), varchar(n)	Longueur variable avec limite
character(n), char(n)	longueur fixe, complété par des espaces
text	longueur variable illimitée

TABLEAU 3 – Types date et heure disponibles dans PosgreSQL.

Nom	**Description**	**Précision**	Exemple
timestamp	Date et heure	1 microseconde	05-Nov-2009 09 :05 :06.456
date	Date seule	1 jour	05-11-2009
time	Heure seule	1 microseconde	10 :37 :06.731

PostgreSQL fournit le type `boolean` du standard SQL. Ce type ne dispose que de deux états : « true » (vrai) et « false » (faux). Les libellés valides sont

pour l'état « vrai » :	pour l'état « faux » :
`TRUE`	`FALSE`
`'t'`	`'f'`
`'true'`	`'false'`
`'y'`	`'n'`
`'yes'`	`'no'`
`'1'`	`'0'`

PostgreSQL fournit de nombreux autres types tels que les types trigonométriques, les tableaux, le type xml, le type chaînes de bits ainsi que la possibilité de définir des types personnalisés.

Sous catégories de langages

Le langage SQL est constitué de plusieurs langages, que l'on va voir en détails :

— un langage de manipulation de données,

— un langage de définition de données,

— un langage de contrôle de données,

— un langage de contrôle de transactions.

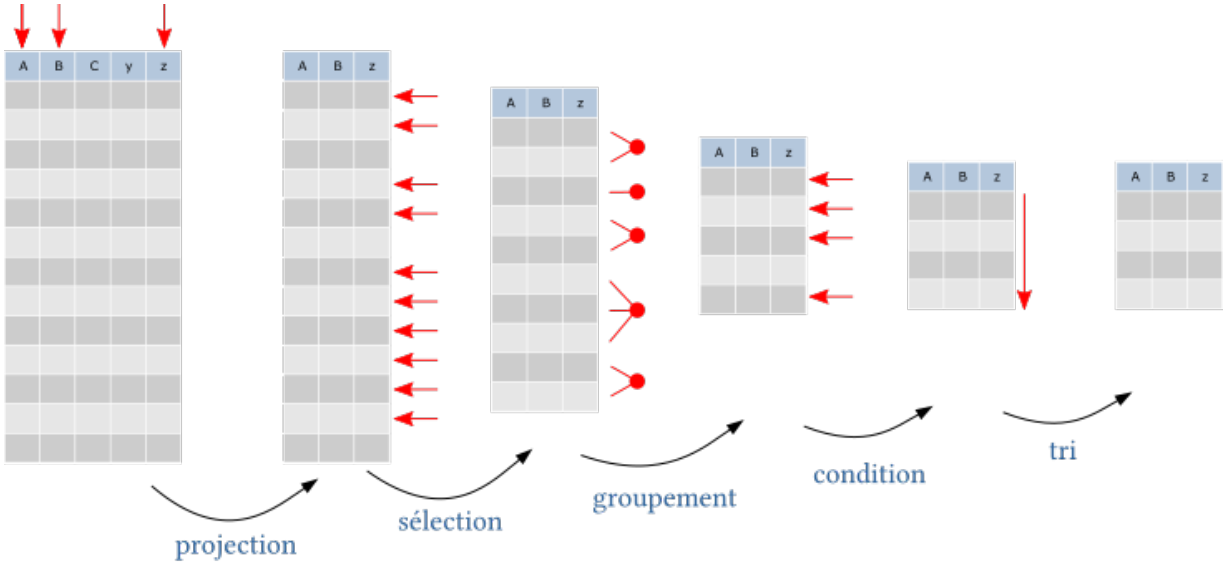
1 Langage de manipulation de données

La manipulation des données comprends la consultation des données, l'insertion et la suppression de données et la mise à jour.

1.1 Consultation des données

La commande `SELECT` permet de retrouver les données selon différents critères à partir d'une ou plusieurs tables. Voici la syntaxe générale :

```
SELECT liste_attributs
    FROM nom_table
    WHERE condition_de_selection_de_lignes
    GROUP BY liste_attributs
    HAVING condition_de_selection_de_groupements
    ORDER BY liste_attributs;
```



projection ..... `SELECT liste_attributs`
`FROM table`
sélection ..... `WHERE condition_de_selection_de_lignes`
groupement ..... `GROUP BY liste_attributs`
condition ..... `HAVING condition_de_selection_de_groupements`
tri ..... `ORDER BY liste_attributs;`

Requête simple :

```
SELECT * FROM etudiants;
```

Le caractère spécial $\star$ permet de prendre en compte tous les attributs de la table. Elle envoie tous les attributs et toutes les lignes de la table `etudiants`.

