Bases de données

TP SQL - Gestion d'un zoo

Introduction

L'objectif de ce TP est de concevoir puis utiliser une base de données destinée à la gestion de l'alimentation des animaux d'un zoo. Le nom donné à cette base sera ZOO. Nous utiliserons le SGBD (Système de Gestion de Bases de Données) PostgreSQL et son interface d'administration PgAdmin.

1 Modélisation de la base de données

La direction du zoo veut informatiser la gestion de l'alimentation des animaux. Nous profiterons de cette informatisation pour mémoriser les informations concernant les animaux.

A leur arrivée dans le zoo, les animaux doivent être enregistrés dans le système. Lorsqu'ils ont été identifiés, ils vivent dans des enclos pour lesquels nous ne possédons pas d'information précise sur la localisation ou l'agencement, mais uniquement des informations de type à l'entrée du parc, dans l'allée principale, etc. pour la localisation et aquarium, cage fermée, prairie, etc. pour l'agencement.

Sur chacun des enclos du zoo, sont apposées des fiches d'informations concernant les animaux qui y vivent (nom de l'animal, sexe, date de naissance, date d'arrivée au zoo, nom de l'espèce, nom scientifique, population estimée de l'espèce, zones géographiques d'habitat dans la nature). Pour chacune des zones d'habitat naturel (pays, description, etc.) le gestionnaire du zoo aimerait connaître approximativement la population locale.

Un enclos peut contenir plusieurs individus d'une même espèce, mais parfois aussi plusieurs animaux d'espèces différentes.

Chaque espèce a des besoins alimentaires. Pour un animal appartenant à une espèce, l'employé responsable de l'enclos doit amener quotidiennement une certaine quantité de nourriture. Par exemple, l'éléphant d'Afrique demande 80 kg de foin, 10 kg d'avoine et 5 kg de carottes par jour. Toutes les quantités sont toujours indiquées en kg.

Le zoo possède un catalogue d'aliments qui permet de gérer le stock disponible. Pour chaque aliment, le catalogue indique aussi les aliments de substitution qui seront utilisés en cas de rupture de stock. Pour chaque aliment de substitution, il y a un taux de remplacement. Par exemple, un kg de foin peut être remplacé par 0.9 kg de luzerne ; donc, si le stock de foin était épuisé, l'éléphant pourrait recevoir 0.9 * 80 kg de luzerne. Pour chaque aliment, il peut y avoir plusieurs aliments de substitution et il doit y en avoir

au moins un.

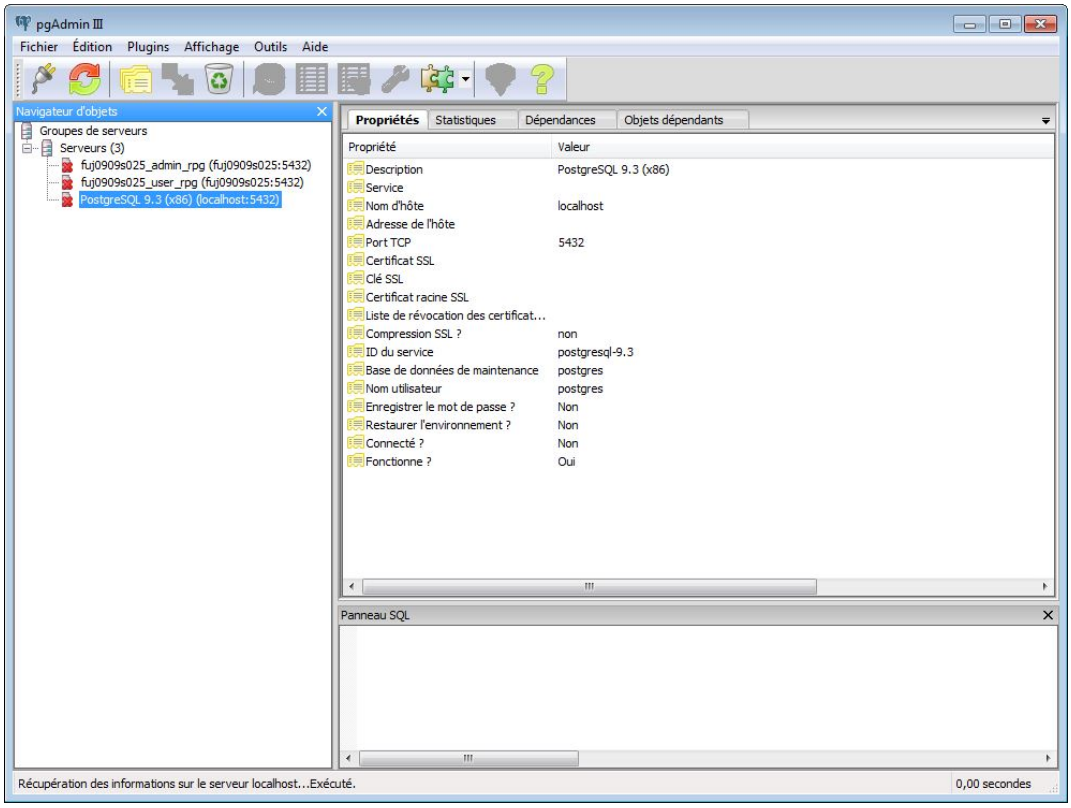
Les informations concernant un animal sont mémorisées lorsque l'animal fait son entrée dans le zoo. Il est évident que certaines informations ne sont pas effacées de la base de données même si elles ne sont pas utiles à un moment donné. Par exemple, on n'efface pas les informations concernant une espèce même si le zoo ne possède plus d'animal de cette espèce ; on ne supprime pas un enclos même s'il est vide.

⇒ En utilisant le formalisme UML, réalisez le modèle relationnel de la base de données à mettre en place.

2 Création de la base de données

Nous utiliserons PgAdmin pour manipuler notre base de données. PgAdmin est un outil d'administration graphique pour PostgreSQL.

