SQL - Bases de données

Ministère de la Transition Ecologique et de la Cohésion des Territoires Licence ouverte ETALAB

janv 2025

Table des matières

Introduction

Ce module va vous permettre de :

- Connaître les rudiments du SQL
- Savoir utiliser et gérer des tables dans une base Spatialite

Version PDF du module 4 (cf. M04_SQL_BDD_papier.pdf)

Notions SQL

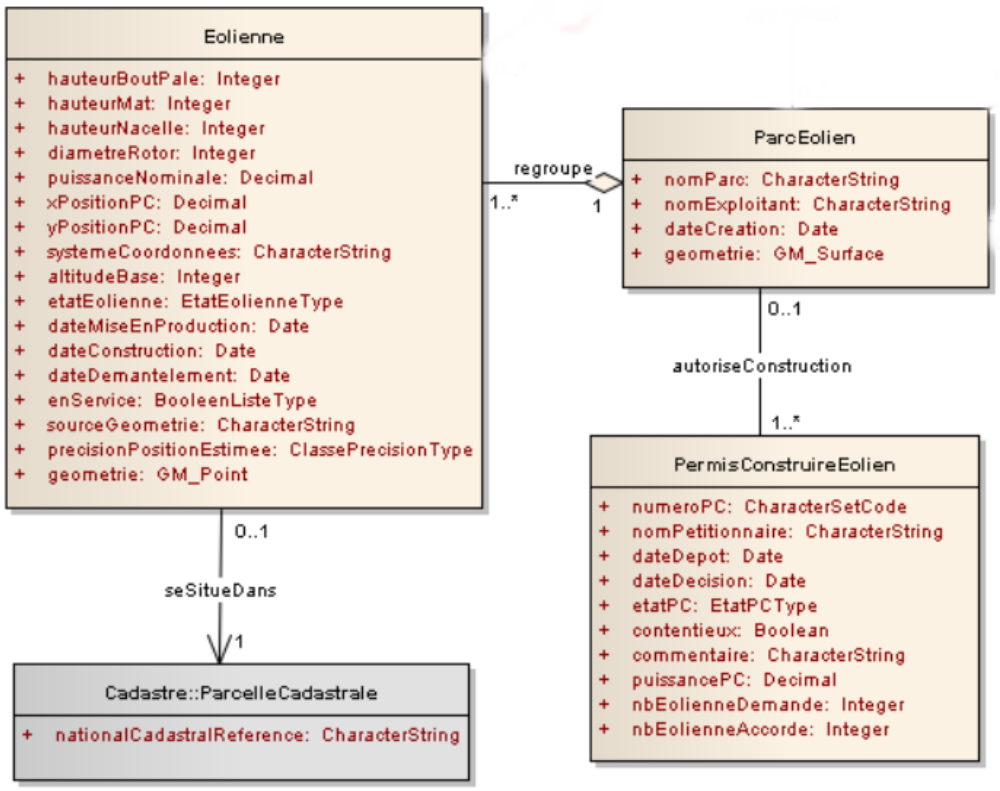
1. Introduction

Introduction aux SGBDR

Un **S**ystème de **G**estion de **B**ase de **D**onnées (**SGBD**) est un logiciel permettant de stocker de la donnée dans une base de données en garantissant la qualité, la pérennité et la confidentialité des informations.

La complexité des opérations de traitement des données ne nécessite pas d'être totalement connue par les utilisateurs. Ce module ne vise donc pas à former des spécialistes des SGBD. Les SGBD[1] que nous utiliserons sont basés sur un modèle de données **relationnel** (SGBD**R**).

Dans ce modèle, la base de données est composée d'un ensemble de **tables** et chaque ligne d'une table est un **enregistrement**.



Extrait du modèle relationnel du standard COVADIS de l'Éolien terrestre (formalisme UML)

La conception et la gestion d'une base de données relationnelle sont un domaine en soi qui est hors du périmètre de cette formation. Les personnes désireuses d'en savoir plus sont invitées à suivre le stage 'Concevoir et structurer des bases de données géographiques' .

Dans cette formation nous n'exploiterons que des modèles très simples. Nous utiliserons le SGBD **Spatialite** installé avec QGIS (un SGBD très léger).

1. http://fr.wikipedia.org/wiki/Syst%C3%A8me_de_gestion_de_base_de_donn%C3%A9es#Typologie

Il peut être qualifié de 'bureautique' dans le sens où il est plutôt orienté vers un usage personnel sur son poste de travail.

Le SGBD **PostGIS** est beaucoup plus complet et avancé fait l'objet d'une formation spécifique[2] (orientée vers les administrateurs de données).

Il doit être envisagé, pour ce qui est du partage de données, comme un composant du système d'information d'un service avec un administrateur dédié. Une utilisation personnelle de ce SGBD est cependant envisageable pour certains besoins d'analyse ou de production de données.

SQL

SQL (Structured Query Language signifie ***l*angage de *requêtes structurées*** est un langage destiné à la manipulation des bases de données au sein d'un SGBD.

SQL est composé de trois sous-ensembles :

- Le **L**angage de **D**éfinition de **D**onnées (**LDD**) permet de créer et supprimer des objets dans la base de données et que nous n'aborderons pas explicitement dans le cadre de cette formation.
- Le **L**angage de **C**ontrôle de **D**onnées (**LCD**) permet de gérer les droits sur les objets et que nous n'aborderons pas.
- Le **La**ngage de **M**anipulation de **D**onnées (**LMD**) permet la recherche, l'insertion, la mise à jour et la suppression de données et sera le seul abordé partiellement.

A noter que le **SQL** est utilisé également dans QGIS dans les requêtes de filtrages sur les tables et qu'il est également possible au travers du mécanisme des *virtual layer* d'utiliser le SQL sur les couches ouvertes dans QGIS.

Les Virtuals layers[3] utilisent de façon sous-jacente le mécanisme des drivers virtuels de **Sqlite**. Ils ne portent pas directement les données, mais sont des Vues (requête SQL) sur d'autres couches.

Se former à SQL... *Conseil*

Dans les pages suivantes nous survolons le SQL pour une première approche. Il existe une formation '*initiation au langage SQL*' régulièrement organisée par le réseau formation en présentiel ou en distanciel sur 3 jours.

2. La sélection

Syntaxe générale

La requête de sélection est la base de la recherche de données en SQL.

Une requête **SQL** respecte une syntaxe de type :

SELECT (liste des attributs) **FROM** (liste des tables) **WHERE** (Conditions)

La partie **SELECT** indique le sous-ensemble des attributs (les colonnes) qui doivent apparaître dans la réponse.

La partie **FROM** décrit les relations (les tables) qui sont utilisées dans la requête.

Les attributs de la clause **SELECT** doivent appartenir aux tables listées dans la clause **FROM**.

La partie **WHERE** exprime les conditions, elle est optionnelle.

Nous verrons d'autres options plus tard...

ex 1: `SELECT * FROM commune WHERE population > 1000`

2. http://www.geoinformations.developpement-durable.gouv.fr/postgis-support-pedagogique-a3347.html
3. https://docs.qgis.org/latest/fr/docs/user_manual/managing_data_source/create_layers.html?highlight=virtual%20layers#creating-virtual-layers

sélectionne les enregistrements de la table COMMUNE dont la population est supérieure à 1000 avec tous les attributs (c'est le sens de *) de la table COMMUNE

ex 2 : **`SELECT nom_comm, insee_comm, population FROM commune`**

sélectionne tous les enregistrements de la table COMMUNE (cf pas de conditions, c'est à dire pas de clause WHERE) et renvoi une table avec les attributs NOM_COM, INSEE_COMM et POPULATION.

Résultat :

	NOM_COMM	INSEE_COMM	POPULATION
1	SAINT-JEAN-DE-LA-MOTTE	72291	900
2	ARTHEZE	72009	400
3	VAULANDRY	49380	300
4	CLEFS	49101	900

résultats de la sélection sur la table COMMUNE

Il est possible de donner un nom d'alias aux attributs en sortie avec le mot clef **AS**.

Ex 3 : `SELECT nom_comm AS COMMUNE , insee_comm AS INSEE, population FROM commune`

on peut également écrire directement (on omet le **AS**) :

`SELECT nom_comm COMMUNE , insee_comm INSEE, population FROM commune`

Résultat :

	COMMUNE	INSEE	POPULATION
1	SAINT-JEAN-DE-LA-MOTTE	72291	900
2	ARTHEZE	72009	400
3	VAULANDRY	49380	300
4	CLEFS	49101	900

Utilisation des alias de nom de colonne

3. Les opérateurs de comparaison et les opérateurs logiques

Les opérateurs de comparaison

La clause **WHERE** est définie par une condition qui s'exprime à l'aide d'opérateurs de comparaison et d'opérateurs logiques.

Les opérateurs de comparaison sont :

A = B

A <> B (différent)

A < B

A > B

A <= B (inférieur ou égal)

A >= B (supérieur ou égal)

A BETWEEN B AND C (compris entre B et C)

A IN (B1, B2,...) liste de valeurs :

ex : `SELECT nom_comm, insee_comm, population FROM commune WHERE statut IN('Commune simple', 'Chef-lieu de canton')`

A LIKE 'chaîne'

permet d'insérer des caractères jokers dans l'opération de comparaison, % désignant 0 à plusieurs caractères quelconques , _ désignant un seul caractère.

Ex : `SELECT * FROM commune WHERE nom_comm LIKE 'A%'` sélectionne toutes les communes dont le nom commence par A

`SELECT * FROM commune WHERE nom_comm LIKE '%SAINT%'` sélectionne toutes les communes dont le nom contient la chaîne 'SAINT'

Sensibilité à la casse *Attention*

SQL est **sensible à la casse (majuscule / minuscule)** pour les constantes, ainsi NOM_COMM LIKE '%A' est différent de NOM_COMM LIKE '%a'.

Les mots clefs et les noms de colonnes sont insensibles à la casse. On peut ainsi écrire SeLecT * fRom ma_TabLE. Une convention couramment utilisée est d'écrire les mots clefs en majuscule et les noms en minuscules exemple : `SELECT * FROM ma_table`.

Dans PostgreSQL mettre les noms de colonnes entre guillemets double permet de les rendre sensibles à la casse, "ma_table" est différent de "MA_TABLE". Il est conseillé de donner des noms de champs en minuscules dans PostgreSQL.

Les chaînes de caractères des constantes sont en général entourées de guillemets simples (ex : 'SAINT %') qui est le caractère chr(39), cependant si la chaîne constante contient elle-même une apostrophe il faut la doubler (ex : `SELECT * FROM commune WHERE nom_com LIKE 'l'' %'` sélectionne toutes les communes dont le nom commence par l'

Nombre ou chaîne de caractères *Remarque*

Les opérandes (A ou B) peuvent être des nombres ou des chaînes de caractères.

Ainsi NOM_COMMUNE <> 'PARIS' est correct et sélectionne toutes les communes dont le nom n'est pas Paris

NULL *Fondamental*

Une valeur par défaut peut-être attribuée à une colonne lors de la définition d'une table. Si aucune valeur par défaut n'est attribuée la valeur par défaut de la colonne est positionnée à NULL (0 ou espace n'est pas équivalent à NULL).

Il est possible d'utiliser l'opérateur logique **IS** pour tester si un champ est ou non nul.

Exemple : `SELECT * FROM commune WHERE nom_comm IS NULL` récupère les enregistrement qui n'ont pas de nom de commune.

`SELECT * FROM commune WHERE nom_comm IS NOT NULL` récupère ceux qui ont effectivement un nom (non positionné à NULL).

Les opérateurs logiques

OR : pour séparer deux conditions dont au moins une doit être vérifiée.

Ex : `SELECT * FROM commune WHERE statut = 'Commune simple' OR STATUT = 'Chef-lieu de canton'`

Cette requête sélectionne les communes pour lesquelles le statut est commune simple ou chef-lieu de canton.

Bien penser dans l'exemple ci-dessus que le OR lie deux conditions. Une condition contient nécessairement un des opérateurs de comparaison. Ainsi on ne peut écrire

```
SELECT * FROM commune WHERE statut = 'Commune simple' OR 'Chef-lieu de canton'
```

AND : pour séparer deux conditions qui doivent être vérifiées simultanément.

Ex : `SELECT * FROM commune WHERE statut = 'Sous-préfecture' AND population > 10000`

seules les sous-préfectures de plus de 10 000 habitants sont sélectionnées.

Attention, l'opérateur AND ne peut être utilisé pour vérifier des conditions basées sur le même champs, auquel cas il sera impossible que les conditions soient respectées.

Par exemple, une commune ne peut pas être à la fois chef-lieu de canton et commune simple. Ainsi la requête ci-dessous ne renvoie aucun enregistrement :

```
SELECT * FROM commune WHERE statut = 'Commune simple' AND statut = 'Chef-lieu de canton'
```

NOT : permet d'inverser une condition.

Ex : `SELECT * from commune WHERE NOT (statut = 'Commune simple' OR statut = 'Chef-lieu de canton')`

sélectionne les communes qui ne sont ni commune simple, ni chef lieu de canton.

4. Les types de données et les fonctions

Les types de données

Les principaux types de données en SQL sont :

CHARACTER (ou **CHAR**) : valeur alpha de longueur fixe.