Il est possible de donner des alias aux identifieurs pour faciliter l'écriture de la requête et la lecture des résultats avec la clause `AS` :

```
SELECT nom AS "Nom étudiant" FROM etudiants AS e;
```

Une requête peut renvoyer plusieurs lignes identiques, pour éviter cela on utilise la clause `DISTINCT` :

```
SELECT DISTINCT * FROM nom_table;
```

1.1.1 Conditions de sélection

La clause `WHERE` permet de spécifier les lignes à sélectionner selon un critère particulier. Cette requête sélectionne toutes les lignes de la table `etudiants` pour lesquelles l'attribut `nom` est égal à `'Dupont'` :

```
SELECT * FROM etudiants WHERE nom = 'Dupont';
```

Cette requête sélectionne toutes les lignes de la table `etudiants` pour lesquelles l'attribut `note` est supérieur ou égal à `15` :

```
SELECT * FROM etudiants WHERE note >= 15;
```

Dans le cas des valeurs nulles il faut utiliser la clause `IS NULL` :

```
SELECT * FROM etudiants WHERE note IS [NOT] NULL;
```

Cette requête sélectionne toutes les lignes de la table `etudiants` pour lesquelles l'attribut `note` prend ses valeurs dans celles renvoyées par la requête imbriquée :

```
SELECT * FROM etudiants WHERE note IN ( SELECT ... );
```

expression régulière Il existe aussi des opérateurs permettant de mettre en correspondance une chaîne avec un motif décrit sous la forme d'une expression régulière mais les expressions régulières utilisent des caractères spéciaux différents de ceux utilisés par `LIKE`. Contrairement aux motifs de `LIKE`, une expression régulière peut avoir une correspondance en toute place de la chaîne, sauf si l'expression régulière est explicitement ancrée au début ou à la fin de la chaîne.

Opérateurs de comparaison avec une expression régulière :

`~`	correspondance avec le motif, en tenant compte de la casse
`~*`	correspondance avec le motif, sans tenir compte de la casse
`!~`	non-correspondance avec le motif, en tenant compte de la casse
`!~*`	non-correspondance avec le motif, sans tenir compte de la casse

Description des motifs d'expresssions régulières :

`x`	correspondance avec le caractère x
	Classes de caractères :
`.`	correspondance avec tout caractère unique quelqu'il soit
`[...]`	correspondance avec un des caractères entre crochets
`[A-Z]`	correspondance avec une lettre majuscule
`[a-z]`	correspondance avec une lettre minuscule
`[0-9]`	correspondance avec un chiffre
	Quantificateurs :
`*`	une séquence de 0 ou plusieurs correspondances
`+`	une séquence de 1 ou plusieurs correspondances
`{m}`	une séquence d'exactement m correspondances
`{n,m}`	une séquence de n à m correspondances
	Ancrages :
`^`	correspondance de début de chaîne
`$`	correspondance de fin de chaîne

Quelques exemples d'expressions régulières :

`^a`	commence par a
`a*`	une suite de 0 ou plusieurs a
`.*`	une suite de 0 ou plusieurs caractères quelconques
`[abc]{6}`	6 caractères exactement parmi a, b ou c
`[A-Za-z0-9]+`	une suite alphanumérique d'au moins 1 caractère

Exemples de comparaisons :

```
SELECT 'abc' ~ 'abc';                    TRUE
SELECT 'abc' ~ '^a'                      TRUE
SELECT 'aaaabc' ~ '[abc]{6}';            TRUE
SELECT 'abccbbc' ~ '^a[bc]*';            TRUE
```

1.1.2 Opérateurs

Il existe différents opérateurs pour manipuler les données selon leur type. Voir le tableau 1.1 pour les opérateurs numériques, le tableau 1.2 pour les comparaisons, le tableau 1.3 pour les opérateurs booléens et le tableau 1.4 pour les opérateurs de chaînes. Les opérateurs peuvent s'appliquer à aux attributs ou constantes, du type correspondant.

TABLEAU 1.1 – Opérateurs et fonctions numériques disponibles dans PosgreSQL.

Opérateur	Description
+	addition
-	soustraction
*	multiplication
/	division (la division entière tronque les résultats)
%	modulo (reste)
^	exponentiel
|/	racine carrée
||/	racine cubique
!	factoriel
@	valeur absolue
ln(numerique)	logarithme
log(numerique)	logarithme base 10
round(numerique)	arrondi à l'entier le plus proche

Exemples d'utilisations :

```
SELECT ln(1);

SELECT note_cc + note_tp AS "Somme des notes" FROM etudiants;
```

TABLEAU 1.2 – Opérateurs de comparaison disponibles dans PosgreSQL.