Son interface se présente de la manière suivante :

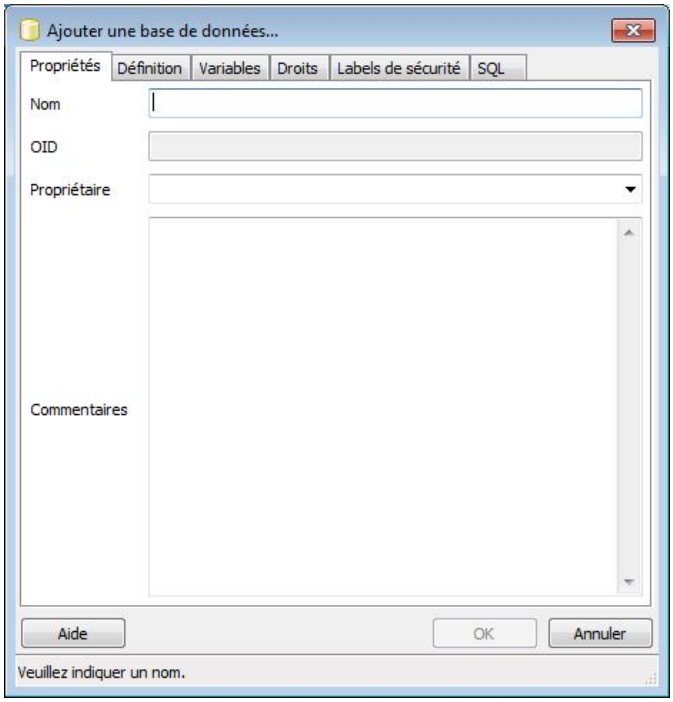


Sur la gauche, dans la partie *Navigateur d'objets*, on retrouve la liste des serveurs avec lesquels une connexion a été enregistrée. Ils sont barrés d'une petite croix rouge si aucune connexion n'a encore été établie dans la session en cours.

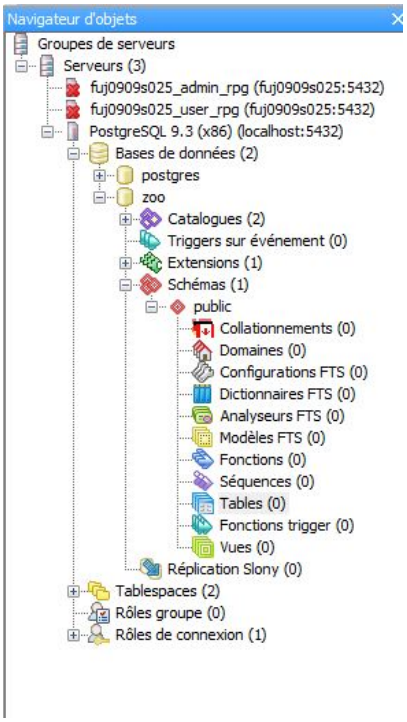
⇒ Connectez vous au serveur *localhost* en double cliquant dessus puis dépliez l'item *Bases de données*.

Cela permet d'afficher la liste des bases de données stockées sur le serveur *localhost* (ie. le poste sur lequel vous travaillez).

⇒ Ajoutez une nouvelle base de données à l'aide d'un clic droit sur *Bases de données* > *Ajouter une base de données...*. Nommez la **ZOO**.



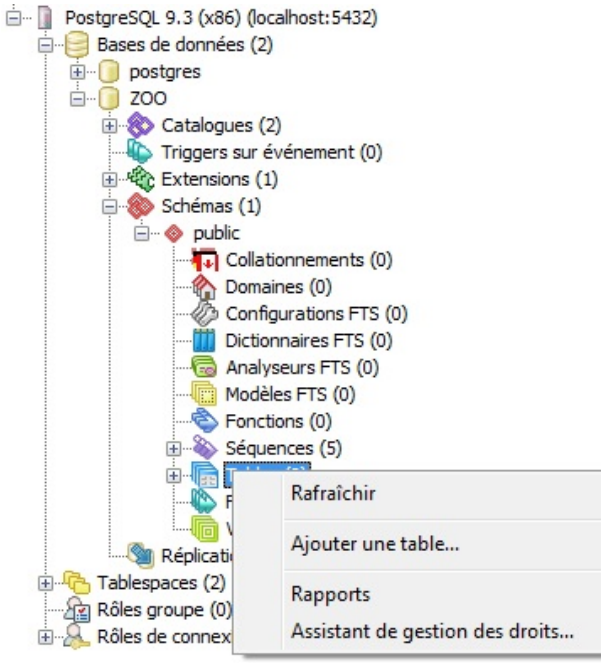
En dépliant la base de données créée, on obtient :