CHARACTER VARYING (ou **VARCHAR**) : valeur alpha de longueur maximale fixée.

TEXT : suite longue de caractères (sans limite de taille).

NUMERIC (ou DECIMAL ou DEC) : décimal

INTEGER (ou **INT**) : entier long

REAL : réel à virgule flottante dont la représentation est binaire.

BOOLEAN (ou LOGICAL) : vrai/faux

DATE : date du calendrier grégorien.

Le typage des données *Remarque*

SQLite propose une gestion spécifique et simplifiée[4] des types de données.

PostgreSQL propose une gestion beaucoup plus complète[5]. Certains de ses types sont spécifiques (non normalisés).

Les extensions spatiales de ces **SGBDR** ajoutent des types géométriques (points, lignes,...) que nous verrons plus tard

4. http://fr.wikipedia.org/wiki/SQLite#Types_de_donn.C3.A9es
5. http://docs.postgresql.fr/14/datatype.html

Les fonctions

SQL propose des **fonctions** dont on trouvera une description par exemple ici[6]

En voici quelques unes...

Fonctions de transtypage:

cast (expr as type) : Est la fonction standard SQL qui permet de convertir un type en un autre.

Exemple :

Si x_commune est un champ de type INTEGER dans la table commune

```
SELECT x_commune FROM commune LIMIT 1
```

renvoi 478935

(noter l'utilisation de la clause **LIMIT** qui permet d'indiquer le nombre maximum d'enregistrements en retour.

Il est également possible d'utiliser la clause **OFFSET** pour décaler le nombre de lignes à obtenir

ex :

`SELECT * FROM commune LIMIT 10 OFFSET 5` (pour renvoyer les enregistrements de 6 à 15)

`SELECT cast(x_commune as real) FROM commune LIMIT 1` renvoie 478935.0

`SELECT cast(x_commune as text) FROM commune LIMIT 1` renvoie '478935' c'est à dire une chaîne de caractère, puisque entre ''.

PotsgreSQL propose une notation compacte sous la forme expr::type

exemple : `SELECTx_commune :: real FROM commune`

Une opération de transtypage est parfois nécessaire pour obtenir le résultat souhaité, en particulier avec **SpatiaLite**.

Prenons l'exemple de calcul d'un indicateur (ratio de deux entiers) avec **SpatiaLite**.

Exemple : `SELECT (population/superficie) AS densite FROM commune LIMIT 10`

renvoie :

Résultat :

	densite
1	0
2	0
3	0
4	0

non utilisation du cast avec spatialite

Ce résultat est inattendu !

Il est dû au fait que dans **SpatiaLite**, le résultat de la division de deux entiers est un entier.

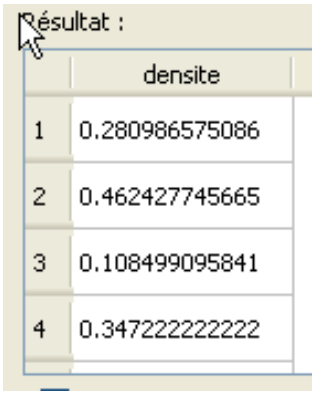
Pour obtenir un résultat satisfaisant il faut au minimum convertir le numérateur ou le dénominateur en flottant:

```
SELECT cast(population as float)/superficie AS densite FROM commune
LIMIT 10
```

6. http://sqlpro.developpez.com/cours/sqlaz/fonctions/

On remarquera à nouveau l'utilisation de **LIMIT** qui permet d'indiquer le nombre maximum d'enregistrements retournés... c'est une clause très utile pour la mise au point de requêtes sur des grosses tables ou pour récupérer juste le premier enregistrement après un tri.

Le résultat devient bien celui attendu :


Résultat :

	densite
1	0.280986575086
2	0.462427745665
3	0.108499095841
4	0.347222222222

Utilisation de la fonction cast

Fonctions de chaînes de caractères :

LENGTH : renvoie la longueur d'une chaîne

exemple : `SELECT length(nom_comm) FROM commune`

CHR : renvoie le caractère correspondant au code ASCII (exemple CHR(184) renvoi ©)

|| : concatène deux chaînes (on obtient ce symbole en tapant ALTGr 6)

exemple : `SELECT nom_comm|| ' '||insee_comm FROM commune LIMIT 1`

renvoie 'SAINT-JEAN-DE-LA-MOTTE 72291'

SUBSTR : extraction d'une sous-chaîne de caractères substr(chaîne, position , longueur)

Exemple : `SELECT * FROM troncon_hydrographique WHERE substr(ID_BDCARTO, 1, 3) = '239'`

sélectionne tout les tronçons dont l'identifiant commence par '239'

UPPER : convertit en majuscule

LOWER : convertit en minuscule

exemple : `SELECT lower(nom_comm) FROM commune` renvoie les noms de communes en minuscules.

Fonctions mathématiques et numériques :

SQL dispose des fonctions mathématiques classiques... notons en particulier :

POW : pour élever à une puissance quelconque ex : POW(champ, 2) pour élever au carré.

SQRT : pour obtenir la racine carrée.

ROUND : qui permet d'arrondir un résultat

exemple : `SELECT round(cast(population AS float)/superfice,2) AS densite FROM commune`

renvoie :

Résultat :

	densite
1	0.28
2	0.46
3	0.11
4	0.35

fonction round

Remarque

sous **PostGIS** on écrira `SELECT (round (population/superficie) :: numeric, 2) AS densite FROM commune`

le :: étant une forme compacte sous **PostGIS** pour réaliser le cast. Le format numérique (numeric) étant obligatoire pour la fonction round sous **PostGIS**.

Sites de références pour les fonctions SQL dans spatialite et PostGIS ***Complément***

Les principales fonctions disponibles sous SpatiaLite sont décrites ici[7]

Les fonctions de PostgreSQL 14 sont décrites ici[8]

Nous vous conseillons vivement de parcourir une première fois ces sites et d'y revenir régulièrement par la suite...

5. Tri et agrégation

Tri

Il est possible de classer le résultat d'une requête en ajoutant le mot clef **ORDER BY** suivi d'une liste de champs.

Exemple : `SELECT * FROM commune ORDER BY nom_comm`

pour classer le résultat par nom de commune.

Un tri décroissant peut-être obtenu en ajoutant **DESC**.

Exemple : `SELECT * FROM commune ORDER BY nom_comm DESC`

```
SELECT nom_comm, round(cast(population as float)/superficie,2) AS densite FROM commune ORDER BY densite
```

retourne la densité de population par ordre croissant de densité.

7. http://www.sqlite.org/lang_corefunc.html
8. https://docs.postgresql.fr/14/functions.html

Résultat :

	NOM_COMM	densite
1	VAULANDRY	0.11
2	COURCELLES-LA-FORET	0.2
3	THOREE-LES-PINS	0.25
4	SAINT-JEAN-DE-LA-MOTTE	0.28
5	BOUSSE	0.33
6	CLEFS	0.35

Densite de population triée

Remarque

Sous **PostGIS** on écrira

```
SELECT nom_comm, round(population/superficie :: numeric,2) AS densite
FROM commune ORDER BY densite)
```

Agrégations

Une agrégation est une opération qui permet de regrouper les enregistrements de la table en sortie selon des critères et d'obtenir des informations statistiques sur ces regroupements. Il faut utiliser l'expression **GROUP BY** suivi du critère de regroupement.

Prenons un exemple à partir de la table **COMMUNE**. Nous souhaitons obtenir la population totale par département.

```
SELECT Nom_comm, nom_dept, population FROM commune
```

nous donne :

	NOM_COMM	NOM_DEPT	POPULATION
1	SAINT-JEAN-DE-LA-MOTTE	SARTHE	900
2	ARTHEZE	SARTHE	400
3	VAULANDRY	MAINE-ET-LOIRE	300
4	CLEFS	MAINE-ET-LOIRE	900
5	MAREIL-SUR-LOIR	SARTHE	600
6	BOUSSE	SARTHE	400
7	LE BAILLEUL	SARTHE	1200
8	CLERMONT-CREANS	SARTHE	1200
9	MALICORNE-SUR-SARTHE	SARTHE	2000
10	THOREE-LES-PINS	SARTHE	700
11	LA FONTAINE-SAINT-MARTIN	SARTHE	600
12	LA FLECHE	SARTHE	15400
13	VILLAINES-SOUS-MALICORNE	SARTHE	1000
14	CRE	SARTHE	800
15	CROSMIERES	SARTHE	900
16	SAINT-QUENTIN-LES-BEAUREPAIRE	MAINE-ET-LOIRE	300
17	BAZOUGES-SUR-LE-LOIR	SARTHE	1200
18	COURCELLES-LA-FORET	SARTHE	400
19	LIGRON	SARTHE	500

Population des communes

la requête :

```
SELECT nom_dept, sum(population) AS population_dept FROM commune GROUP BY nom_dept
```

renvoie :

	NOM_DEPT	population_dept
1	MAINE-ET-LOIRE	1500
2	SARTHE	28200

Agrégation par département

La clause **GROUP BY** fonctionne de concert avec les fonctions d'agrégation (ici sum()).

Les principales fonctions d'agrégation sont :

count() : renvoie le nombre d'enregistrements

sum() : renvoie la somme

max() : maximum

min() : minimum

avg() : moyenne

La clause HAVING *Complément*

Il se peut que l'on souhaite mettre un critère de sélection sur une colonne calculée par l'opération d'agrégation.

Dans l'exemple ça serait le cas si on souhaite n'afficher que les départements de plus de 20000 habitants.

On pourrait être tenté d'écrire une requête de la forme :

```
SELECT nom_dept, sum(population) AS population_dept FROM commune WHERE population_dept > 20000 GROUP BY nom_dept
```

Ca ne marche pas car la clause where est exécutée avant l'agrégation.

La clause **HAVING** permet d'indiquer au SQL d'effectuer une nouvelle sélection à la fin du calcul sur les résultats du regroupement.

On écrira donc:

```
SELECT nom_dept, sum(population) AS population_dept FROM commune GROUP BY nom_dept HAVING population_dept > 20000
```

Sous **PostgreSQL** il faut répéter la fonction d'agrégation dans la clause having

```
SELECT nom_dept, sum(population) AS population_dept FROM commune GROUP BY nom_dept HAVING sum(population) > 20000
```

nb : On n'utilisera la clause **HAVING** que dans le cas ou la sélection porte sur une colonne d'agrégation calculée, pour une sélection sur une colonne existante dans la table de départ on utilisera une condition dans la clause WHERE.

6. Extensions spatiales

Sqlite et **PostgreSQL** proposent des extensions spatiales (respectivement Spatialite et PostGIS) permettant d'ajouter le stockage et la manipulation d'objets spatiaux en ajoutant des types de données géométriques et des fonctions spatiales.

Les spécifications *Complément*

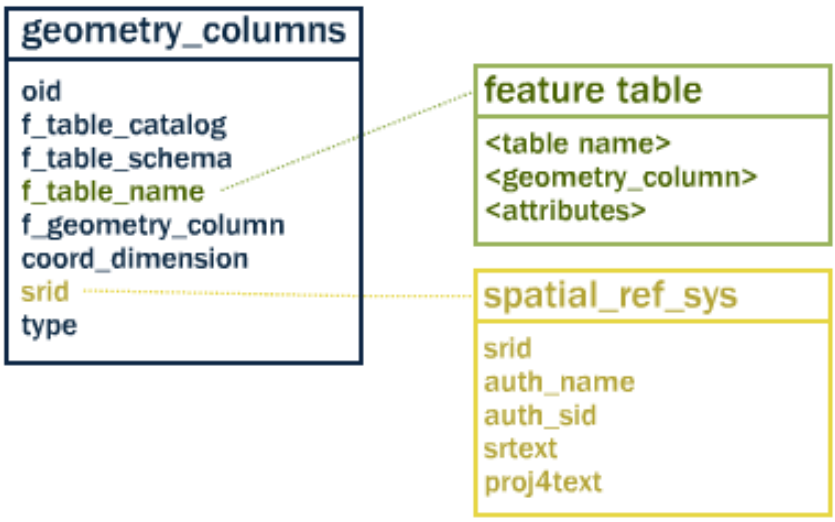
La spécification **SFSQL** (Simple Features for SQL[9]) définit les types et les fonctions qui doivent être disponibles dans une base de données spatiale selon l'OGC. La spécification SQL/MM étend le modèle. On pourra également se référer au document **Matrices de Clementini et prédicats spatiaux de l'OGC** .

Les types de données géométriques

Dans cette formation nous ne considérerons que les objets en dimension 2 et plus précisément : les points, les lignes, et les polygones.

La géométrie est stockée dans un format binaire '**WKB**' (ou éventuellement texte '**WKT**', exemple : POLYGON ((30 10, 10 20, 20 40, 40 40, 30 10)) dans une colonne de table qui est souvent nommée geometry ou the_geom). Le système utilise au moins deux autres tables internes supplémentaires qu'il maintient à jour : geometry_columns et spatial_ref_sys (PostGIS 1.5)

9. http://www.opengeospatial.org/standards/sfs



Tables internes OGC

SRID est l'identifiant du système de projection. Par exemple **2154 pour le RGF93/Lambert93**

Les fonctions spatiales

Il existe plusieurs catégories de fonctions spatiales, comme par exemple celles qui permettent de passer du format WKT au WKB ou inversement. Voici quelques fonctions de départ :

ST_SRID() : retourne le code du système de projection de l'objet

ST_IsValid() : vérifie la géométrie des objets (pas d'erreur topologique)... Ceci concerne essentiellement les polygones voir par exemple[10]

Remarque

Vérification de géométrie sous PostGIS

PostGIS ajoute d'autres fonctions de vérification de la géométrie

ST_IsValidReason() : retourne un texte indiquant les raisons d'une éventuelle invalidité.