Opérateur	Description
<	inférieur à
>	supérieur à
<=	inférieur ou égal à
>=	supérieur ou égal à
=	égal à
<> ou !=	différent de

Exemples d'utilisations :

```
SELECT 10 > 2 AS "Sup";

SELECT 10 = 2 AS "égalité";
```

TABLEAU 1.3 – Opérateurs booléens disponibles dans PosgreSQL.

Opérateur	Description
AND	ET logique
OR	OU logique
NOT	NON logique

Exemples d'utilisations :

```
SELECT * FROM etudiants WHERE moyenne > 10 AND notes >= 5;
```

TABLEAU 1.4 – Fonctions de chaînes disponibles dans PosgreSQL.

Fonction	Type renvoyé	Description	Exemple	Résultat
chaîne || chaîne	text	Concaténation de chaînes	'To' ||'to'	'Toto'
upper (chaîne)	text	Convertit une chaîne en majuscule	upper('toto')	'TOTO'
lower (chaîne)	text	Convertit une chaîne en minuscule	lower('TOTO')	'toto'
length (chaîne) ou char_length (chaîne)	int	Nombre de caractères de la chaîne	char_length('toto')	4
position(sous-chaîne in chaîne)	int	Emplacement de la sous-chaîne indiquée	position('ot' in 'toto')	2
substring(chaîne [from int] [for int])	text	Extrait une sous-chaîne	substring('toto' from 3 for 2)	'to'
regexp_split_to_ table(chaîne, motif)	lignes	Divise une chaîne selon une expression régulière	regexp_split_to_ table('Bonjour le monde', ' +')	'Bonjour' 'le' 'monde'

Exemples d'utilisations de fonctions de chaînes :

```
SELECT nom || ' ' || prenom AS "Nom complet" FROM etudiants;

SELECT upper(nom) AS "Nom majuscule" FROM etudiants;
```

1.1.3 Jointure

Une jointure relie deux tables entre elles. Il existe plusieurs types de jointures : les jointures croisées et les jointures qualifiées.

Jointure croisée

La jointure croisée entre deux tables renvoie le produit cartésien de ces deux tables. Ce n'est pas très utile en pratique.

```
SELECT * FROM table1 CROSS JOIN table2;
```

Exemple :

```
SELECT * FROM regions;                SELECT * FROM villes ;

 id_region | nom_region               id_ville | nom_ville | id_region
-----------+------------             ----------+-----------+-----------
         1 | Europe                          1 | Faro      |         1
         2 | Afrique                         2 | Tokyo     |
         3 | Amérique                        3 | Conakry   |         2

SELECT * FROM regions CROSS JOIN villes;

 id_region | nom_region | id_ville | nom_ville | id_region
-----------+------------+----------+-----------+-----------
         1 | Europe     |        1 | Faro      |         1
         1 | Europe     |        2 | Tokyo     |
         1 | Europe     |        3 | Conakry   |         2
         2 | Afrique    |        1 | Faro      |         1
         2 | Afrique    |        2 | Tokyo     |
         2 | Afrique    |        3 | Conakry   |         2
         3 | Amérique   |        1 | Faro      |         1
         3 | Amérique   |        2 | Tokyo     |
         3 | Amérique   |        3 | Conakry   |         2
```

Jointures qualifiées

Une jointure qualifiée entre deux tables renvoie le sous ensemble du produit cartésien de ces deux tables qui satisfait la condition de jointure. C'est avec une jointure qualifiée que l'on joint deux tables qui se font référence avec un couple Clef primaire / Clef étrangère. Il existe plusieurs sortes de jointures qualifiées :

— jointure interne,

— jointure externe gauche,

— jointure externe droite,

— jointure externe complète.

Jointure interne Pour chaque ligne de la table_1, le résultat contient une ligne pour chaque ligne de table_2 qui satisfait la condition de jointure.
Il y a plusieurs syntaxes possibles :

```
SELECT * FROM table_1 AS T1 JOIN table_2 AS T2
          ON T1.attribut_clef_P = T2.attribut_clef_E;

SELECT * FROM table_1 INNER JOIN table_2
          USING ( liste_attributs );

SELECT * FROM table_1 NATURAL JOIN table_2;
```

Ces syntaxes sont équivalentes lorsque les attributs Clef primaire et Clef étrangère qui se correspondent ont le même nom dans les deux tables. Les deux premières formes permettent de faire une jointure sur des attributs qui ont des noms différents. Attention : la jointure naturelle prends en compte tous les attributs qui ont le même nom dans les deux tables. Pour le natural join, tous les attributs identiques sont pris en compte dans la jointure.