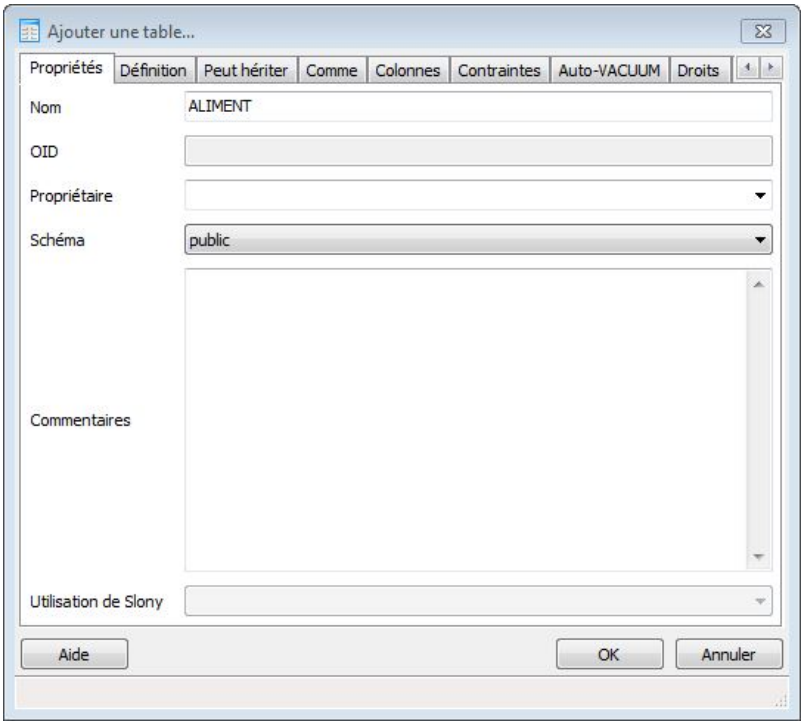
3 Création des tables

3.1 Table ALIMENT

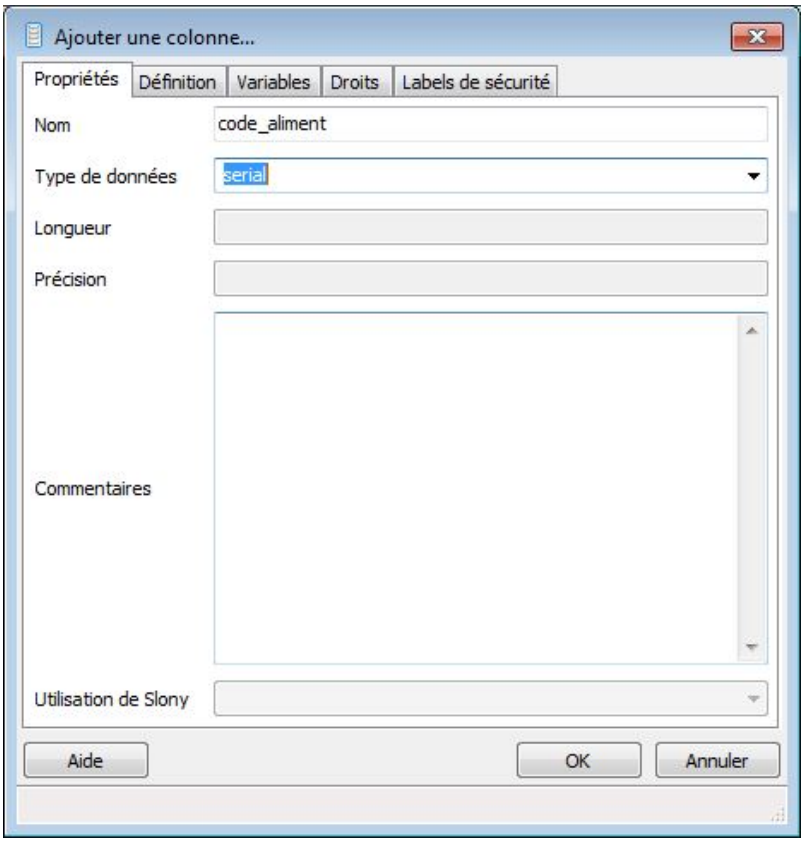
⇒ Dépliez le schéma *public* et, par clic droit sur *Tables*, ajoutez une nouvelle table.



⇒ Nommez votre table **ALIMENT**



⇒ Dans l'onglet *Colonnes*, ajoutez les colonnes **code_aliment** de type serial, **nom_aliment** de type Text et **stock** de type entier.



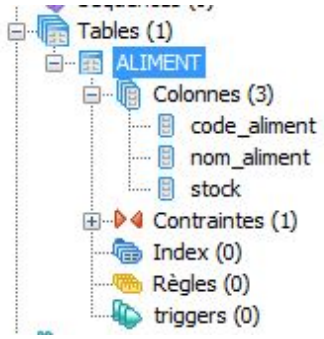
Le type serial est un type numérique spécifique à PostgreSQL. Il permet de définir un entier qui s'auto-incrémente lorsque des enregistrement sont ajoutés à la table. Il est particulièrement adapté pour des identifiants uniques de table.

Les types de données couramment utilisés dans Postgres sont :
- numériques : smallint, integer, bigint (entiers plus ou moins longs), decimal, numeric, real, double precision (nombres à virgules plus ou moins précis), serial (entier auto-incrémenté) ;
- monétaire ;
- caractères : character, char, text (qui se différencient par le nombre de caractères autorisés) ;
- binaire ;
- date/heure : date, heure, date-heure, intervalle ;
- booleen (valeurs possibles : TRUE, t, true, y, yes, on, 1 / FALSE, f, false, n, no, off, 0) ;
- géométriques : point, line, lseg, box, path, polygon, circle ;
- adresse réseau : IPv4, IPv6 et MAC ;
- types composites (listes de types simples, peu recommandées).

⇒ Enfin, dans l'onglet *Contrainte*, ajoutez une clé primaire *code_ aliment_ cle_ primaire*. Faite portez cette clé primaire sur le champs **code_aliment** (sélectionnez la colonne adéquat dans l'onglet *Colonnes*).

Une clé primaire permet de distinguer de façon unique les éléments d'une table. La clé primaire peut être composée d'une ou de plusieurs colonnes.

⇒ Le paramétrage de la table est maintenant terminé : validez sa création.



⇒ Retournez dans l'explorateur pour sélectionner la table.

Cela fait apparaître des instructions dans le *Panneau SQL* de PgAdmin :

```
Panneau SQL
-- Table: "ALIMENT"

-- DROP TABLE "ALIMENT";

CREATE TABLE "ALIMENT"
(
  code_aliment serial NOT NULL,
  nom_aliment character(50),
  stock integer,
  CONSTRAINT code_aliment_cle_primaire PRIMARY KEY (code_aliment)
)
WITH (
  OIDS=FALSE
);
ALTER TABLE "ALIMENT"
  OWNER TO postgres;
```

Il s'agit des instructions SQL de création de la table **ALIMENT**.

Le SQL (Structured Query Language) est un langage informatique normalisé servant à interagir avec des bases de données relationnelles. C'est le langage pour base de données le plus répandu, même si l'on peut observer quelques petites différences d'implémentation selon les SGBDR.

⇒ A partir de cet exemple d'instruction SQL, déduire la syntaxe générique de création d'une table en SQL.

Réponse .....
.....

3.2 Table ENCLOS

⇒ Dans la barre d'outils de PgAdmin, cliquez sur le bouton d'ouverture du constructeur de requêtes SQL.

Le constructeur de requêtes permet d'écrire des instructions en SQL pour les envoyer à la base de données.

⇒ A l'aide d'une requête SQL créez la table **Famille**. Elle comportera deux colonnes : **code_enclos** de type serial et **situation** de type text, et une clé primaire.

3.3 Table ESPECE

⇒ De la même manière, créez la table Enclos.

Elle comportera 4 colonnes : **code_espece** de type serial, **nom** de type varchar(50), **nom_scientifique** de type varchar(50) et **population** de type integer. Elle contriendra également une clé primaire sur la colonne code_espece.

3.4 Table ANIMAL

⇒ De la même manière, créez la table ANIMAL. Elle comportera 6 colonnes :
- **code_animal** de type serial
- **nom_animal** de type varchar(50)
- **sexe** de type char
- **date_naissance** de type date
- **date_arrivee** de type date
- **remarques** de type texte

Elle comportera également une clé primaire sur la colonne code_animal.

Par ailleurs, pour représenter la relation entre les tables ANIMAL et ENCLOS d'une part et ANIMAL et ESPECE d'autre part, nous ajouterons deux colonnes dans la table :
- **code_espece** de type integer
- **code_enclos** de type integer

Ces deux colonnes pointerons vers les colonnes de même nom des tables ESPECE et ENCLOS. Pour s'assurer que les identifiants saisis dans ces colonnes soient bien présent dans les tables d'origine, nous ajouterons des clés étrangères.

Les clés étrangères permettent de gérer les relations entre plusieurs tables. Elles garantissent la cohérence des données.