ST_IsValidDetail() : retourne en plus un pointeur vers la partie non valide (à partir de PostGIS 2.0).

ST_MakeValid() : Tente de corriger les géométries invalides (PostGIS 2.0)

ST_X() : retourne la coordonnée X d'un point (et uniquement d'un point).

ST_Y() : coordonnée Y d'un point

ST_Centroid() : retourne le centroide d'un polygone

Exemple : `ST_X(ST_Centroid(Geometry))` retourne la coordonnée `X` du centroide d'un polygone.

```
SELECT nom_comm, ST_X(ST_centroid(Geometry)) AS X,
ST_Y(ST_centroid(Geometry)) AS Y FROM commune
```

10. http://www.postgis.fr/chrome/site/docs/workshop-foss4g/doc/validity.html

	NOM_COMM	X	Y
1	SAINT-JEAN-DE-LA-MOTTE	479450.144121	6743878.0476
2	ARTHEZE	467080.097228	6747807.62701
3	VAULANDRY	472864.934183	6727346.88669
4	CLEFS	469751.991174	6729698.50091
5	MAREIL-SUR-LOIR	475727.074588	6739744.85239
6	BOUSSE	470585.821961	6744536.6741
7	LE BAILLEUL	461895.99786	6746143.72513
8	CLERMONT-CREANS	473348.706355	6741255.95128
9	MALICORNE-SUR-SARTHE	469080.240414	6750303.39204

Utilisation St_Centroid

ST_Area() retourne la surface d'un objet

ST_Buffer() retourne un nouvel objet tampon construit autour d'un objet

ST_Length() : retourne la longueur d'un objet de type ligne ou multi-ligne (attention à ne pas utiliser length() qui retourne la longueur du champ, spatialite autorise aussi Glength()).

ST_Perimeter() : retourne le périmètre d'un objet polygone ou multi-polygone

Prefixe ST_ *Attention*

Il faut préfixer les commandes par ST_ (Spatial Temporal) pour être conforme au standard SQL/MM.

7. Présentation de DBManager

Mise en oeuvre

Nous allons mettre en pratique **SQL** dans le SGBD **SpatiaLite**.

Il existe de nombreux 'clients' permettant d'écrire et d'exécuter les requêtes **SQL**.

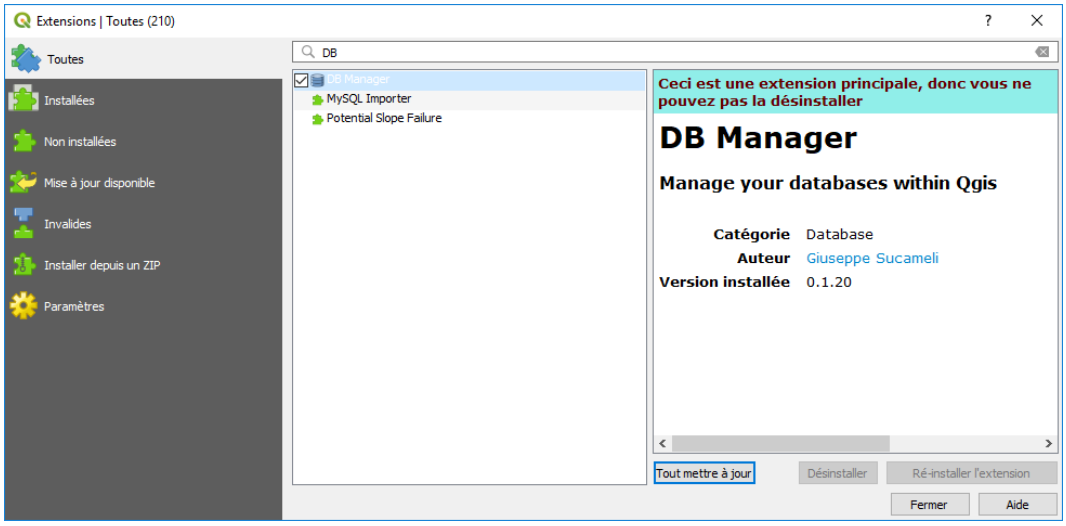
Depuis QGIS nous allons utiliser le plugin **DBManager** qui s'interface aussi bien avec **SpatiaLite, PostGIS, ou les virtuals Layer** (toute couche ouverte dans QGIS). C'est le plugin qui est porté par la communauté QGIS.

Avec **PostgreSQL**, il est possible d'utiliser PgAdmin [11]qui est le client le plus populaire de **PostGIS** et qui dispose de fonctionnalités intéressantes comme un assistant à l'écriture de requêtes SQL. Ce client est présenté en détail dans la formation '*PostgreSQL Administrer ses données*[12]'

Pour mettre en œuvre le plugin DB Manager...

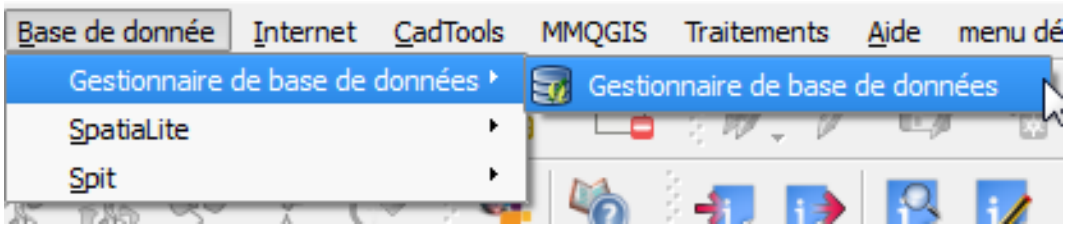
Vérifier qu'il est bien activé, sinon activer-le.

11. http://www.pgadmin.org/?lang=fr_FR
12. http://www.geoinformations.developpement-durable.gouv.fr/postgis-support-pedagogique-a3347.html



Gestionnaire d'extension de QGIS

Le plugin est alors disponible dans le menu 'bases de données' de QGIS, ou dans la barre d'outils base de données avec le bouton 



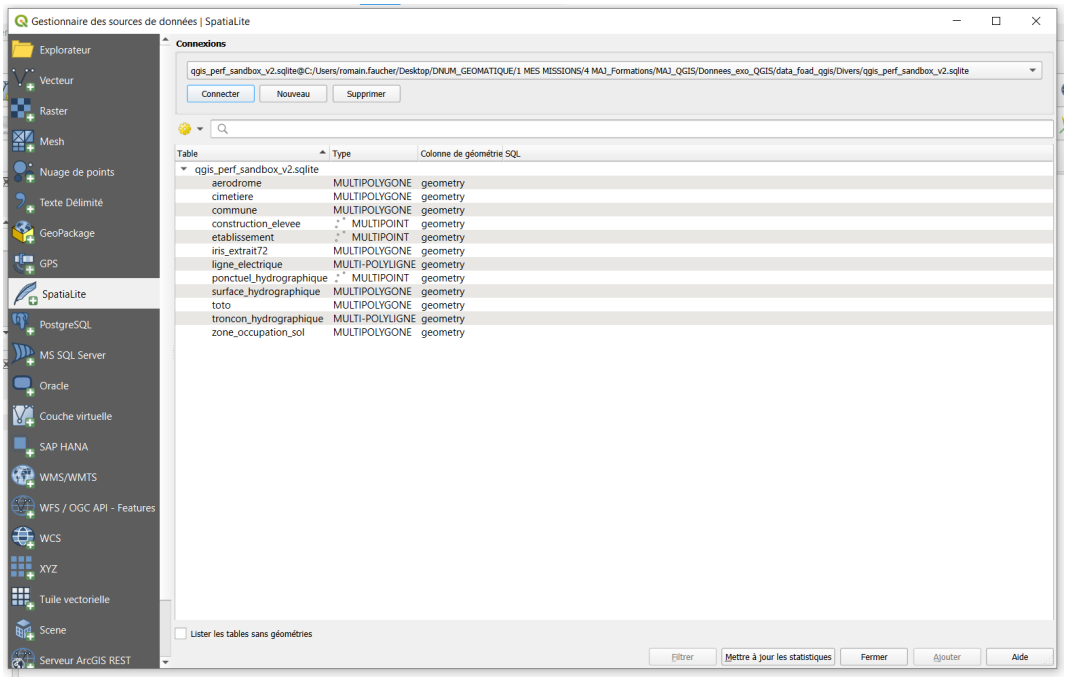
Menu Base de données

Il est possible de créer une nouvelle base de données **spatialite** à partir de QGIS en exportant une première couche (clic droit, enregistrer sous) puis de **choisir le format spatialite (et pas sqlite).**

Pour se connecter la première fois à une base de données *existante* (spatialite ou postgis) dans **DBManager**, il est nécessaire de le faire par l'intermédiaire de QGIS.

Établir la connexion avec la base de données **QGIS_perf_sandbox_V2.sqlite** en ouvrant le gestionnaire de sources de données, puis en utilisant le bouton '**ajouter une couche spatialite**' 

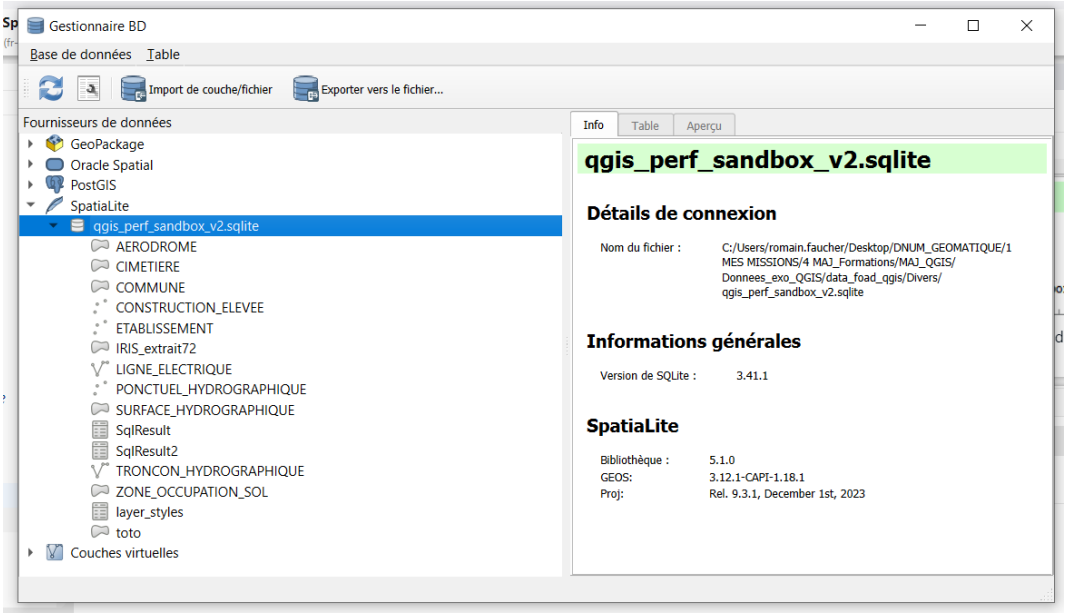
et désigner le fichier **QGIS_perf_sandbox_V2.sqlite** fourni dans le jeux de données (répertoire Divers). Puis connecter... vous devez voir apparaître cette boîte de dialogue :