Exemple :

```
SELECT * FROM regions;

 id_region | nom_region
-----------+------------
         1 | Europe
         2 | Afrique
         3 | Amérique

SELECT * FROM villes ;

 id_ville | nom_ville | id_region
----------+-----------+-----------
        1 | Faro      |         1
        2 | Tokyo     |
        3 | Conakry   |         2

SELECT * FROM regions JOIN villes ON regions.id_region = villes.id_region;

 id_region | nom_region | id_ville | nom_ville
-----------+------------+----------+-----------
         1 | Europe     |        1 | Faro
         2 | Afrique    |        3 | Conakry
```

Jointure externe Une jointure externe gauche est une jointure interne pour laquelle est ajouté au moins une ligne pour chaque ligne de la table de gauche. Ainsi pour chaque ligne de la table de gauche qui ne satisfait pas la condition de jointure, une ligne est ajoutée au résultat avec des valeurs NULL dans les colonnes de la table de droite. La jointure a donc au moins une ligne pour chaque ligne de la table de gauche. La jointure externe droite est l'inverse d'une jointure externe gauche. La jointure externe complète est la combinaison des deux. Syntaxe :

```
SELECT * FROM table_1 AS T1 LEFT JOIN table_2 AS T2
          ON T1.attribut_clef_P = T2.attribut_clef_E;

SELECT * FROM table_1 AS T1 RIGHT OUTER JOIN table_2 AS T2
          ON T1.attribut_clef_P = T2.attribut_clef_E;

SELECT * FROM T1 FULL OUTER JOIN T2 USING ( nom_attributs );
```

Exemple :

```
SELECT * FROM regions;                SELECT * FROM villes ;

 id_region | nom_region                id_ville | nom_ville | id_region
-----------+------------              ----------+-----------+-----------
         1 | Europe                          1 | Faro      |         1
         2 | Afrique                         2 | Tokyo     |
         3 | Amérique                        3 | Conakry   |         2

 SELECT * FROM regions LEFT JOIN villes USING (id_region);

 id_region | nom_region | id_ville | nom_ville
-----------+------------+----------+-----------
         1 | Europe     |        1 | Faro
         2 | Afrique    |        3 | Conakry
         3 | Amérique   |          |

 SELECT * FROM regions RIGHT JOIN villes USING (id_region);

 id_region | nom_region | id_ville | nom_ville
-----------+------------+----------+-----------
         1 | Europe     |        1 | Faro
           |            |        2 | Tokyo
         2 | Afrique    |        3 | Conakry

SELECT * FROM regions FULL JOIN villes USING (id_region);

 id_region | nom_region | id_ville | nom_ville
-----------+------------+----------+-----------
         1 | Europe     |        1 | Faro
           |            |        2 | Tokyo
         2 | Afrique    |        3 | Conakry
         3 | Amérique   |          |
```

1.1.4 Agrégats

Une fois que les lignes ont été sélectionnées il est possible de faire des regroupements en utilisant la clause `GROUP BY`, la clause `HAVING` permet de les filtrer :

```
SELECT nom_attributs FROM T1 GROUP BY nom_attributs
         HAVING condition;
```

```
SELECT nom FROM etudiants GROUP BY nom;
```

Il est possible d'appliquer une fonction d'agrégat à un ensemble de valeurs pour obtenir un résultat unique. Par exemple la somme des valeurs d'une colonne, la moyenne des valeurs d'une colonne, le min ou le max, cf tableau 1.5.

```
SELECT nom, avg(note) FROM notes_semestre GROUP BY nom
    HAVING avg(note) > 10;
```

Cette requête renvoie la moyenne des notes de chaque étudiant dont la moyenne est supérieure à 10.

TABLEAU 1.5 – Fonctions d'agrégat disponibles dans PosgreSQL.

Fonction	Type d'argument	Description
avg(expression)	numérique	la moyenne arithmétique de toutes les valeurs en entrée
count(*)	tout type	nombre de lignes en entrée
count(expression)	tout type	nombre de lignes en entrée pour lesquelles l'expression n'est pas NULL
max(expression)	tout type	valeur maximale de l'expression pour toutes les valeurs en entrée
min(expression)	tout type	valeur minimale de l'expression pour toutes les valeurs en entrée
sum(expression)	numérique	somme de l'expression pour toutes les valeurs en entrée

1.1.5 Tri du résultat

Une fois que les lignes ont été sélectionnées il est possible de les trier. La clause optionnelle `ORDER BY` permet de spécifier le ou les attributs sur lesquels les lignes seront triées ainsi que l'ordre du tri.

```
SELECT * FROM nom_table ORDER BY nom_attribut [ASC | DESC];
```

Exemples :

```
SELECT * FROM etudiants ORDER BY nom, notes_cc;

SELECT * FROM etudiants WHERE note_cc >= 10 ORDER BY nom DESC;
```

1.1.6 Limitation du nombre de lignes

`LIMIT` et `OFFSET` permettent de limiter le nombre de lignes renvoyées par la requête, `LIMIT` permet de spécifier le nombre de lignes et `OFFSET` permet de spécifier le nombre des premières lignes qui seront omises :

```
SELECT *
    FROM nom_table
    [ LIMIT nombre | ALL  ] [OFFSET nombre]
```

Renvoie les 10 premières lignes :

```
SELECT * FROM personnes LIMIT 10;
```

Renvoie à partir de la 11e lignes :

```
SELECT * FROM personnes OFFSET 10;
```

1.1.7 Combiner des requêtes

Il est possible de combiner deux requêtes avec les opérations d'ensemble : union, intersection et différence. Pour pouvoir être combinées, les deux requêtes doivent renvoyer le même nombre de colonnes et les colonnes correspondantes doivent avoir des types de données compatibles.