3.5 Autres tables du modèle

Complétez la base avec les tables pas encore implémentées du modèle relationnel : ZONE_GEOGRAPHIQUE, ALI_SUBSTITUTION, ESP_ANI et ESP_ZONEGEO.

3.6 Modification de la structure de la base

La base que nous avons implémenté jusqu'à présent ne tient pas compte de toute les contraintes prévues dans le modèle conceptuel.

On peut modifier une table en utilisant une instruction `ALTER TABLE ...`. Par exemple pour ajouter une condition `NOT NULL` sur la colonne stock de la table ALIMENT, on exécutera l'instruction SQL :

```
ALTER TABLE "ALIMENT" ALTER COLUMN "stock" SET NON NULL
```

⇒ Recopiez et exécutez cette instruction.

⇒ De manière similaire, écrivez et exécutez les instructions permettant de ajouter les conditions suivantes dans la base :
- Table ANIMAL, code_espece non nul
- Table ESPECE, nom_scientifique non nul
- Table ESP_ALI, quantite non nulle
- Table ZONE_GEOGRAPHIQUE, pays non nul

On souhaite également ajouter une contrainte sur la table ANIMAL pour s'assurer que la date de naissance est postérieure à la date d'arrivée. La syntaxe est :

```
ALTER TABLE "ANIMAL" ADD CONSTRAINT nom_contrainte contrainte
```

⇒ Ajoutez une contrainte date_naissance > date_arrivee

Le libellé initialement choisi pour la colonne quantite de la table des aliments de substitution apparait finalement mal choisi et on souhaite le modifier. Pour renommez une colonne, la structure de l'instruction SQL sera de type :

```
ALTER TABLE nom_table RENAME COLUMN colonne TO nouvelle_colonne
```

⇒ Dans la table ALI_SUBSTITUTION, renommez la colonne quantite en taux et ajoutez une contrainte non nul sur cette colonne.

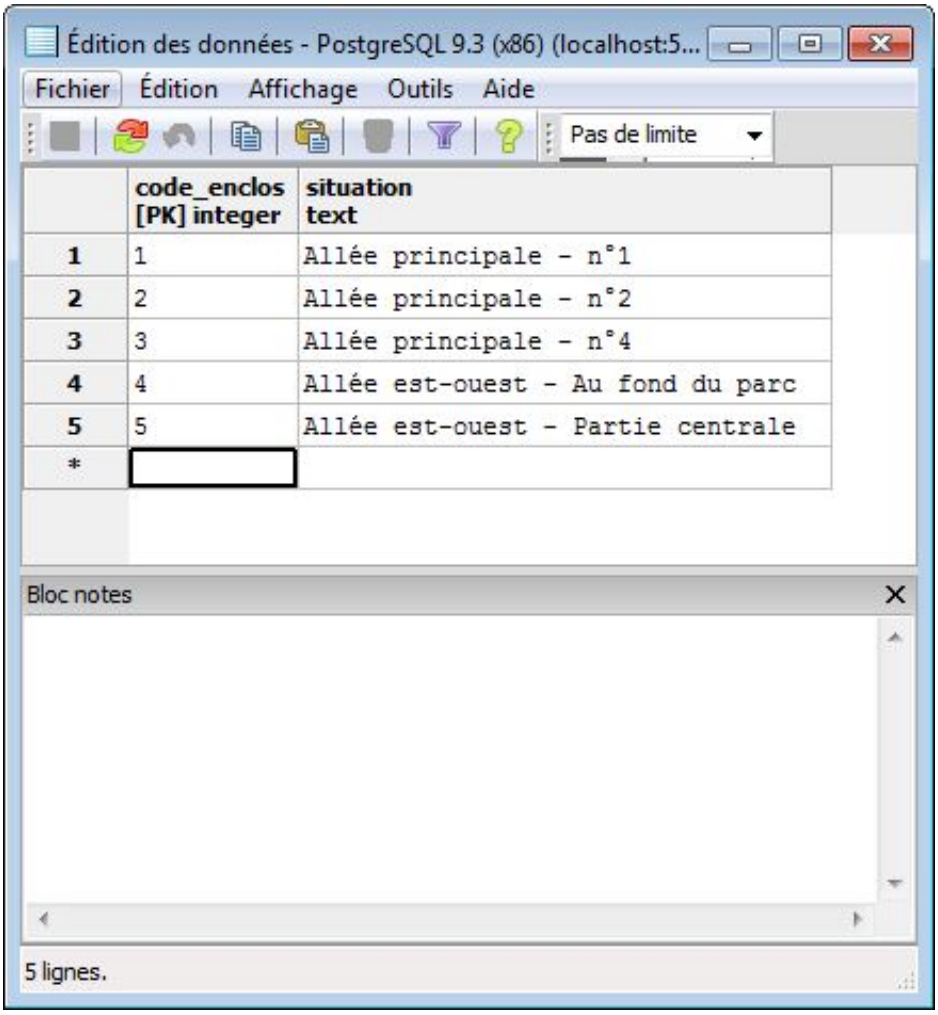
4 Insertion des données

4.1 Dans PgAdmin

Maintenant que la structure de notre base est en place, il est temps de commencer à y insérer des enregistrements. Dans PgAdmin, cette opération peut s'effectuer en ouvrant la fenêtre d'édition des données : sélectionner la table dans le navigateur d'objets et cliquer sur le bouton d'ouverture de la fenêtre.

L'interface est alors très similaire à un tableau excel.

⇒ Ajoutez 5 lignes à la table ENCLOS en respectant les éléments de la capture d'écran ci-dessous :



4.2 A l'aide d'une requête SQL

L'insertion d'enregistrement est bien évidemment également accessible en SQL. La syntaxe est :

```
INSERT INTO table ([colonne1, colonne2, ...]) VALUES (expression1, expression2
    , ...);
```

⇒ Recopiez la ligne suivante pour insérer une ligne dans la table ALIMENT :

```
INSERT INTO "ALIMENT" ("code_aliment", "nom", "stock") VALUES (1, "Ble", 100);
```

⇒ Insérez ensuite les lignes suivantes dans la table ALIMENT :

code_aliment	nom	stock
2	Foin	75
3	Herbe	450
4	Poisson	47
5	Bananes	32
6	Bambou	79
7	Insectes	13
8	Viande	27
9	Carcasse animale	50
10	Baies	4
11	Carottes	30

4.3 A l'aide d'un fichier .sql

L'insertion manuelle de l'ensemble des données de la base serait assez fastidieuse. Heureusement, PostgreSQL nous met à disposition des outils pour exécuter sur un serveur un ensemble d'instructions SQL contenues dans un fichier. Nous utiliserons cette possibilité pour charger une base ZOO complète.