Connexion sandbox

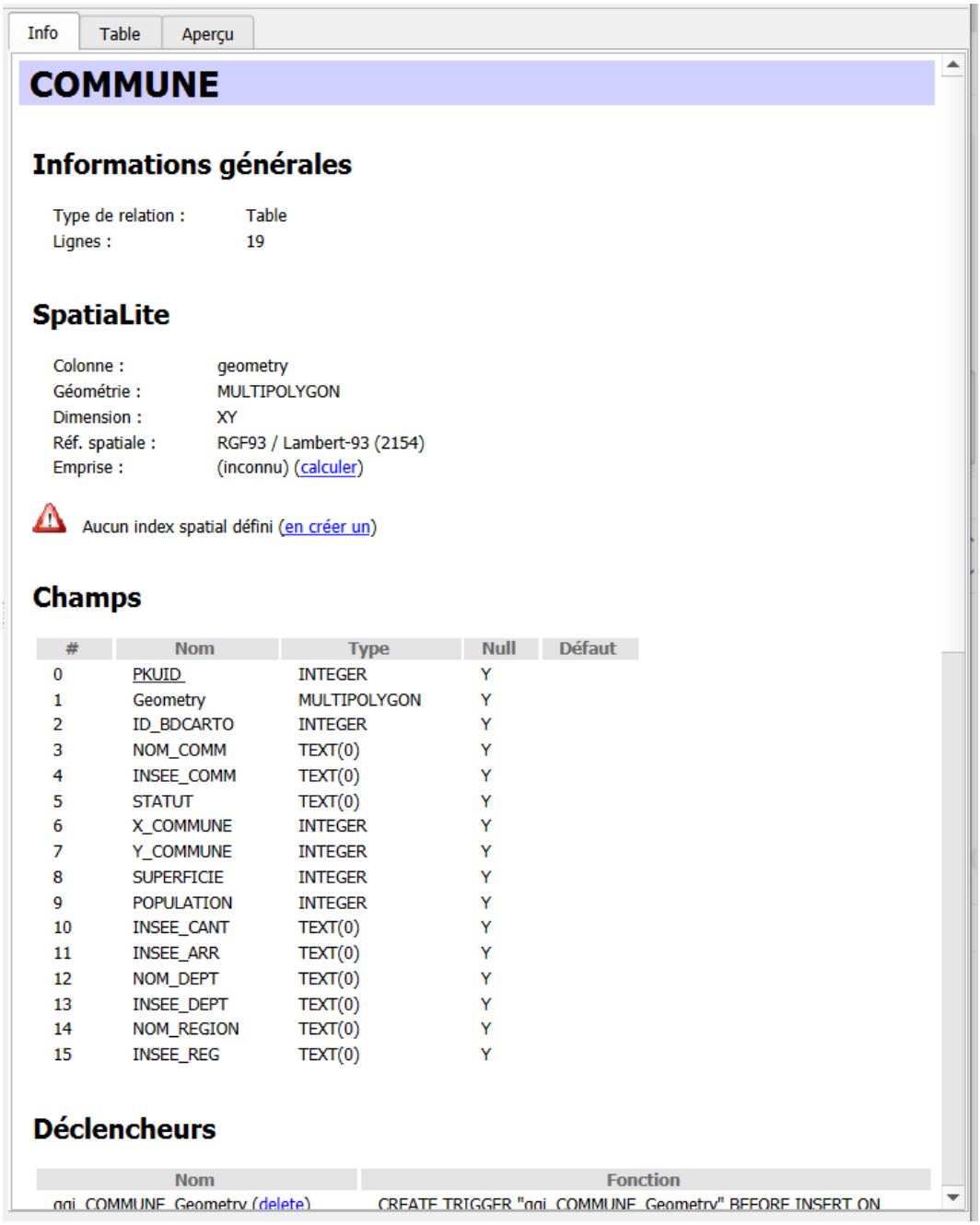
En lançant **DB Manager** vous devez maintenant pouvoir vous connecter à cette base.

Nb : Une base de données de type **PostGIS** peut être protégée par mot de passe, dans ce cas il faut le saisir dans la fenêtre qui apparaît pour cela.



DbManager

L'onglet **info** fournit les informations sur les tables



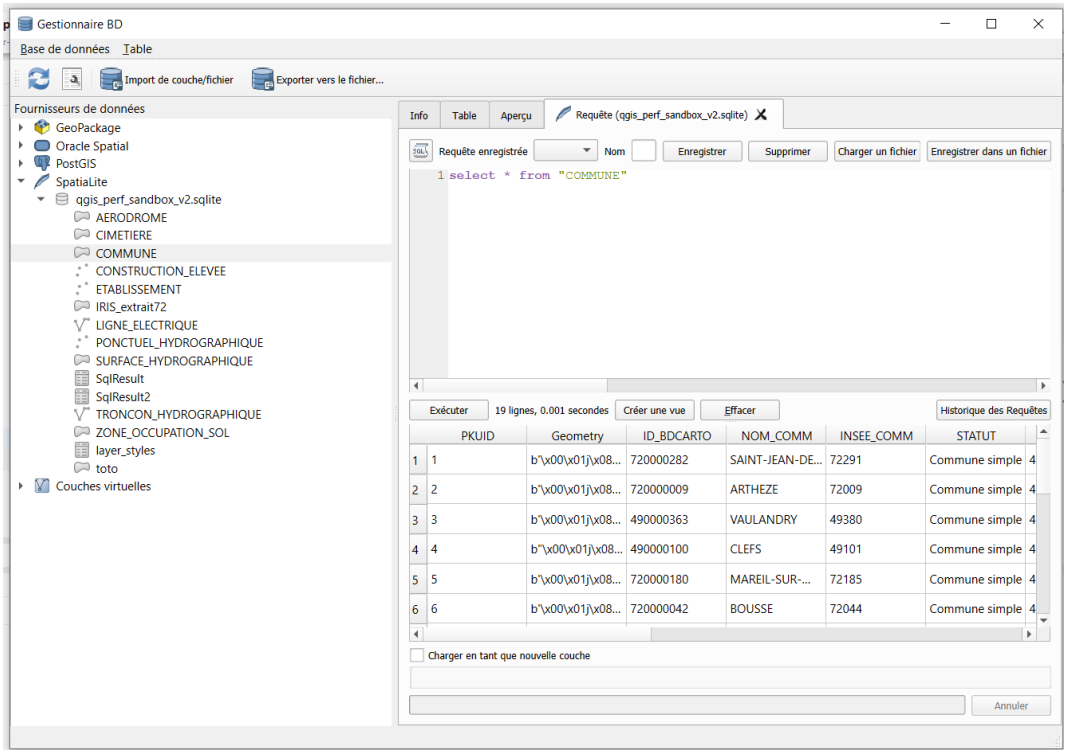
DBManager Informations

On peut par exemple lire que la table **COMMUNE** contient 19 enregistrements (lignes), qu'il y a une colonne de géométrie contenant des objets 'MULTIPOLYGON', que la projection est Lambert 93 et qu'il n'y a pas d'index spatial (aucun index spatial défini...nous verrons ce que cela signifie concrètement plus tard).

L'onglet '**table**' fournit une vision des données de la table et l'onglet aperçu une visualisation de la géométrie.

Le bouton '**Fenêtre SQL**' ouvre un nouvel onglet dans lequel nous allons pouvoir exécuter des ordres SQL.

Exemple : sélectionner tous les objets de la table commune :



Sélection avec DBManager

Chargement d'un résultat comme couche *Complément*

Le résultat d'une requête SQL peut-être chargé comme une nouvelle couche dans QGIS en cochant la case **'Charger en tant que nouvelle couche'.**

Il faut préciser une colonne avec des valeurs entières et uniques. Ce doit être un champ de type INTEGER, par exemple pour la table **COMMUNE** se pourrait être **ID_BDCARTO.**

Si on ne dispose pas d'un tel champ on peut ne pas cocher la case.

La colonne géométrique est en général la colonne de nom Geometry.

8. Exercice : Exercice 6 : sélections SQL avec DBManager

Réaliser ses premières requêtes SQL avec DBManager sous QGIS

Envoi de votre réponse aux tuteurs :

Envoyez vos requêtes sous la forme suivante :

Q1 : SELECT ...

Q2 : SELECT...

...

dans la boîte mel de l'équipe de formation qui vous a été indiquée dans votre protocole individuel de formation.

En utilisant les tables de '**QGIS_perf_sandbox_V2.sqlite**' ou avec DBManager rédiger les 8 requêtes répondant aux questions suivantes :

Question 1

[solution n°1 p. 42]

Q1 : sélectionner tous les IRIS (table **IRIS_extrait72)** de la commune de la FLECHE (colonne **Nom_Com**)

Indice :

Utiliser la table **iris_extrait72** et mettre une condition après la clause **WHERE** permettant d'indiquer qu'on se limite à la commune de la Flèche.

Question 2

[solution n°2 p. 42]

Q2 : sélectionner les communes du département de la Sarthe de plus de 1500 habitants en affichant un tableau avec les noms de communes et leur population.

Indice :

Utiliser la table commune, sélectionner les champs demandés (nom des communes et population) dans la clause **SELECT**.

Mettre deux conditions 'département de la Sarthe' ET 'population de plus de 1500 habitants' dans la clause **WHERE**.

Question 3

[solution n°3 p. 42]

Q3 : sélectionner les communes de la table **COMMUNE** dont le statut n'est pas chef-lieu de canton et afficher les colonnes **NOM_COMM** en lui donnant comme alias **NOM** et les colonnes, **STATUT, POPULATION et SUPERFICIE**

Indice :

traduire le "n 'est pas" par l'utilisation de **NOT**.

Question 4

[solution n°4 p. 42]

Q4 : Établir la liste des noms des tronçons comportant le nom 'ruisseau' dans la colonne **TOPONYME** de la table **TRONCON_HYDROGRAPHIQUE**

Indice :

Utiliser la table **troncon_hydrographique**. On pourra utiliser **LIKE** pour indiquer que le nom de tronçon doit contenir la chaîne '**ruisseau**'.

Question 5

[solution n°5 p. 42]

Q5 : à partir de la table **COMMUNE**, calculer pour chaque département ; la population totale, la densité moyenne de population des communes = moyenne(population commune /superficie commune) arrondie à deux décimales, la population de la commune la plus peuplée et celle de la moins peuplée, la superficie moyenne des communes.

Indice :

Le résultat doit être :

	NOM_DEPT	population_dept	densite_moy_communes	pop_max_commune	pop_min_commune	surface_moy_commune
1	MAINE-ET-LOIRE	1500	0.29	900	300	2036.0
2	SARTHE	28200	0.57	15400	400	2255.19

exo6 - question 5

on cherche des sommes, moyennes,...par département il faut donc utiliser un **GROUP BY** (agrégation) avec comme critère le nom de département (**NOM_DEPT**).

Qui dit agrégation implique automatiquement l'utilisation de fonctions d'agrégation...On utilisera les fonctions d'agrégation donnant la somme, la moyenne, le maximum et le minimum.

Question 6

[solution n°6 p. 42]

Q6 : quels sont les surfaces (en km2) et périmètres (en km), arrondis à deux chiffres après la virgule, des communes du département de la Sarthe ?

Indice :

trouver la fonction géométrique qui renvoie une aire, et celle qui renvoie un périmètre. Ces fonctions ne prennent pas de paramètres d'unités, il faut donc faire la conversion soi-même par une division.

Question 7

[solution n°7 p. 42]

Q7 : sélectionner le nombre de tronçons de la 'rivière le loir', par classe de largeur (colonne **LARGEUR**)

Indice :

Il faut 'compter' le nombre de tronçons, donc utiliser une agrégation avec la fonction d'agrégation qui permet de compter.

il y a 3 tronçons dans la classe 0 à 15 métres et 29 dans la classe plus de 50 mètres.

Question 8

[solution n°8 p. 42]

Q8 : quelle est la longueur de la 'rivière le loir' par type de largeur sur ce jeu de données ?

Indice :

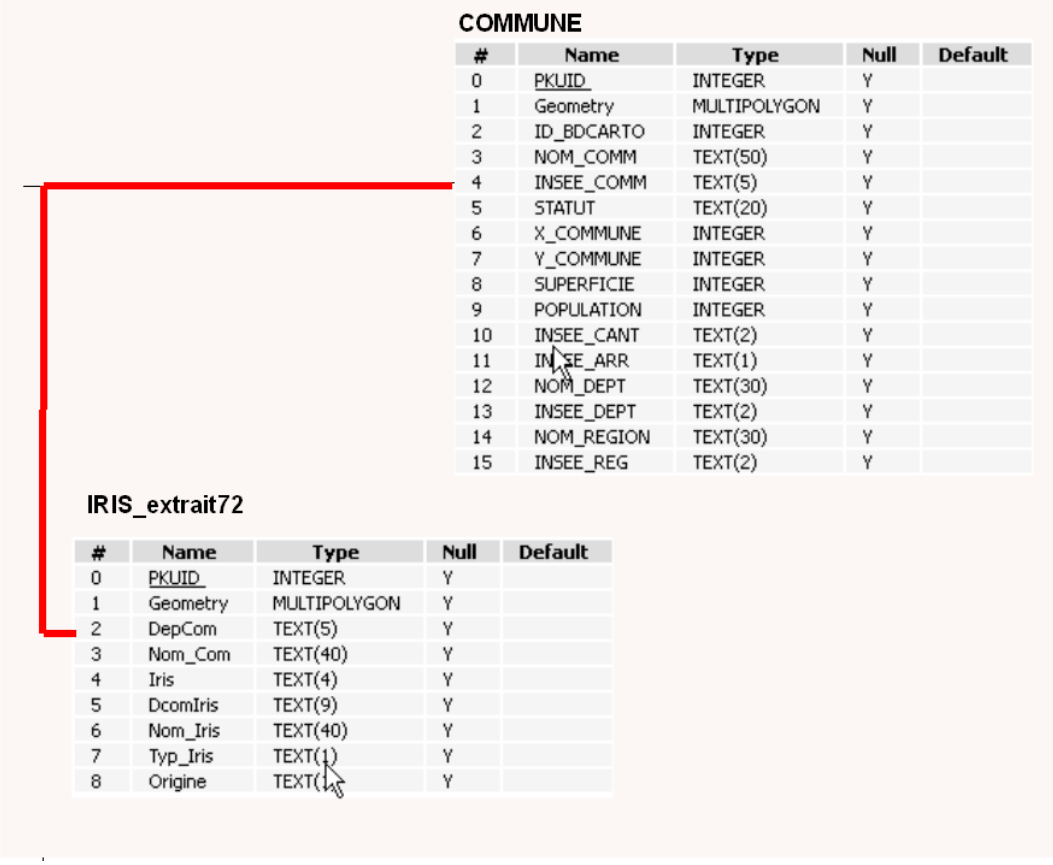
Il faut partir de la requête précédente et ajouter une colonne qui va calculer la somme de la longueur des tronçons... on utilisera la fonction st_length qui donne la longueur d'un objet linéaire.