L'union ajoute le résultat de requete_2 au résultat de requete_1, les lignes dupliquées sont éliminées sauf si `UNION ALL` est utilisé :

```
requete_1 UNION [ALL] requete_2;
```

L'intersection renvoie toutes les lignes qui sont à la fois dans le résultat de requete_1 et dans le résultat de requete_2. Les lignes dupliquées sont éliminées sauf si on utilise `INTERSECT ALL` :

```
requete_1 INTERSECT [ALL] requete_2;
```

La différence renvoie toutes les lignes qui sont dans le résultat de requete_1 mais pas dans le résultat de requete_2. Les lignes dupliquées sont éliminées sauf si `EXCEPT ALL` est utilisé :

```
requete_1 EXCEPT [ALL] requete_2;
```

Ces différentes opérations peuvent êtres utilisées conjointement.

1.2 Insertion de données

Pour entrer des données dans une table on utilise la commande `INSERT INTO` :

```
INSERT INTO nom_table
   VALUES ( valeur_1, ... valeur_n );
```

Le nombre de valeurs de la clause `VALUES` et l'ordre des valeurs doit correspondre aux attributs de la table. Sinon il faut préciser la liste des attributs pour lesquels des données sont fournies :

```
INSERT INTO nom_table ( attribut_1, ... attribut_n )
   VALUES ( valeur_1, ... valeur_n );
```

Une table peut être remplie à partir du résultat d'une requête :

```
INSERT INTO nom_table SELECT DISTINCT nom_attributs
    FROM nom_table_2;
```

PostreSQL fournit également une commande `COPY` qui permet d'insérer une grande quantité de données à partir d'un fichier texte. Cette commande ne peut fonctionner que si le serveur peut lire le fichier qui contient les données. Ce dernier doit se trouver sur le même ordinateur dans un répertoire lisible par le serveur PostreSQL. Les données lues dans le fichiers sont directement intégrées dans la table passée en argument à la commande `COPY`. Les données doivent êtres stockées sous forme de colonnes qui doivent être dans le même ordre que les attributs de la table dans laquelle elles sont importées. Dans l'exemple suivant la commande s'exécute sur un ordinateur sous windows :

```
COPY nom_table FROM 'C:/Documents and Settings/User/import.txt';
```

Ici la commande s'exécute sur un ordinateur sous linux, elle importe des données stockées au format CSV avec un séparateur de type tabulation :

```
COPY nomtable
    FROM '/Home/gallut/import.csv'
    WITH
      DELIMITER E'\t'
      CSV HEADER QUOTE AS '"' ;
```

Dans psql on peut utiliser la commande `\copy` qui permet de lire un fichier qui se trouve sur le même ordinateur que le client :

```
\copy importation FROM 'polymorphisme.csv' DELIMITER E'\t'
CSV HEADER QUOTE AS '"'
```

1.3 Mise à jour de données

Il est possible de mettre à jour une ligne spécifique, toutes les lignes ou un sous-ensemble de lignes de la table. Chaque colonne peut être actualisée séparément.

```
UPDATE nom_table
    SET nom_attribut = valeur
    WHERE condition;
```

Exemple :

```
UPDATE etudiants
    SET nom = 'Dupont'
    WHERE nom = 'Dupond';
```

1.4 Suppression de données

La suppression touche une ou plusieurs lignes qui remplissent une condition :

```
DELETE FROM nom_table
    WHERE condition;
```

Exemple :

```
DELETE FROM etudiants
    WHERE nom = 'De Mesmaeker';
```

2 Langage de définition de données

C'est un ensemble de commandes qui permettent de créer les éléments de la base de données.