⇒ Dans PgAdmin renommez votre base ZOO en ZOO2.

⇒ Récuperez le fichier ZOO.sql. Il contient l'ensemble des instruction SQL pour créer la base, les tables, contraintes... et de peupler la base.

PostgreSQL est livré avec un ensemble d'utilitaires (on parle aussi d'applications client de PostgreSQL) qui permettent de manipuler une base de données sur n'import quel serveur sans avoir à ouvrir d'instance de PgAdmin ou à écrire d'instruction SQL. L'autre avantage des ces utilitaires est qu'ils sont identiques sur l'ensemble des plateformes où un client PostgreSQL est installé.

Nous utiliserons ces utilitaires pour créer une nouvelle base de données vierge puis la remplir à l'aide des instructions du fichier ZOO.sql.

⇒ Ouvrez un terminal windows et tapez la commande `cd "C:\Program Files (x86)\PostgreSQL\9.3\bin"` pour vous placez dans le répertoire contenant les utilitaires de PostgreSQL 9.3.

⇒ Saisissez ensuite : `createdb.exe -h localhost -p 5432 -U postgres ZOO` pour créer une nouvelle base vierge ZOO.

Lors de l'appel à l'outil, on a précisé quelques options :
- h (host) = localhost : on se connecte au serveur local
- p (port) = 5432 : port par défaut utilisé par postgres
- U (user) = postgres : nom de l'utilisateur avec lequel on se connecte à la base.

Le paramétrage de options se fait de la même manière pour l'ensemble des utilitaires PostgreSQL.

⇒ Saisissez ensuite la ligne `psql.exe -h localhost -p 5432 -U postgres -f D:\G1_Bases_de_donnees\ZOO.sql` en remplaçant le chemin par le chemin du fichier ZOO.sql sur votre poste.

Elle permet d'appeler l'utilitaire psql qui est utilisé pour envoyer des requêtes SQL à un serveur PostgreSQL. L'outil psql accepte en entrée une instruction SQL simple ou un fichier .sql contenant un ensemble d'instruction.

Retournez dans PgAdmin pour vérifier que la base a correctement été créée et remplie.

5 Sélection de données

5.1 Requêtes SELECT simples

La conception de la base étant maintenant achevée, il nous est possible de l'exploiter.

Le mot clé `SELECT` permet la sélection et l'affichage de données. Testez par exemple :

```
SELECT 'Hello world !';
SELECT 2+3;
```

On souhaite d'abord connaître l'ensemble des animaux du zoo.

⇒ Recopiez la requête suivante et observez le résultat :

```
SELECT * FROM "ANIMAL";
```

⇒ Sélectionnez de la même manière l'ensemble des espèces enregistrées dans la base.

Il peut être intéressant de ne pas afficher toutes les colonnes dans le résultat de la requête. Dans ce cas, on nomme explicitement les colonnes que l'on souhaite afficher.

⇒ Recopiez la requête suivante et observez le résultat :

```
SELECT "nom", "sexe", "date_naissance" FROM "ANIMAL";
```

⇒ Sur le même principe, affichez les colonnes nom et population de la table ESPECE

Le langage SQL offre aussi la possibilité de renommer une colonne lors de l'affichage du résultat :

⇒ Recopiez la requête suivante et observez le résultat :

```
SELECT "nom" AS "Nom", "sexe" AS "Sexe", "date_naissance" AS "Date de
    naissance" FROM "ANIMAL";
```

⇒ De manière similaire, affichez les colonnes *nom* et *nom_ scientifique* de la table ESPECE en les renommant respectivement en *Nom* et *Nom scientifique*.

5.2 La clause WHERE

Il est possible de restreindre le résultat d'une requête avec le mot clé `WHERE`.

⇒ Par exemple, la requête suivante sélectionne les animaux né avant 2000.

```
SELECT * FROM "ANIMAL" WHERE "date_naissance" < '2000-01-01';
```

⇒ En utilisant une clause `WHERE`, sélectionnez les espèces dont la population est inférieure à 10000.

L'opérateur de comparaison pour les chaînes de caractères est *like* :

```
SELECT * FROM "ANIMAL" WHERE "sexe" like 'M';
```

Opérateur	Signification
=	égal
<	inférieur
<=	inférieur ou égal
>	supérieur
>=	supérieur ou égal
<> ou ! =	différent
<=>	égal (valable aussi pour `NULL`)

⇒ Sélectionnez la ligne de l'espèce tigre.

⇒ Créez une requête pour afficher les colonnes *nom* et *population* des espèces ayant une population supérieure à 100000.

Pour des résultats plus évolués, on peut combiner différents critères. Par exemple pour sélectionner les animaux "éléphant" (code_espece = 2) et "girafe" (code_espece = 4) :

```
SELECT "nom", "sexe", "date_naissance", "date_arrivee" FROM "ANIMAL" WHERE
    code_espece = 2 OR code_espece = 4;
```

Opérateur	Signification
`AND`	ET
`OR`	OU
`NOT`	NON

⇒ Affichez le nom et la date d'arrivée au zoo (titre de colonne renommé en "Date d'arrivée") des animaux de sexe féminin arrivés dans le zoo avant 2010.

⇒ Sélectionnez les animaux qui sont soit des mâles, soit des girafes (code_espece = 4), mais pas les deux en même temps.

⇒ Imaginons maintenant une autre requête un peu plus complexe qui nous permette de sélectionner les éléphants nés avant 2000 et les animaux de toutes les autres espèces qui soient nés arrivées au zoo avant 2007 pour les femelles et avant 2005 pour les mâles.

5.3 Clauses WHERE évoluées

Les clauses `WHERE` que nous avons utilisé jusqu'ici permettent de tester des conditions simples : égalité, inférieur, supérieur. Mais qu'en est-il si nous souhaitons sélectionner les animaux dont le nom commence par la lettre T.

Par ailleurs, la sélection de ceux arrivés au parc entre 2000 et 2005, ou bien de ceux dont le code_espece est 1, 3 ou 6, est possible avec les expressions déjà introduites, mais la syntaxe résultante sera assez lourde à mettre en place.

Le SQL nous met à disposition des expression pour ce genre de requêtes. Pour les tests sur les chaînes de caractères, on peut utiliser le mot clé `LIKE` ("comme") qui permet de tester si une chaîne de caractère ressemble à une autre :

```
SELECT * FROM table WHERE colonne LIKE valeur;
```

La valeur testée peut contenir les symboles _ pour remplacer un caractère inconnu, ou % pour remplacer plusieurs caractères inconnus.