9. Les jointures attributaires

Une jointure permet de mettre en relation deux (ou plus) tables afin de combiner leurs colonnes.

Il existe plusieurs natures de jointure, mais les cas simples se font en imposant l'égalité d'une valeur d'une colonne d'une table à une colonne d'une autre table.

Exemple : Table **IRIS** (extrait) et table des **COMMUNES**



Exemple de jointure attributaire

On recherche pour chaque **IRIS** le nom de la commune d'appartenance. Ce nom n'est pas dans la table **IRIS_extrait72**, mais il existe dans la table **COMMUNE**.

La solution est donc d'établir un lien (une jointure) entre les deux tables afin que pour chaque enregistrement de la table **IRIS_extrait72** on retrouve le nom de la commune dans la table **COMMUNE**.

Une analyse des tables **IRIS_extrait72 et COMMUNE** permet de voir que **DepCom** représente le **N° INSEE** de la commune d'appartenance dans la table **IRIS_extrait72**.

On retrouve ce **N°INSEE** dans le champ **INSEE_COMM** de la table **COMMUNE**.

Le lien peut donc s'établir par égalité des ces colonnes ce que l'on écrira :

```
IRIS_extrait72.DepCom = COMMUNE.INSEE_COMM
```

C'est ce que l'on appelle **la condition de jointure**.

Remarque

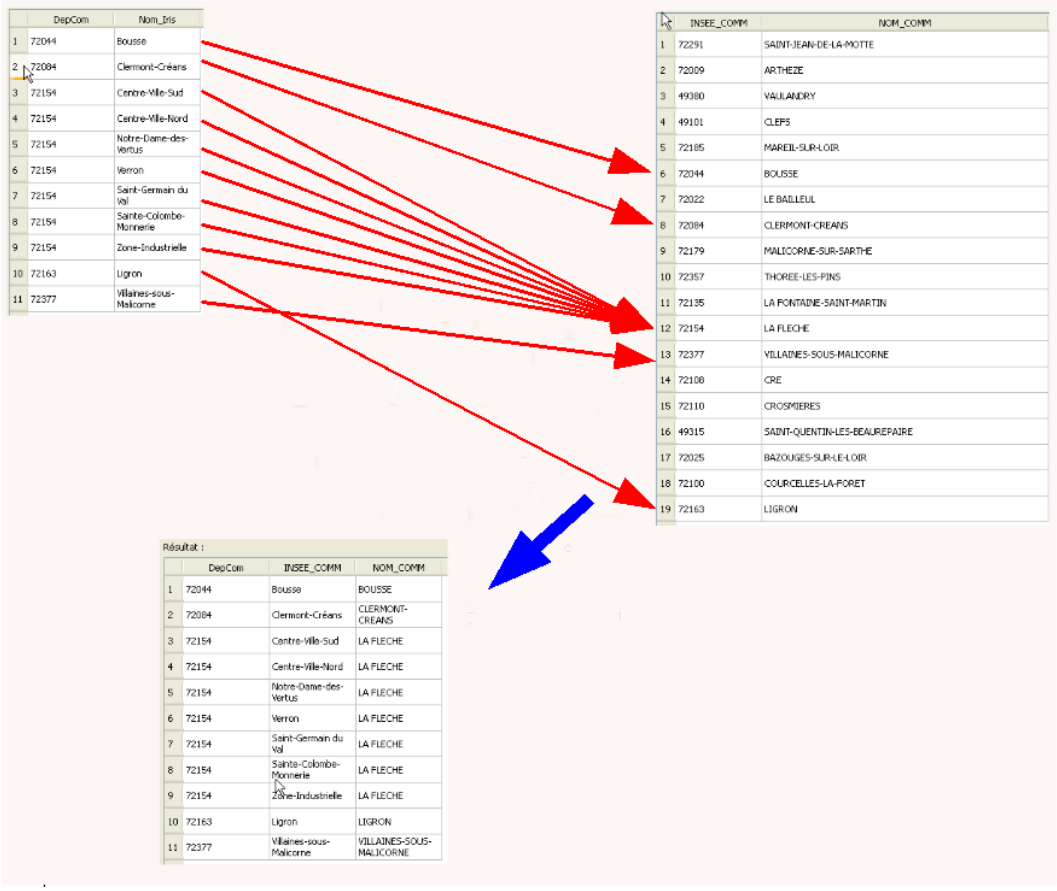
Lorsqu'on utilise plusieurs tables, il devient nécessaire s'il y a risque d'ambiguïté de préciser le nom de la table devant le nom des colonnes sous la forme **NomTable.NomColonne** d'où par exemple COMMUNE.INSEE_COMM.

Lorsqu'on réalise une jointure attributaire entre deux tables (deux noms de tables après le from) il faut retenir qu'en général **il faut** une condition de jointure qui sera pour nous une égalité de champ.

La requête pourrait être la suivante (on ne retient que certains champs) :

```
SELECT DepCom, Nom_Iris, insee_comm, nom_comm FROM iris_extrait72, commune WHERE DepCom = insee_comm
```

Ici il n'y a pas d'ambiguïté sur les noms de colonnes et on peut ne pas utiliser la notation NomTable.NomColonne



Principe de jointure attributaire

La table résultat est une table qui a le même nombre d'enregistrements que la table **IRIS_extrait72** dans laquelle on récupère le nom des communes de la table **COMMUNE**.

Complément

On peut également utiliser une syntaxe normalisée qui est dans ce cas strictement équivalente :

```
SELECT <colonnes> FROM <table1> JOIN <table2> ON <condition de jointure>
```

dans notre exemple cela donne :

```
SELECT DepCom, Nom_Iris, insee_comm, nom_comm FROM iris_extrait72 JOIN commune ON iris_extrait72.DepCom = INSEE_COMM
```

SQL autorise beaucoup de subtilité dans les types de jointures, on pourra par exemple consulter Le SQL de A à Z sur les jointures[13].

10. Les jointures spatiales

Il n'est pas toujours possible de réaliser une jointure attributaire s'il n'y a pas de colonne commune entre deux tables. Dans le cas de tables ayant chacune un champ géométrique il est possible de réaliser des **jointures spatiales**.

La jointure spatiale utilisera une fonction spatiale (voir ci-dessous) dans la clause WHERE d'une requête SQL :

exemple :

```
SELECT * FROM tableA, tableB WHERE ST_Intersects(tableA.geometry, tableB.geometry)
```

13. http://sqlpro.developpez.com/cours/sqlaz/jointures/

Les relations spatiales

Les prédicats spatiaux de l'OGC sont représentés dans le tableau suivant :

Prédicat	Types d'objets (P point, L polyligne, S polygone)	Conditions
Equals	**Tous**	A **Equals** B si les objets sont géométriquement identiques : relation topologique (le nombre de sommets des 2 objets peut être différent)
Disjoint	**Tous**	A **Disjoint** B si les objets n'ont aucun point commun (intérieur et limite). (Inverse de Intersects)
Touches	**S/S, L/S, L/L, P/S, P/L**	A **Touches** B si les limites des objets ont au moins un point commun et si les intérieurs n'ont pas de point commun (non applicable à P/P)
Crosses	**P/S, P/L, L/S, L/L**	A **Crosses** B si les intérieurs ont au moins un point commun mais pas tous et si la dimension de l'intersection des intérieurs est inférieure à la dimension maximale des objets A et B (non applicable à P/P, S/S)
Within	**Tous**	A **Within** B si tout point de A est un point de B et si les intérieurs ont au moins un point commun (aucun point de A n'est à l'extérieur de B). (Inverse de Contains)
Contains	**Tous**	A **Contains** B si tout point de B est un point de A et si les intérieurs ont au moins un point commun (aucun point de B n'est à l'extérieur de A). (Inverse de Within)
Overlaps	**S/S, L/L, P/P**	A **Overlaps** B si à la fois : - A et B ont la même dimension (non applicable à P/L, P/S, L/S) - A et B ont des points en commun, mais pas tous - L'intersection des intérieurs de A et de B a la même dimension que A et B
Intersects	**Tous**	A **Intersects** B si A et B ont au moins un point commun (intérieur ou limite) (Inverse de Disjoint)
Covers (*)	**Tous**	A **Covers** B si aucun point de B n'est à l'extérieur de A (tout point de B est un point de A) (à comparer à Contains)
CoveredBy (*)	**Tous**	A **CoveredBy** B si aucun point de A n'est à l'extérieur de B (tout point de A est un point de B) (à comparer à Within)
Relate (A,B, DE-9IM Pattern Matrix)	**Tous**	Exprime la relation spatiale de A et de B à l'aide de la matrice modèle DE-9IM Permet la généralisation des prédicats spatiaux aux 98 relations topologiques. Ex : **Relate** (A, B, « 0F1FF0102 ») <=> A Intersects B

(*) : Prédicats non définis par la norme OGC, présents dans Oracle spatial, JTS, GEOS, PostGis

Prédicats de l'OGC

Ils sont disponibles sous formes de fonctions spatiales qui renvoient VRAI (1) ou FAUX (0) :

ST_Equals(geometry A, geometry B) retourne vrai si les géométries sont de même type et ont les mêmes coordonnées.

ST_Intersects(geometry A, geometry B) retourne vrai s'il y a au moins un point commun.

ST_Disjoint(geometry A, geometry B) retourne vrai s'il n'y a aucun point commun (équivalent à n'intersecte pas ou NOT ST_Intersect, qu'il est préférable d'utiliser pour des questions de performance liée aux possibilités d'indexation spatiale)

ST_Crosses(geometry A, geometry B) retourne vrai si le résultat de l'intersection des géométries est de dimension immédiatement inférieure à la plus grande des dimensions des objets (ex : si A est un polygone et B une ligne, la dimension de l'intersection doit être une ligne) ET que le résultat de l'intersection est à l'intérieur des deux géométries.

ST_Overlaps(geometry A, geometry B) retourne vrai si les deux géométries sont de même dimension et que l'intersection est de même dimension mais de géométrie différente (renvoi faux si les deux géométries sont identiques).

ST_Touches(geometry A, geometry B) retourne vrai si les contours s'intersectent ou si un seul des intérieurs intersecte le contour de l'autre.

ST_Within(geometry A, geometry B) retourne vrai si le premier objet est complétement dans le deuxième.

ST_Contains(geometry A, geometry B) retourne vrai si le deuxième objet est complétement dans le premier.

Les fonctions suivantes sont également intéressantes :

ST_Dwithin(geometry A, geometry B, distance) qui retourne vrai si la distance la plus courte entre A et B est inférieure ou égale à distance.

ST_Distance(geometry A, geometry B) qui calcule la distance la plus courte entre deux géométries.

11. Exercice : Exercice 7 : Requêtes et fonctions spatiales

Réaliser des requêtes et fonctions spatiales

Envoi de votre réponse aux tuteurs :

Envoyez vos requêtes sous la forme suivante :

Q1 : SELECT ...

Q2 : SELECT...

...

dans la boîte mel de l'équipe de formation qui vous a été indiquée dans votre protocole individuel de formation.

En utilisant les tables de 'QGIS_perf_sandbox.sqlite' réaliser les 4 requêtes suivantes

Question 1

[solution n°9 p. 42]

Q1 : quels sont les ponctuels hydrographiques de la commune de La Flèche ?

Indice :

On utilisera les tables **PONCTUEL_HYDROGRAPHIQUE** et **COMMUNE**... trouver la relation géométrique entre **PONCTUEL_HYDROGRAPHIQUE.Geometry** et **COMMUNE.Geometry.**

Question 2

[solution n°10 p. 42]

Q2 : quelle est la longueur de la 'rivière le loir' dans chacune des communes intersectées par le cours d'eau ?

résultat à obtenir :

	NOM_COMM	TOPONYME	longueur
1	BAZOUGES-SUR-LE-LOIR	rivière le loir	2251.34
2	CLERMONT-CREANS	rivière le loir	1351.55
3	CRE	rivière le loir	1034.7
4	LA FLECHE	rivière le loir	14155.64
5	MAREIL-SUR-LOIR	rivière le loir	1079.16
6	THOREE-LES-PINS	rivière le loir	283.56

résultat exo7 question 2

Indice :

Chaque commune peut contenir plusieurs tronçons, il faut donc calculer la somme des longueurs des tronçons pour chaque commune... donc utiliser un **GROUP BY** et une fonction **sum()**. On utilisera la fonction **ST_Length()** pour obtenir la longueur de chaque tronçon. Il faut également tenir compte que certains tronçons sont à cheval sur plusieurs communes, et donc ne prendre en compte que la longueur des tronçons qui sont à l'intérieur de chaque commune pour ce faire on utilisera st_intersection(a.geom, b.geom) qui permet de récupérer la géométrie de l'objet a qui intersecte celle de l'objet b.