2.1 Création d'une base

Il existe différentes méthodes pour créer une nouvelle base qui dépendent du client utilisé. En SQL la commande est la suivante :

```
CREATE DATABASE nom_base
  WITH OWNER = proprietaire
       ENCODING = 'encodage';
```

Exemple :

```
CREATE DATABASE polymorphisme
  WITH OWNER = gallut
       ENCODING = 'UTF8';
```

Cette commande crée une base « `polymorphisme` » dont le propriétaire est « `gallut` » et dans laquelle les données seront encodées en « `UTF8` ». PostgreSQL peut gérer de nombreux encodages différents, pour le français les encodages `iso latin1` ou `UTF8` permettent de prendre en compte les accents. Attention : l'encodage par défaut dépend du serveur, généralement il s'agit de `SQL_ASCII` qui ne permet pas l'utilisation de caractères accentués.

2.2 Création d'une table

Pour créer une table, on utilise la commande `CREATE TABLE`. Il faut indiquer, au minimum, le nom de la table, les noms des colonnes et le type de données de chacune d'elles.

```
CREATE TABLE nom_table (
    nom_attribut_1 type,
    ...
    nom_attribut_n type
);
```

Lors de la création d'un nouvel enregistrement, si aucune valeur n'est fournie pour un attribut, cet attribut prend la valeur `NULL` par défaut. Une valeur par défaut peut être définie à la création de la table, c'est cette valeur qui sera attribuée si aucune valeur n'est fournie lors de la création d'un nouvel enregistrement.

Par ailleurs, il est possible d'interdire la valeur `NULL` pour un attribut et de rendre unique chaque valeur d'un attribut.

Exemple :

```
CREATE TABLE compte_bancaire (
    nom varchar(25) NOT NULL,
    numero text UNIQUE,
    solde int DEFAULT 0
);
```

2.3 Création d'une contrainte

Une contrainte peut s'exercer sur un attribut ou sur une table. Une contrainte sur une table peut porter sur un ou plusieurs attribut contrairement à une contrainte sur un attribut qui ne porte que sur ce dernier.

2.3.1 Clef primaire

Pour ajouter une clef primaire à une table, plusieurs syntaxes sont possibles :

```
CREATE TABLE nom_table (
    nom_attribut_1 int PRIMARY KEY,
    nom_attribut_2 varchar(10),
    nom_attribut_n text
);
```

Dans ce cas il s'agit d'une contrainte d'attribut. Cette syntaxe ne permet pas de définir une clef portant sur plusieurs attributs. D'autre part le nom de la contrainte n'est pas précisé. La syntaxe suivante est préférable :

```
CREATE TABLE nom_table (
    nom_attribut_1 int,
    nom_attribut_2 varchar(10),
    nom_attribut_n text,
    CONSTRAINT nom_contrainte PRIMARY KEY (liste_attributs)
);
```

La contrainte de clef primaire `nom_contrainte` est un objet de la base. Il est d'usage d'utiliser le nom de la table suivi de `pkey`, par exemple : `pays_pkey` pour la clef primaire de la table `pays`. La liste des attributs représentes la ou les colonnes qui constituent la clef primaire.

Exemple :

```
CREATE TABLE pays (
    nom_pays varchar(30),
    langue   varchar(2),
    population integer,
    CONSTRAINT pays_pkey PRIMARY KEY (nom_pays)
);
```

2.3.2 Clef étrangère

Pour ajouter une clef étrangère à une table lors de la création de la table :

```
CREATE TABLE nom_table (
    nom_attribut_1 int,
    nom_attribut_2 varchar(10),
    nom_attribut_n text,
    CONSTRAINT nom_contrainte
      FOREIGN KEY (liste_attributs_enfants)
      REFERENCES nom_table_parent (liste_attributs_parents)
      MATCH FULL
);
```

Il est d'usage d'utiliser le nom de la table ou des tables suivi de `fkey`. Le nombre et le type des attributs des deux listes d'attributs qui se font référence doivent correspondre. `MATCH FULL` permet de préciser comment se fait la correspondance entre les colonnes, ici elle doit être complète.

Ces contraintes peuvent être créées soit comme option d'une colonne soit comme contrainte en tant que tel :

```
CREATE TABLE pays (
    nom_pays varchar(30) PRIMARY KEY,
    langue   varchar(2),
    population integer
);

CREATE TABLE villes (
    ville_id  serial PRIMARY KEY,
    nom_ville varchar(50),
    nom_pays  varchar(30)
       REFERENCES pays
);
```

```
CREATE TABLE pays (
    nom_pays varchar(30),
    langue   varchar(2),
    population integer,
    CONSTRAINT pays_pkey
      PRIMARY KEY (nom_pays)
);

CREATE TABLE villes (
    ville_id  serial,
    nom_ville varchar(50),
    nom_pays  varchar(30),
    CONSTRAINT villes_pkey
      PRIMARY KEY (ville_id),
    CONSTRAINT villes_pays_fkey
      FOREIGN KEY (nom_pays)
      REFERENCES pays (nom_pays)
      MATCH FULL
);
```