⇒ Sélectionnez les animaux dont le nom commence par T :

```
SELECT * FROM "ANIMAL" WHERE "nom" LIKE 'T%';
```

⇒ De la même manière, sélectionnez les animaux dont le nom comporte 'ou'.

Pour tester si une colonne est comprise entre deux valeurs on peut utiliser `BETWEEN(valeur_min, valeur_max)`. Pour tester si elle est dans une plage de valeurs, on utiliser `IN(valeur1, valeur2, ...)`

⇒ Sélectionnez les animaux dont le nom commence par T, C ou B.

⇒ Sélectionnez les animaux arrivés au parc entre 2005 et 2010.

5.4 Tri des données

Jusqu'à présent, lorsque l'on a exécuté une requête `SELECT`, les lignes sélectionnées sont retournées dans un ordre défini par PostgreSQL et qui n'a pas forcément de sens pour nous. Or il arrivera fréquemment que l'on veuille trier nous même le résultat de la requête. On utilisera pour cela `ORDER BY colonne` à la fin de la requête.

⇒ Sélectionnez les éléphants en ordonnant le résultat par ordre alphabétique.

Pour inverser l'ordre de la sélection, on utilise le mot clé `DESC` :

```
SELECT "nom", "sexe", "date_naissance" FROM "ANIMAL" ORDER BY "date_naissance"
    DESC;
```

On peut également trier sur plusieurs colonnes : `ORDER BY colonne1, colonne2`.

⇒ Sélectionnez les animaux en triant le résultat par ordre d'arrivée dans le zoo et en cas d'arrivées le même jour, en mettant l'animal le plus vieux en premier.

5.5 Limiter les résultats

Sur les grosses bases de données, le nombre de résultats retournés par une requête de sélection peut être énorme et faire baisser les performances de la machine. Aussi, pour limiter l'affichage à un nombre donné

de ligne, on peut utiliser la clause `LIMIT` :

```
LIMIT nombre_de_ligne [OFFSET decalage];
```

⇒ Testez les deux requêtes suivantes pour comprendre le fonctionnement de la clause `LIMIT` :

```
SELECT * FROM "ANIMAL" ORDER BY "nom";

SELECT * FROM "ANIMAL" ORDER BY "nom" LIMIT 4 OFFSET 2;
```

⇒ Affichez le nom des 3 animaux les plus jeunes du zoo.

5.6 Eliminer les doublons

Si l'on exécute une requête `SELECT "code_espece"FROM "ANIMAL";` pour récupérer la liste des codes espèce des animaux du zoo, le résultat nous présentera plusieurs fois le même code si plusieurs animaux de la même espèce sont présents. Cela ne sera pas très exploitable.

Pour supprimer les doublons de l'affichage, on peut utiliser le mot clé `DISTINCT`, placé juste après le `SELECT`.

⇒ Affichez la liste des codes espèce des animaux présent dans le zoo en éliminant les doublons.

On peut également utiliser l'option `GROUP BY (colonne1, colonne2...)` pour fusionner les résultats ayant les mêmes valeurs dans une colonne ou un groupe de colonnes données.

⇒ Testez les requête :

```
SELECT "code_espece" FROM "ANIMAL";

SELECT "code_espece" FROM "ANIMAL" GROUP BY "code_espece";
```

Cette seconde option permet, en plus, d'effectuer des comptages sur les résultats :

- `count(*)` : compte le nombre de lignes regroupées
- `sum(colonne)` : fait la somme des valeurs de la colonne indiquée pour les lignes regroupées
- ...

Ces comptages s'affichent comme des colonnes classiques dans la requête.

⇒ Pour comprendre le fonctionnement, testez la requête :

```
SELECT "code_espece", count(*) AS "Nombre" FROM "ANIMAL" GROUP BY "code_espece
    ";
```

⇒ Pour chaque espèce, affichez le nombre d'aliment qu'elle mange et la quantité totale que cela représente.

6 Jointures

Jusqu'à présent nous avons uniquement manipulé les tables unes à unes. Mais l'utilisation des bases de données ne se limite à cela. Elles permettent en effet de mettre les tables en relation et de croiser les

informations pour effectuer certaines analyses. On parlera de jointures.

Il existe plusieurs types de jointures. Le premier que nous allons voir est la jointure interne.

⇒ Observez le résultat de la requête :

```
SELECT "ANIMAL"."nom", "ANIMAL"."code_espece", "ESPECE"."nom"
FROM "ANIMAL" INNER JOIN "ESPECE" ON "ANIMAL"."code_espece" = "ESPECE"."
    code_espece";
```

⇒ Affichez le nom de chaque animal, le numéro de son enclos et la situation de l'enclos.

Si l'on revient sur la première jointure, le résultat fait apparaître deux colonnes *nom*, ce qui peut n'est pas très pratique pour une exploitation future. On prendra donc soin d'utiliser pertinemment des alias.

⇒ Ré-écrivez la requête en renommant les champs en sortie.

On peut également utiliser des alias pour renommer les tables et ainsi avoir moins de code à écrire :

```
SELECT "A"."nom", "E"."nom"
FROM "ANIMAL" AS "A" INNER JOIN "ESPECE" AS "E" ON "A"."code_espece" = "E"."
    code_espece";
```

⇒ Sélectionnez le nom des animaux dont la population de l'espèce est inférieure à 50000.

La jointure interne ne permet de sélectionner que les lignes qui correspondent dans les deux tables. Pour sélectionner tous les éléments de la table de référence (aussi appelée table de gauche) et ceux correspondant de la table jointe (ou table de droite), on effectue une jointure par la droite.

La jointure inverse (jointure par la gauche) est également possible.

⇒ Testez et comparez :

```
SELECT "A"."nom"
FROM "ALIMENT" AS "A" INNER JOIN "ESP_ALI" AS "EA" ON "A"."code_aliment" = "EA
    "."code_aliment";

SELECT "A"."nom"
FROM "ALIMENT" AS "A" LEFT JOIN "ESP_ALI" AS "EA" ON "A"."code_aliment" = "EA"
    ."code_aliment";

SELECT "A"."nom", "EA"."quantite", "EA"."code_espece"
FROM "ALIMENT" AS "A" LEFT JOIN "ESP_ALI" AS "EA" ON "A"."code_aliment" = "EA"
    ."code_aliment";
```

⇒ Affichez les noms des animaux vivant dans chaque enclos, avec la situation de l'enclos. Les enclos vide doivent apparaître dans la liste. Triez le résultat par numéro d'enclos.