Question 3

[solution n°11 p. 42]

Q3 : sélectionner les **'PONCTUELS HYDROGRAPHIQUES**' qui sont à moins de 5 km d'un établissement d'enseignement (couche **ETABLISSEMENT**)

Indice :

On pourra utiliser une fonction **st_distance()** ou une fonction **st_buffer()** associée à un opérateur de type **st_contains()** ou **st_intersects()**.

Question 4

[solution n°12 p. 42]

Q4 : Quel est l'établissement le plus proche du centroide de la commune de la Flèche?

On utilisera les coordonnées X_COMMUNE et Y_COMMUNE et la fonction **st_makepoint()** ou **Makepoint()** sous spatialite ou encore **st_point()** qui est un alias de **Makepoint()**. Le SRID (Identifiant du Système de Référence Spatial) est 2154, mais on pourra le cas échéant généraliser la requête à tout SRID en utilisant la fonction **srid()** qui récupère le srid d'une géométrie.

Indice :

utiliser la fonction **distance()**, **ORDER BY** pour trier et **LIMIT 1** pour prendre le 1er objet renvoyé.

Une requête de type

```
SELECT nom_comm, srid(Geometry) AS SRID, MakePoint(X_COMMUNE, Y_COMMUNE, srid(Geometry)) AS Geometry FROM commune WHERE commune.nom_comm = 'LA FLECHE'
```

retourne des points au centroide calculé à partir des coordonnées X_COMMUNE, Y COMMUNE.

Spatialite

1. Gérer les bases et les tables

Utiliser Spatialite sous QGIS

SpatiaLite est l'extension spatiale de Sqlite. Il est conforme à la norme **OGC-SFS**[14][15] (Open Geospatial Consortium - Simple Feature SQL) et ouvre la porte à la réalisation d'analyses spatiales au-delà des fonctions natives de QGIS.

SpatiaLite est utilisé par QGIS pour stocker ses propres informations. Il est donc disponible de façon sous-jacente lorsqu'on installe QGIS.

Remarque

Le plug-in **'édition hors connexion**[16]**'** permet de gérer la synchronisation avec une base **SpatiaLite** (offline.sqliter) embarquée.

Il est donc possible d'envisager des utilisations avec saisie terrain sous SpatiaLite puis synchronisation au retour avec une base partagée centrale (sous PostGIS par exemple).

Nous allons utiliser spatialite sous QGIS avec **DBManager**.

Spatialite ou Geopackage comme base de données 'embarquée' ? *Complément*

Spatialite ne permet pas de stocker des fichiers raster. C'est pourquoi l'OGC (Open Geospatial Consortium) a entrepris de définir en 2014 un format de ficher dérivé de Spatialte permettant de stocker également des rasters. Tout comme Spatialite, ce format a été défini essentiellement pour être efficace sur des appareils mobiles (base 'embarquée').

Geopackage diffère de manière subtile de Spatialite, il faut cependant noter que Spatialite dispose depuis la version 4.2.0 de fonctions SQL permettant le support de Geopackage[17].

En ce qui concerne l'utilisation du SQL sur des couches vecteurs, il n'y a pas de différences importantes entre **Spatialite** et **Geopackage**.

C'est pourquoi dans la suite nous utilisons **Spatialite**, mais il serait tout à fait possible de faire les mêmes manipulation avec des fichiers geopackage.

Note : Bien que nous ne voyons pas la notion de Vue[18] dans ce cours, il peut-être intéressant de connaître une différence notable entre **Spatialite** et **Geopackage** sous QGIS.

Pour **Spatialite**, avec **Dbmanager**, il est possible de transformer le résultat d'une requête SQL en vue avec le bouton '**creer vue**',

14. http://www.opengeospatial.org/standards/sfs
15. http://www.opengeospatial.org/standards/sfs
16. https://docs.qgis.org/latest/fr/docs/user_manual/plugins/core_plugins/plugins_offline_editing.html
17. https://www.gaia-gis.it/fossil/libspatialite/wiki?name=4.2.0+functions#8
18. https://fr.wikipedia.org/wiki/Vue_(base_de_donn%C3%A9es)

Mais, au moins jusqu'à la version 3.22 de QGIS, ce bouton n'est pas disponible pour les bases de données geopackage.

En réalité la création des vues avec DBManager pour les geopackages n'est pas gérée[19].

Il faut mettre à jour manuellement les tables internes `*gpkg_contents*' et '*gpkg_geometry_columns*' :

A titre d'information, voici les commandes SQL nécessaires :

---- 2.1) Si vue sans géométries

--- choisir data_type=attributes dans le référencement de la table

INSERT INTO gpkg_contents(table_name, data_type, identifier, srs_id) VALUES ('test', 'attributes', 'test',0);

---- 2.2) Si vue avec géométries

---- 2.2.1) choisir data_type=features dans le référencement de la table

INSERT INTO gpkg_contents(table_name, data_type, identifier, srs_id) VALUES ('test', 'features', 'test',0);

---- 2.2.2) et définir la géométrie dans la table des géométries

INSERT INTO gpkg_geometry_columns (table_name, column_name, geometry_type_name,srs_id,z,m) VALUES ('test','geom','POLYGON',4326,0,0);

---- 2.3) Remarques : si erreur dans la définition de la table :

---- 2.3.1) Il faut supprimer l'ancienne référence :

DELETE FROM gpkg_geometry_columns WHERE table_name = 'test';

---- 2.3.2) on insert la nouvelle référence de la table :

INSERT INTO gpkg_geometry_columns (table_name, column_name, geometry_type_name,srs_id,z,m) VALUES ('test','geom','MULTIPOLYGON',2154,0,0);

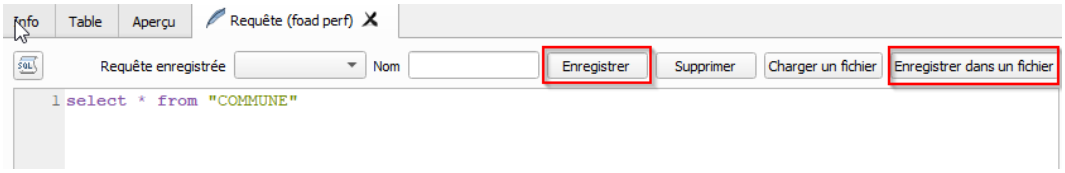
Note : Le site mygeodata.cloud propose plusieurs convertisseurs de format de données dont un convertisseur[20] de base de données Spatialite vers Geopackage.

2. Requêtes SQL sous DBManager

Gestion des requêtes

La fenêtre **SQL** sous **DBManager** permet de mémoriser et de rappeler des requêtes **SQL**,

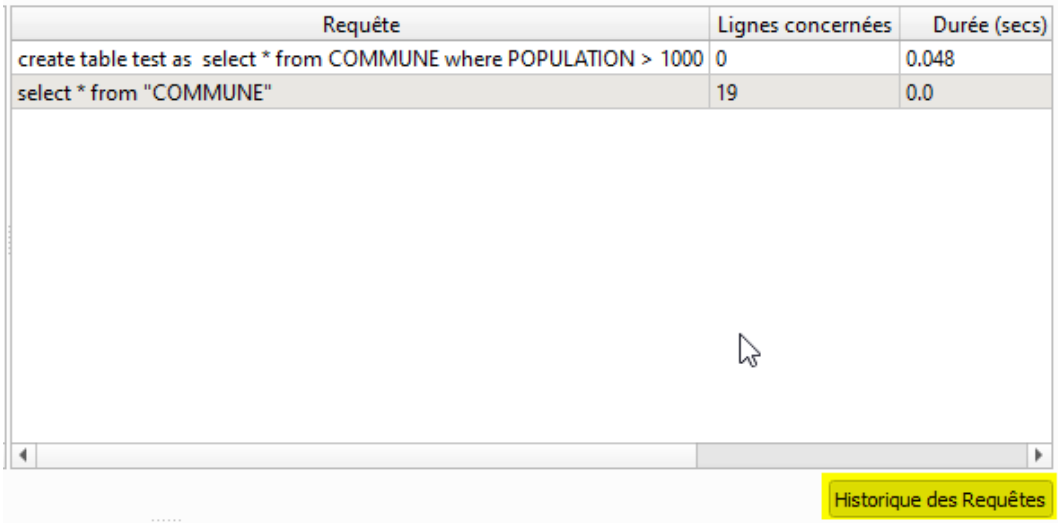
- soit dans le projet QGIS en cours avec le bouton '***Enregistrer***',
- soit sous forme d'un fichier *sql* avec le bouton '***Enregistrer dans un fichier***'



Le bouton ***Supprimer*** permet de supprimer une requête mémorisée. La liste déroulante donne accès aux requêtes mémorisées dans le projet.

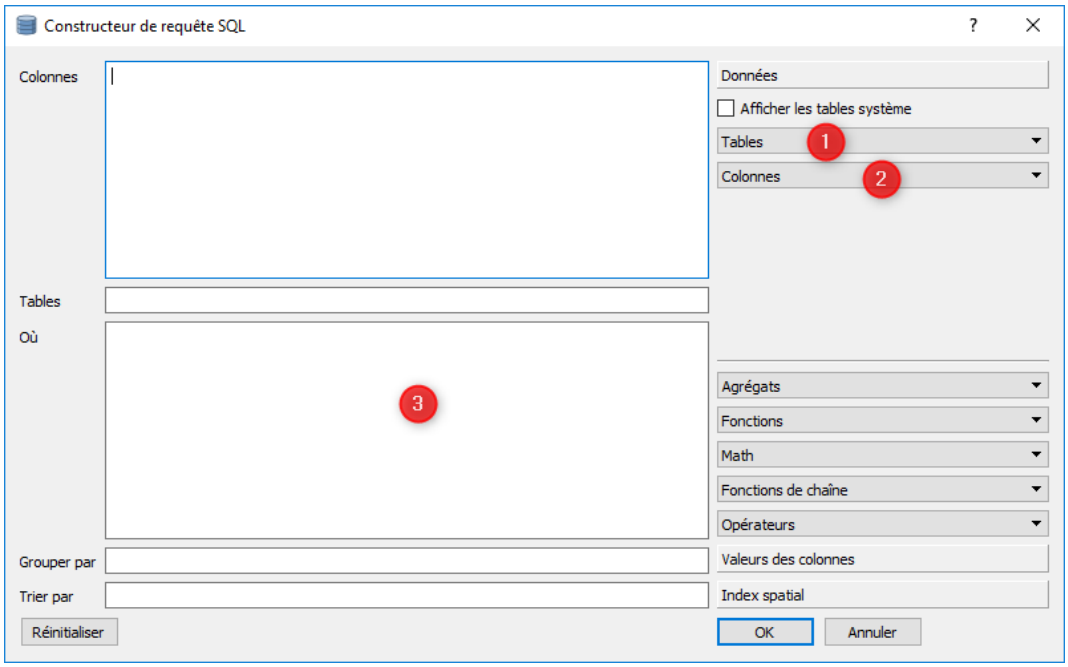
Le bouton '***Historique des Requêtes***' donne l'historique des requêtes exécutées.

19. https://github.com/qgis/QGIS/issues/25922
20. https://mygeodata.cloud/converter/spatialite-to-geopackage



L'assistant de requête SQL

L'assistant de requête est une aide pour réaliser des requêtes SQL. Son utilisation n'est pas obligatoire, mais utile pour les débutants en SQL et pour les utilisateurs connaissant MapInfo qui ne seront pas trop dépaysés. Il est lancé à l'aide du bouton 



Advance SQL

Cette boîte de dialogue permet de construire la requête SQL. La démarche est d'abord de sélectionner la ou les tables sur lesquelles on souhaite travailler (ex : ''AERODROME'', puis les colonnes de ces tables que l'on souhaite en sortie ou mettre * dans la case Columns pour choisir toutes colonnes.