2.3.3 Contrainte de vérification

Une contrainte de vérification permet de s'assurer que les valeurs d'un attribut sont bien comprises dans son domaine de valeurs :

```
CREATE TABLE etudiants (
    etudiant_id serial PRIMARY KEY,
    note integer CONSTRAINT note_valide CHECK (note >= 0 AND note <= 20),
    nom varchar(30) CONSTRAINT maj CHECK (nom ~ E'^[A-Z]+$'),
    prenom varchar(30) CONSTRAINT minu CHECK (prenom ~ E'^[A-Z][a-z]+$')
);
```

Dans cette exemple, les valeurs du champs `note` doivent être comprises entre 0 et 20, les valeurs du champs `nom` doivent écrites en majuscules et celles du champ `prenom` doivent commencer par une majuscule suivie de minuscules, ces deux contraintes sont exprimées au moyen d'une expression régulière, voir page 5.

2.4 Modification d'une table

Pour modifier une table on utilise la commande `ALTER TABLE`. Soit comme précédemment pour ajouter une contrainte, soit pour ajouter ou supprimer un attribut. Il est possible de modifier tous les éléments d'une table. Pour cela on utilise

Ajout d'une clef primaire :

```
ALTER TABLE nom_table
    ADD CONSTRAINT nom_contrainte PRIMARY KEY (liste_attributs);
```

Suppression d'une contrainte :

```
ALTER TABLE nom_table
   DROP CONSTRAINT nom_contrainte;
```

Ajout d'un attribut :

```
ALTER TABLE nom_table
   ADD COLUMN nom_attribut type;
```

Toutes les options possibles à la création sont également possibles ici.

Suppression d'un attribut :

```
ALTER TABLE nom_table DROP COLUMN nom_attribut;
```

Renommer une table :

```
ALTER TABLE nom_table RENAME TO nouveau_nom_table;
```

Renommer un attribut :

```
ALTER TABLE nom_table RENAME COLUMN
        nom_attribut TO nouveau_nom_attribut;
```

2.5 Suppression d'un élément de la base

Pour supprimer un élément on utilise la commande `DROP` suivis de la nature de l'élément puis de son nom. Pour supprimer une table :

```
DROP TABLE nom_table;
```

Pour supprimer la table `"villes"` on utilise la commande :

```
DROP TABLE villes;
```

Pour supprimer la base `"geographie"` :

```
DROP DATABASE geographie;
```

2.6 Gestion des dépendances

Une table avec une contrainte de clef étrangère dépend de la table à laquelle elle fait référence. Il est nécessaire de préciser le comportement que doit suivre le SGBD en cas de modification ou suppression de la table parent. L'option `CASCADE` permet de lever la contrainte de clef étrangère. Pour la modification et la suppression de données dans la table parent les options suivantes permettent de définir le comportement :

```
CREATE TABLE nom_table (
        nom_attribut_1 int,
        nom_attribut_2 varchar(10),
        nom_attribut_n text,
        CONSTRAINT nom_contrainte
          FOREIGN KEY (liste_attributs_enfants)
          REFERENCES nom_table_parent (liste_attributs_parents)
          MATCH FULL
          ON UPDATE CASCADE ON DELETE CASCADE
);
```

L'option `'ON DELETE CASCADE'` entraîne la suppression des lignes de la table fille qui font référence aux lignes supprimées dans la table parent.

L'option `'ON UPDATE CASCADE'` entraîne la modification de l'attribut clef étrangère de la table fille lorsque l'attribut référencé de la table parent est modifié.

```
CREATE TABLE pays (
  nom_pays varchar(30) PRIMARY KEY,
  langue   varchar(2),
  population integer
);
```

```
CREATE TABLE villes (
  ville_id  serial PRIMARY KEY,
  nom_ville varchar(50),
  nom_pays  varchar(30),
  CONSTRAINT villes_fkey
    FOREIGN KEY (nom_pays)
    REFERENCES pays (nom_pays)
    MATCH FULL
    ON UPDATE CASCADE
    ON DELETE CASCADE
);
```

3 Langage de contrôle de données

Le SQL permet de définir finement les privilèges des utilisateurs sur chaque objet de la base. Ainsi, un utilisateur peut avoir le droit de lire dans une table particulière mais pas d'y écrire alors qu'il peut lire et écrire dans une autre. Sur une table on peut accorder les privilèges de sélection, d'insertion, de mise à jour, de suppression, de référencement et de création de déclencheur (trigger). Sur une base on peut accorder les privilèges de création et de connexion. Il est également possible d'accorder des privilèges sur les autres éléments tels que les schémas, les fonctions etc..