Il est possible d'effectuer plusieurs jointures dans la même requête :

⇒ Affichez la table ALI_SUBSTITUTION en remplaçant code_aliment et code_substitution par le nom de l'aliment.

Un petit problème...

Pour optimiser les tournées du personnel nourrissant les animaux, on désire connaître à l'avance les quantités et types de nourriture à apporter dans un enclos donné.

⇒ Dans un premier temps, sélectionnez tous les animaux de l'enclos 2. Affichez leur nom, les aliments qu'ils consomment et la quantité consommée.

⇒ Affinez la requête pour afficher la quantité consommée par type d'aliment. On souhaite maintenant savoir si les stocks d'aliments sont suffisant pour nourrir tous les animaux du zoo.

Un second problème...

On souhaite maintenant savoir quels sont les aliments pour lesquels le stock est inférieur à la quantité consommée.

⇒ Pour commencer, afficher pour chaque animal : son nom, les noms des aliments qu'il consomme et la quantité consommée.

⇒ Ajoutez le stock de l'aliment à la requête.

⇒ Sélectionnez dans le résultat de cette dernière requête les lignes pour lesquelles le stock est inférieure à la quantité consommée.

Si cet opération retourne bien des aliments pour lesquels le stock n'est pas suffisant, on notera que le résultat peut être incomplet si plusieurs animaux consomment le même aliment.

⇒ Modifiez la requête pour pouvoir afficher le stock et la somme des quantités consommées.

Dans la requête ainsi construite, il n'est pas possible de comparer les colonnes dans la clause `WHERE`. On va utiliser une vue pour pouvoir y arriver. Une vue dans une base de données est une synthèse d'une requête de sélection. On peut la voir comme une table virtuelle, dépendant du contenu des autres tables et définie par une requête.

La syntaxe est :

```
CREATE VIEW nom_de_la_vue AS requete_select;
```

⇒ Créez une vue à partir de votre dernière requête et sélectionnez les lignes de la vue pour lesquelles le stock est inférieur à la quantité totale consommée.

7 Sous-requêtes

Une sous-requête est une requête imbriquée dans une autre requête. Avec ce mécanisme, la complexité des requêtes peut rapidement augmenter, mais il sera alors aussi possible de faire en une seule fois ce qui aurait parfois demandé plusieurs étapes auparavant.

Une sous-requête peut se faire, sous certaines conditions, dans des requêtes de type `SELECT`, `INSERT`, `UPDATE` et `DELETE`. Pour ce qui est des requêtes de sélection avec sous-requêtes, elles sont généralement réalisables

avec des jointures. La sous-requête aura peut-être pour avantage un peu plus de la clarté, la jointure étant légèrement plus performante.

7.1 Sous-requête dans le FROM

Une requête de type `SELECT` retourne une table. Le principe de la sous-requête dans le `FROM` sera d'utiliser cette table résultante comme entrée d'une nouvelle requête :

```
SELECT colonne FROM (
SELECT colonne1, colonne2... FROM table WHERE condition
) AS alias_de_requete;
```

⇒ Sélectionnez la date de naissance de l'animal le plus âgé.

7.2 Sous-requête dans la clause WHERE

⇒ Exécutez la requête suivante et décrivez son fonctionnement.

```
SELECT "nom", "nom_scientifique", "population"
FROM "ESPECE"
WHERE "code_espece" = (SELECT "code_espece" FROM "ANIMAL" WHERE "nom" LIKE '
    Tintin');
```

⇒ En adaptant un peu la requête précédente, sélectionnez les noms, noms scientifiques et populations des espèces des animaux dont le nom commence par 'C'.

Une autre possibilité intéressante des sous-requêtes est l'utilisation des conditions `EXISTS` ou `NOT EXISTS` : la requête principale ne sera exécutée que si la sous-requête renvoie quelque chose.

```
SELECT *
FROM "ESPECE"
WHERE EXISTS (SELECT * FROM "ANIMAL" WHERE "nom" = 'Tintin');
```

En l'état, cette requête n'a pas forcément trop d'intérêt. Cela devient beaucoup plus intéressant si on corrèle la sous-requête avec la requête :

```
SELECT *
FROM "ESPECE"
WHERE EXISTS (SELECT * FROM "ANIMAL" WHERE "ESPECE"."code_espece" = "ANIMAL"."
    code_espece");
```

La table ESPECE de la requête principale est ici réutilisée dans la sous-requête. Plus précisément, la sous-requête est évaluée pour chaque ligne de la requête principale.

⇒ Sélectionnez les enclos occupés en utilisant une sous requête corrélée.

⇒ Sélectionnez les enclos vides en utilisant une sous requête corrélée.

8 Modification, suppression de données

Dans cette partie, nous présenterons le dernier type de requêtes que nous aborderons dans ce TD. Elles concerne la modification ou la suppression des enregistrements d'une table.

8.1 Modification

⇒ Ajoutez une remarque à l'enregistrement de l'animal nommé Nina à l'aide de la requête suivante :

```
UPDATE "ANIMAL" SET "remarques" = 'blablabla' WHERE "nom" = 'Nina';
```

⇒ Après l'avoir recherchée sur internet, renseignez la population totale de crocodiles.

Comme pour une requête `SELECT`, la clause `WHERE` est optionnelle.

⇒ En jouant sur la clause `WHERE`, supprimez toutes les remarques de la table ANIMAL.

8.2 Suppression

La syntaxe d'une requête de suppression est :

```
DELETE FROM table WHERE condition';
```

L'enclos au numéro 4 de l'allée principale du zoo doit être fermé pour travaux. Un nouvel enclos situé juste en face (au numéro 3) va ouvrir en remplacement. ⇒ Effectuer les modifications dans la base pour prendre en compte ces changements (l'enclos au numéro 4 de l'allée principale du zoo doit être supprimé de la base).

9 Annexe : syntaxes SQL

La liste complète des commandes SQL supportées par PostgreSQL est disponible sur la page : http://www.postgresql.org/docs/9.4/static/sql-commands.html. Les paragraphes suivants reviennent sur les commandes les plus utilisées.

9.1 Création et suppression d'une base de données

La commande pour créer une base de données est la suivante :

```
CREATE DATABASE nom_base;
```

Il peut cependant être utile de préciser quelques options comme l'encodage, le propriétaire ou le nombre maximum de connexions simultanées (-1 pour pas de limite) :

```
CREATE DATABASE nom_base
  WITH OWNER=postgres
       ENCODING='UTF8'
       CONNECTION LIMIT=-1;
```

Pour supprimer une base de données, la commande à utiliser est la suivante :

```
DROP DATABASE nom_base;
```

9.2 Création, suppression d'une table

La commande pour créer une table est la suivante :

```
CREATE TABLE nom_table (
  colonne1 description_colonne1,
  [colonne2 description_colonne2,
  colonne3 description_colonne3,
  ...,]
  [[CONSTRAINT nom_contrainte] contrainte]
);
```

La description de la colonne contient au minimum le type de données (entier, chaîne de caractère, date, etc.), toutes les colonnes devant être typées dans le modèle relationnel.