Ex :

"AERODROME"."NATURE",

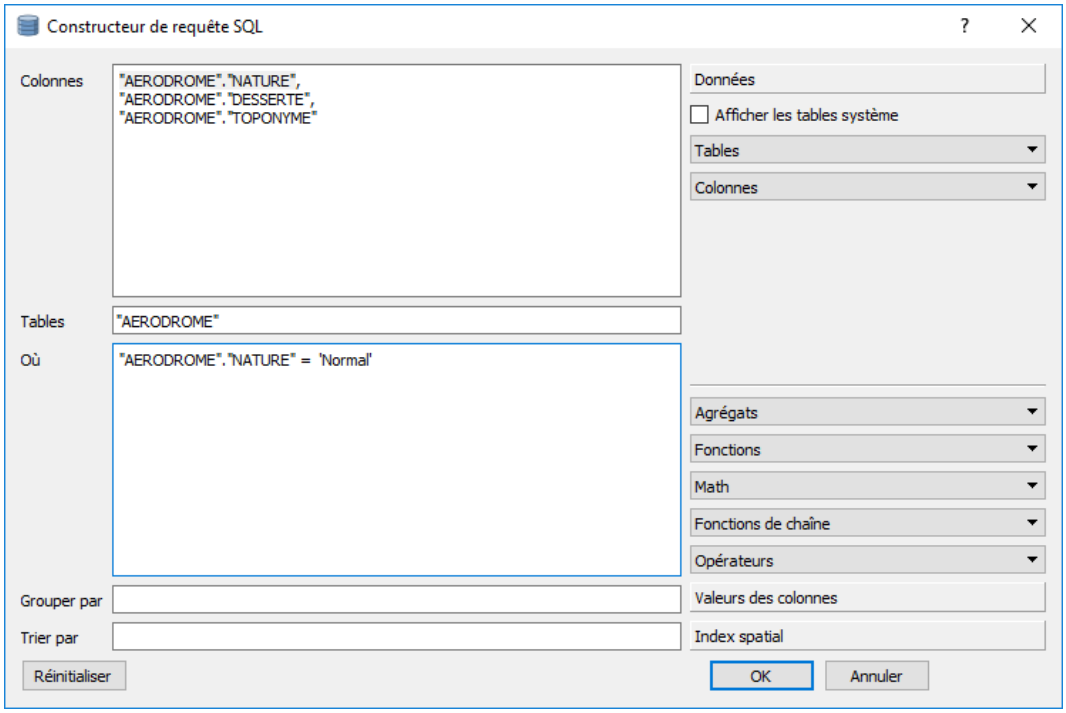
"AERODROME"."DESSERTE",

"AERODROME"."TOPONYME"

et éventuellement d'ajouter une condition (clause **where**) pour laquelle on peut utiliser les listes déroulantes à droite.

exemple :

NATURE = 'Normal'



Exemple SQL avec le requêteur avancé

Remarque

Il n'y a pas * par défaut dans le champ 'Columns' dans la boîte de dialogue pour sélectionner tous les champs, mais on peut taper * directement dans le la champ de saisie des colonnes.

on peut écrire = ou == comme opérateur d'égalité.

on ne dispose pas de = ANY (utiliser IN)

|| → est l'opérateur de concaténation (ne pas utiliser +)

Complément

GLOB : est similaire à l'opérateur LIKE (%= 0 à n caractères, _= 1 caractère) mais utilise les jokers unix (* = 0 à n caractères, ?= 1 caractère) et est sensible à la casse.

BETWEEN n'est pas disponible dans les menus déroulants, mais est utilisable.

MATCH : permet de comparer un ensemble de valeurs de ligne à un ensemble de lignes retourné par une sous-requête (usage rare). Voir ici[21] pour en savoir plus.

REGEXP : permet d'utiliser les expressions régulières ou rationnelles (voir ici[22]). Le documentation précise toutefois qu'il faut se définir sa propre fonction regexp() car il n'y en pas par défaut. L'utilisation de cet opérateur sans définir de fonction génère un message d'erreur. Très peu utile pour les besoins des services.

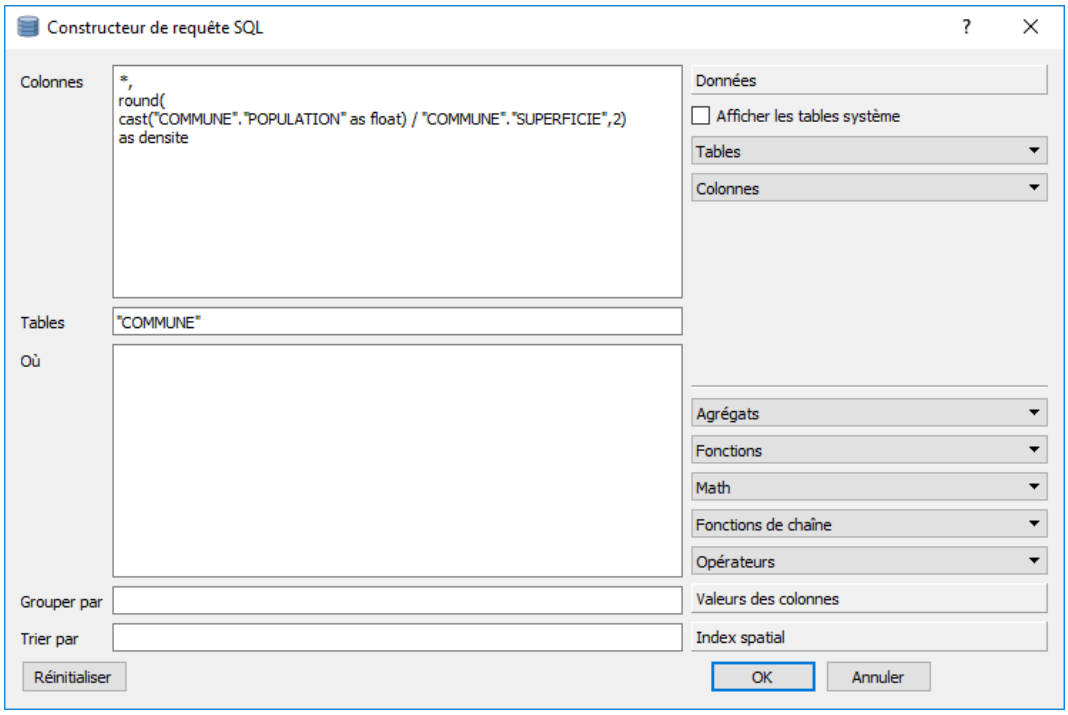
Typage des champs sous SQLite

Attention

SQLite est laxiste sur le contrôle de type des champs, plus précisément SQLite utilise un typage dynamique[23]. S'il y a non-concordance de type de données dans l'expression, SpatiaLite n'affichera rien mais le résultat sera 'Empty résultat' (résultat vide). Rappelons la fonction **cast** pour faire des comparaisons avec changement de type.

21. http://sqlpro.developpez.com/cours/sqlaz/sousrequetes/#L1.3
22. http://fr.wikipedia.org/wiki/Expression_rationnelle
23. http://fr.wikipedia.org/wiki/SQLite#Types_de_donn.C3.A9es

Affichage de toutes les communes avec calcul d'un champ supplémentaire de densité



Qspatialite, editeur SQL avancé, gestion des champs en sortie

Le choix d'une colonne dans la liste déroulante 'columns' ajoute le nom du champ précédé d'une virgule. Ce n'est pas toujours souhaitable comme dans le cas où l'on fait une division de deux champs. Il faut alors supprimer la virgule. Par défaut le nom de la table est ajouté devant le nom de la colonne.

L'appel aux fonctions comme round pour arrondir que l'on peut choisir dans la liste déroulante 'math' n'affiche pas la syntaxe de la fonction.

Pour une aide sur les fonctions spatiale, il est conseillé d'utiliser une description en ligne[24]. (il est possible d'obtenir la version SpatiaLite dans le menu 'a propos' de QGIS).

Pour QGIS 3.22 la version de spatialite est la 5.0.1

Pour QGIS 3.34 la version de spatialite est la 5.1.0

Pour une aide sur les fonctions de sqlite on pourra consulter ce site[25].

La validation par OK affiche alors la syntaxe en SQL. Dans notre cas :

```
SELECT *,
Round(cast("COMMUNE"."POPULATION" as float) /
"COMMUNE"."SUPERFICIE",2) as densite
FROM "COMMUNE"
```

Gestion de la géométrie dans les agrégations *Conseil*

Il est possible de faire des agrégations et des tris en utilisant les champs **group By Columns** et **Order by Columns.**

Lorsqu'on fait une agrégation sur une table géométrique par un 'group by', il est important de comprendre que cela ne traite pas automatiquement la fusion des géométries. Ainsi si on se contente de reprendre le champ geometry dans le résultat, il ne faut pas l'utiliser.

24. http://www.gaia-gis.it/gaia-sins/spatialite-sql-5.0.1.html
25. https://www.sqlite.org/lang_corefunc.html

Exemple :

si on fait un group by sur le statut des communes :

```
select geometry, asText(geometry) as WKT, statut, sum(superficie) as superficie, sum(population) as population from commune group by commune.statut
```

le résultat

	Geometry	WKT	STATUT	superficie	population
1		MULTIPOLYGON(((469609 6747069, 469560 6747119...	Chef-lieu de canton	1513	2000
2		MULTIPOLYGON(((475790 6743360, 475545 6743357...	Commune simple	33257	12300
3		MULTIPOLYGON(((473261 6730630, 473164 6730991...	Sous-préfecture	7421	15400

Group by avec géométrie

contient la colonne Geometry (la transformation en texte de la géométrie au format WKT est demandée à titre illustratif)

Cependant un affichage sous QGIS montre que la géométrie est uniquement celle de Clermont-Créans (première commune dans la table répondant au critère 'commune simple' et non la fusion des géométries de toutes les communes de Statut 'commune simple'.

D'une façon générale si on utilise un GROUP BY, toutes les colonnes en sortie sauf celle du critère de regroupement doivent faire l'objet d'une fonction d'agrégation, ce doit être également le cas pour la géométrie.

Pour obtenir la fusion des géométries il faut donc le demander explicitement avec une commande **ST_UNION**

dans notre exemple cela donne :

```
CREATE TABLE EXEMPLE AS
SELECT STATUT, ST_Union(Geometry) as geometry, sum(superficie) as superficie, sum(population) as population
FROM "COMMUNE"
GROUP BY "COMMUNE".'STATUT'
ORDER BY "COMMUNE".'STATUT'
```

(Pour créer la table EXEMPLE dans spatialite sans l'exporter dans QGIS on utilise un **create table** en amont du select).

La table est affichée comme étant uniquement attributaire.

Il faut utiliser la fonction **RecoverGeometryColumn()** pour mettre à jour les tables internes de métadonnées de spatialite :

```
SELECT RecoverGeometryColumn('EXEMPLE', 'Geometry', 2154, 'MULTIPOLYGON', 'XY')
```

ou `SELECT RecoverGeometryColumn('EXEMPLE', 'Geometry', 2154, 'POLYGON', 'XY')` si les objets sont des polygones simples (ce qui est le cas pour la couche "COMMUNE").

La table devient graphique et peut-être chargée sous QGIS pour vérification.

Sous **PostGIS** (à partir de la version 2.0) il n'est plus nécessaire d'utiliser des fonctions de mise à jour des tables internes. Le type geometry étant un type à part entière on peut écrire :

```
CREATE TABLE EXEMPLE AS
SELECT statut, st_multi(ST_Union(Geom)) :: Geometry(MULTIPOLYGON, 2154) as geom, sum(superficie) as superficie, sum(population) as population
FROM commune
```

```
GROUP BY commune.statut
ORDER BY commune.statut
```

La conversion en type 'multipolygon' avec le modificateur de type Geometry() met à jour automatiquement la vue 'geometry_columns'.

(Le résultat de ST_union étant soit un 'polygon', soit un 'multipolygon' on utilise la fonction **st_multi()** pour convertir tous les résultats en 'multipolygon').

nb : Pour les lignes il est possible de supprimer les discontinuités et éviter la constructions de multilignes avec la fonction **st_linemerge()**

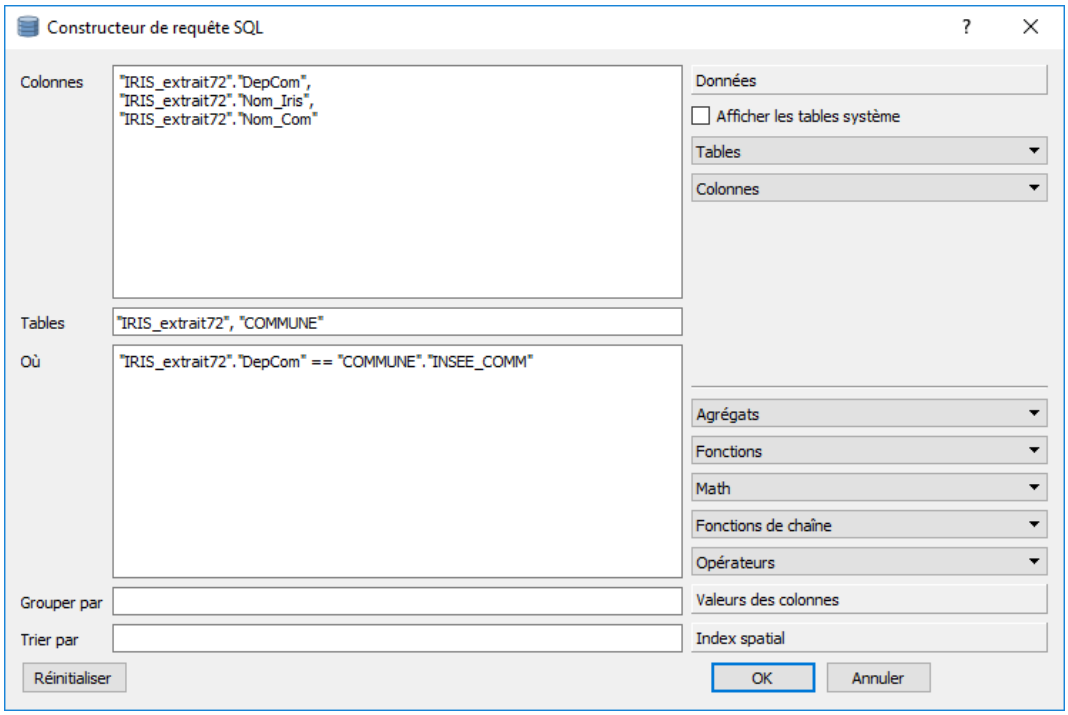
ex : `SELECT toponyme, row_number() over() as id, st_linemerge(st_union(Geom)) as geom from troncon_hydrographique where toponyme <> '' group by toponyme`

DBManager demandant un identifiant de type entier unique pour charger les couches sous QGIS, il est créé ici avec **row_number() over()**.