Les privilèges sont accordés aux utilisateurs ou aux groupes d'utilisateurs, tout privilège accordé à un groupe est automatiquement accordé à tous les utilisateurs de ce groupes ainsi qu'aux futurs utilisateurs de ce groupe. Dans PostgreSQL les utilisateurs et les groupes ont exactement la même nature, accorder un privilège à un utilisateur ou à un groupe est strictement équivalent. Ce sont des rôles (`role`). Le rôle `PUBLIC` contient tous les rôles, accorder un privilège à `PUBLIC` revient à rendre l'objet publique.

Chaque objet d'une base appartient à l'utilisateur qui l'a créé, c'est le propriétaire. Le propriétaire possède tous les privilèges sur l'objet, dont le privilège d'accorder des privilèges aux autres utilisateurs sur cet objet. Par défaut les autres utilisateurs n'ont aucun privilège sur un objet nouvellement créé.

Pour accorder des privilèges on utilise la commande `GRANT` et pour révoquer des privilèges on utilise `REVOKE` :

```
GRANT SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER | ALL
    ON [ TABLE ] nom_table
    TO nomrole | PUBLIC ;

REVOKE SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER | ALL
    ON [ TABLE ] nom_table
    FROM nom_role | PUBLIC;
```

Par exemple, pour accorder à `PUBLIC` tous les privilèges sur la table `regions` :

```
GRANT ALL ON TABLE regions TO PUBLIC;
```

Pour accorder à l'utilisateur `webuser` uniquement le privilège de sélection sur la table `regions` :

```
GRANT SELECT ON regions TO webuser;
```

Pour révoquer tous les privilèges de `webuser` sur `regions` :

```
REVOKE ALL ON regions FROM webuser;
```

Le standard SQL n'autorise pas l'initialisation des privilèges sur plus d'un objet par commande. Ainsi il faut accorder ou révoquer les privilèges objet par objet. Par contre pour accorder ou révoquer les mêmes privilèges à un ensemble d'utilisateurs, il suffit des les placer dans un même groupe et d'accorder ou de révoquer les privilèges pour ce groupe.

4 Langage de contrôle de transactions

Lorsque deux utilisateurs, ou plus, essaient d'accéder en même temps aux mêmes données, il existe un risque de conflit. Si un utilisateur modifie des données pendant qu'un autre est en train de les lire ce derniers risque d'avoir des erreurs. Par exemple, si le premier utilisateur modifie des clefs pendant que l'autre utilisateur réalise une jointure sur ces clefs, la qualité des résultats obtenus par ce dernier est compromise. Il n'y pas de retour en arrière possible lorsque l'on fait des modifications. Cela représente un risque que l'intégrité référentiel soit violée sans retour possible.

Pour éviter ce genre de problèmes, les SGBDR isole les sessions des utilisateurs à l'intérieur de transactions. Ce qui se passe à l'intérieur d'une transaction est invisible pour les autres transactions. Cela revient à créer une copie privée de la base pour chaque session. Ainsi les modifications réalisées par un utilisateur ne seront visibles que par lui jusqu'à ce que la transaction soit validée. Lorsqu'une transaction est validée, les données sont reportées dans la base. Par défaut chaque instruction est comprise dans sa propre transaction. Il peut être nécessaire de regrouper plusieurs instructions à l'intérieur d'une même transaction. En regroupant plusieurs instructions ensembles les modifications seront appliquées à la base uniquement si toutes les instructions ont réussi. Dans le cas contraire il suffit de revenir au début de la transaction pour retrouver l'état initial.

La commande `START TRANSACTION;` ou `BEGIN;` dans PostgreSQL, permet de commencer une transaction qui se finira soit par la commande `COMMIT;` qui permet d'appliquer à la base le résultat des instructions de la transaction, soit par la commande `ROLL BACK` qui permet de revenir à l'état initial.

5 Quelques commandes psql

```
unixpromtp> psql -? (liste des commandes de démarrage)
unixprompt> psql -U utilisateur [-h serveur] [-p port] [base]
unixpromtp> psql -U obi5_07 polymorphisme
Password: ************
```

`dbname=> \q`	– quitter
`dbname=> \c base`	– pour se connecter à une base différente
`dbname=> \l`	– liste des bases
`dbname=> \d`	– liste des tables de la base
`dbname=> \d table`	– description des attributs de la table
`dbname=> \?`	– liste des commandes psql
`dbname=> \h`	– liste des commandes SQL
`dbname=> \h commande`	– syntaxe d'une commande SQL
`dbname=> \i fichier`	– exécuter un script de commandes SQL
`dbname=> \e fichier`	– éditer un fichier de commandes existant et l'exécuter
`dbname=> \e`	– éditer la dernière commande et l'exécuter
`dbname=> \w fichier`	– écrire la dernière commande SQL dans un fichier
`dbname=> \o fichier`	– écrire le résultat des commandes SQL dans un fichier
`dbname=> \o`	– afficher le résultat des commandes SQL