Il est aussi possible de préciser le contenu autorisé dans la colonne, une valeur par défaut, des clé primaires et/ou étrangères :

- `NOT NULL` si la colonne ne peut être nulle ;
- `AUTO_INCREMENT` pour une colonne auto-incrémentée ;
- `CHECK (colonne > valeur)` pour vérifier que le contenu de la colonne respecte bien une condition ;
- `DEFAULT valeur_par_defaut` pour assigner une valeur par défaut ;
- `UNIQUE` pour s'assurer que le contenu de la colonne est unique dans chaque ligne de la table ;
- `PRIMARY KEY` pour indiquer que la colonne est une clé primaire (il s'agit en fait de la combinaison des contraintes `UNIQUE` et `NOT NULL`) ;

— `REFERENCE table (colonne)` pour indiquer une clé étrangère vers la colonne "colonne" de la table "table".

Chaque contrainte ajoutée à la table peut être nommée : `CONSTRAINT nom_contrainte contrainte`, où "contrainte" est une contrainte quelconque (`PRIMARY KEY`, `CHECK (colonne > valeur)`, etc.).

Si une contrainte concerne plusieurs colonnes, elle est ajoutée après les déclarations des colonnes :

```
[CONSTRAINT nom_contrainte] CHECK (colonne1 > colonne2)
[CONSTRAINT nom_contrainte] FOREIGN KEY (colonne1, colonne2) REFERENCE
    autre_table (autre_colonne1, autre_colonne2)
```

La suppression d'une table est réalisée avec la commande :

```
DROP TABLE nom_table;
```

Il est à noter que le comportement lors de la suppression d'éléments référencés dans une clé étrangère est paramétrable :

— `REFERENCE table (colonne)ON DELETE RESTRICT` pour indiquer de bloquer la suppression ;
— `REFERENCE table (colonne)ON DELETE CASCADE` pour indiquer de supprimer également les éléments référençant cette clé.

9.3 Modification d'une table

Pour modifier une table, on utilisera la commande `ALTER TABLE nom_table` suivie de l'opération à effectuer sur la table :

```
ALTER TABLE nom_table ADD ... -- permet d'ajouter quelque chose (colonne,
    contrainte)
ALTER TABLE nom_table DROP ... -- permet de supprimer quelque chose
ALTER TABLE nom_table ALTER COLUMN ...  -- permet de modifier une colonne

ALTER TABLE nom_table RENAME TO nouveau_nom_table -- pour renommer la table
ALTER TABLE nom_table RENAME COLUMN colonne TO nouveau_nom_colonne -- pour
    renommer une colonne
```

Par exemple, pour l'ajour d'une colonne :

```
ALTER TABLE nom_table ADD COLUMN colonne description;
```

La suppression d'une colonne suivra le modèle ci-dessous :

```
ALTER TABLE nom_table DROP COLUMN colonne;
```

9.4 Insertion de données

La commande `INSERT` est utilisée pour ajouter des enregistrements dans une table. La syntaxe est la suivante :

```
INSERT INTO table ([colonne1, colonne2, ...]) VALUES (expression1, expression2
    , ...);
```

Il est possible soit de préciser les valeurs pour toutes les colonnes de la table, soit de n'en indiquer que certaines. Dans ce dernier cas, il faut préciser les noms des colonnes pour lesquelles ont renseigne la valeur.

Les expressions insérées peuvent être le résultat d'une requête de sélection (voir paragraphe suivant).

9.5 Sélection de données

La syntaxe pour la sélection de données utilise la commande `SELECT`. La clause `FROM` permet de spécifier la(les) tables(s) sources. La clause `WHERE` permet quand à elle de préciser des conditions de sélection : toute ligne qui ne rempli pas la clause `WHERE` n'est pas affichée dans le résultat final.

```
SELECT colonne1 [, colonne2 ...] FROM table1 [, table2 ...] [WHERE condition];
```

Il est possible de préciser la manière d'afficher les résultats de la requête en ajoutant à l'instruction :

```
GROUP BY expression -- pour grouper sur une seule ligne les resultats qui
    partagent une valeur commune
ORDER BY expression|ASC|DESC -- pour trier dans un ordre specifique les
    resultats
LIMIT nombre -- pour specifier le nombre maximal de ligne a retourner
OFFSET nombre -- pour ne pas afficher les premiers enregistrements
    selectionnes
```

Pour les chaînes de caractères, le mot clé `LIKE` utilisé dans la condition est utilisé pour tester la ressemblance de deux valeurs :

```
SELECT * FROM table WHERE colonne LIKE valeur;
```

valeur pouvant contenir les symboles _ pour remplacer un caractère inconnu, ou % pour remplacer plusieurs caractères inconnus.

La condition peut également être de type :

- `BETWEEN(valeur_min, valeur_max)` : compris entre valeur_min et valeur_max
- `IN(valeur1, valeur2, ...)` : égal à valeur1 ou valeur2 ou ...

9.6 Suppression de données

La syntaxe pour supprimer une ligne ou un groupe de lignes dans une table est :

```
DELETE FROM table WHERE condition;
```

La condition peut être laissée vide pour supprimer tous les enregistrements de la table.

9.7 Jointure

Jointure interne (données nécessaires de part et d'autre de la jointure) :

```
SELECT *                    -- vous selectionnez les colonnes que vous voulez
FROM nom_table1
```

```
[INNER] JOIN nom_table2   -- INNER explicite la jointure interne, mais c est
    facultatif
ON colonne_table1 = colonne_table2

[WHERE ...]
[ORDER BY ...]            -- les clauses habituelles sont bien sur utilisables
    !
[LIMIT ...]
```

Jointure externe (sélectionne également les lignes sans correspondance dans la table jointe (jointure à gauche) / table de référence (jointure à droite)) :

```
SELECT *                  -- vous selectionnez les colonnes que vous voulez
FROM nom_table1
[LEFT | RIGHT] JOIN nom_table2
ON colonne_table1 = colonne_table2

[WHERE ...]
[ORDER BY ...]            -- les clauses habituelles sont bien sur utilisables
    !
[LIMIT ...]
```