Ceci permet de récupérer le numéro de ligne qui est alors utilisé comme identifiant. Pour ceux qui sont intéressés, cette syntaxe utilise les possibilités avancés de SQL sur le fenêtrage[26].

3. Réaliser des jointures avec l'assistant SQL de DBmanager

Les jointures attributaires peuvent être réalisées avec l'assistant de requête en choisissant (au moins) deux tables.



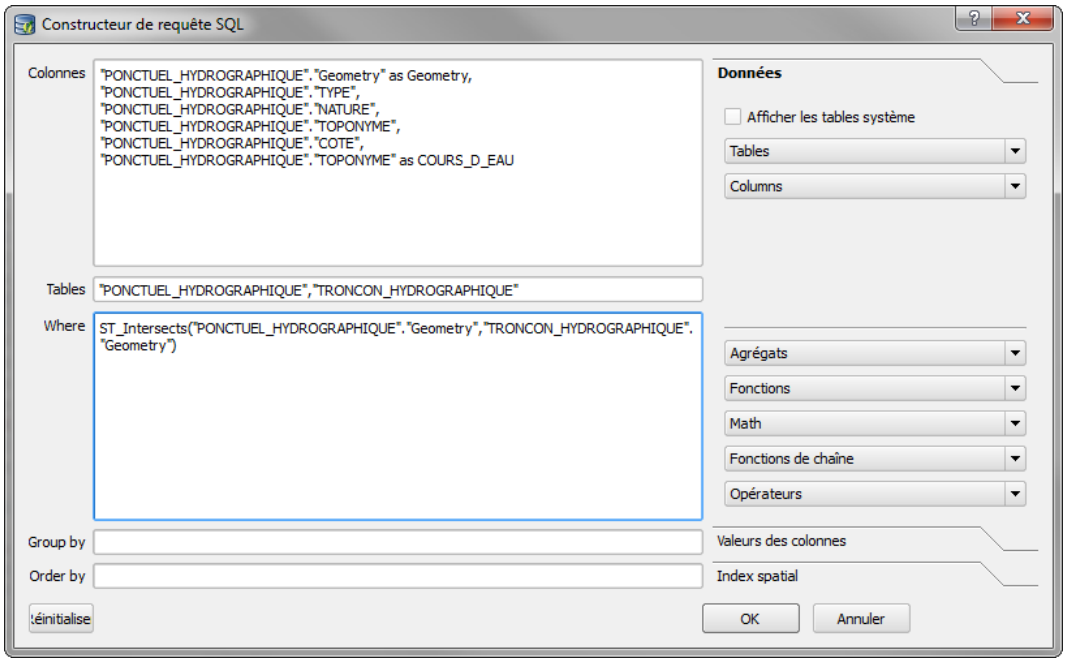
Jointures avec Qspatialite

Dans l'exemple ci-dessus, le paramétrage réalise la commande SQL :

```
SELECT "IRIS_extrait72"."DepCom",
"IRIS_extrait72"."Nom_Iris",
"IRIS_extrait72"."Nom_Com"
FROM "IRIS_extrait72", "COMMUNE"
WHERE "IRIS_extrait72"."DepCom" == "COMMUNE"."INSEE_COMM"
```

26. http://sqlpro.developpez.com/article/olap-clause-window/

De même on peut réaliser des jointures spatiales



Spatialite jointures spatiales

Remarque

Les opérateurs spatiaux sont des fonctions.

Si on souhaite utiliser le résultat comme une table spatiale sous QGIS (option Charger en tant que nouvelle couche), il est nécessaire de choisir une des colonnes de géométrie en sortie, si on indique *, il y a aura deux colonnes de géométrie dans la table résultante. Il faudra donc préciser laquelle on considère comme la source de géométrie (menu déroulant à droite de la case à cocher 'Colonnes de géométrie'.

4. Indexation et optimisation

Dans le cas d'une grosse base de données, les requêtes Sql peuvent être coûteuses en temps de calcul, a fortiori les requêtes spatiales qui utilisent la géométrie des objets.

Créer des index (spatiaux ou non) peut permettre d'améliorer les temps de traitement. Ce n'est cependant pas une recette miracle.

Dans SpatiaLite, un index spatial ne peut accélérer les calculs que dans le cas où le résultat appartient à une petite portion du jeu de données.

Quand les résultats incluent une grande partie du jeu de données, l'index spatial ne permet pas de gains de performance.

Dans un SGBD élaboré comme PostGIS le planificateur de requête choisit de façon adaptée d'utiliser ou non les index et l'index spatial (de type Gist que nous verrons plus tard) est **primordial**. La seule restriction d'utilisation est celle des tables avec de très gros objets en petit nombre (ex : tache urbaine départementale répartie en 10 périodes soit 10 enregistrements).

On trouvera quelques explications sur le principe de l'algorithme R-Tree utilisé par spatialite ici[27] et sur les index Gist ici[28]

27. http://www.gaia-gis.it/gaia-sins/spatialite-cookbook-fr/html/rtree.html
28. https://doc.postgresql.fr/14/gist.html

Clef primaire *Remarque*

Il est indispensable sous spatialite que la table dispose d'une **clef primaire** avant de créer un index spatial. Une clef primaire est une colonne qui représente un identifiant unique pour chaque enregistrement.

Les clefs primaires apparaissent soulignés dans la description des champs dans l'onglet Info de DBManager :

Champs

#	Nom	Type	Null	Défaut
0	PKUID	INTEGER	Y	
1	Geometry	MULTIPOINT	Y	
2	ID_BDCARTO	INTEGER	Y	
3	TYPE	TEXT(0)	Y	
4	NATURE	TEXT(0)	Y	
5	TOPONYME	TEXT(0)	Y	
6	COTE	INTEGER	Y	

Normalement (exemple sous PostGIS) il est possible d'ajouter une clef primaire en passant par le menu **Table -> Editer une table -> onglet contraintes.**

Malheureusement SqLite n'autorise pas (encore?) la création d'une clef primaire après coup, (pour les anglicistes voir les limites de ALTER TABLE[29]) il faut re-créer une autre table. Contacter l'assistance interne si vous êtes confronté à ce type de problèmes.

Index spatial

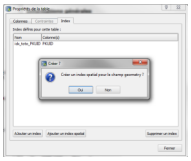
Si aucun index spatial n'existe DBManager le signale et permet de le créer directement :

SpatiaLite/Geopackage

Colonne : geometry
Géométrie : POLYGON
Dimension : XY
Réf. spatiale : RGF93 / Lambert-93 (2154)
Emprise : (inconnu) (calculer)

Aucun index spatial défini (en créer un)

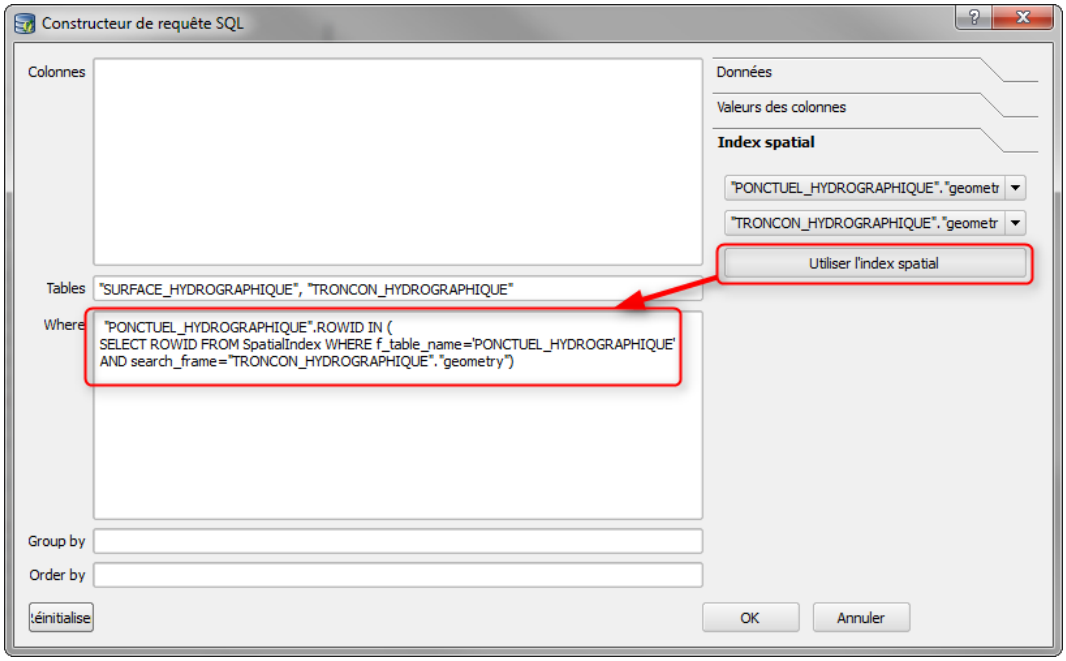
Il est également possible de passer par le menu **Table -> Modifier une table -> onglet index** -> Ajouter un index spatial



Construire un index spatial sur la table **TRONCON_HYDROGRAPHIQUE**, ainsi que sur la table **PONCTUEL_HYDROGRAPHIQUE**.

Avec l'assistant SQL, construisons maintenant une requête utilisant l'index spatial

29. http://sqlite.org/lang_altertable.html



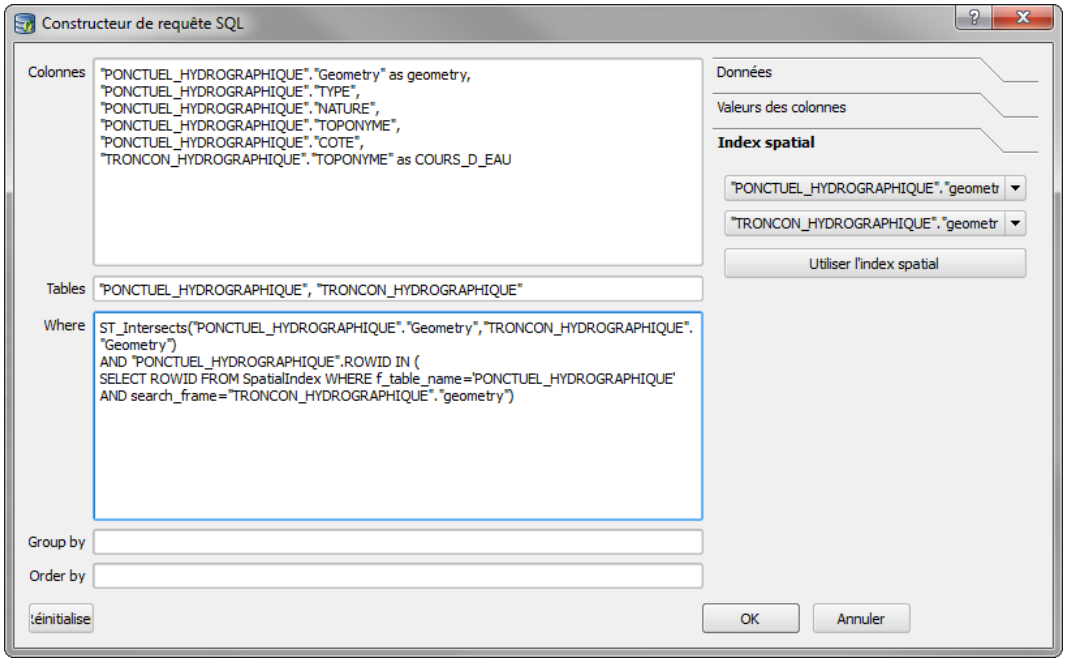
Index spatial

En appuyant sur **'Utiliser l'index spatial**' l'assistant ajoute automatiquement une syntaxe dans la clause where.

```
AND "PONCTUEL_HYDROGRAPHIQUE".ROWID IN (
SELECT ROWID FROM SpatialIndex WHERE f_table_name='PONCTUEL_HYDROGRAPHIQUE' AND search_frame="TRONCON_HYDROGRAPHIQUE"."geometry")
```

Sans rentrer trop dans les détails une **sous-requête** est ici utilisée (ordre SELECT dans le IN). Cette sous-requête utilise les colonnes `f_table_name` et `search_frame` de la table système `SpatialIndex` (cette table n'est pas interrogeable).

Une requête optimisée utilisant l'index spatial donne :



Requête complète

La différence de temps de traitement n'est pas significative dans notre cas (gain en millisecondes), Mais dans d'autres cas cette petite gymnastique qui après quelques essais n'est pas si difficile à mettre en œuvre peut faire gagner beaucoup de temps.

Complément

Les possibilités de manipulation spatiale sont très grandes... voici quelques références le livre de cuisine ![30]

Ne pas hésitez à consulter à partir de cette page [31]le 'Spatial SQL functions reference guide' qui est la liste de référence des fonctions disponibles dans la dernière version de spatialite (attention ce n'est pas forcement celle de votre version de QGIS). Pour aller plus loin, on pourra en particulier regarder avec intérêt les fonctions 'GEOS Advanced', ainsi que les fonctions 'LWGEOM'

(Sur SQL d'une façon générale on pourra consulter un cours en ligne[32] ou ce site[33] de référence en français

5. Les couches virtuelles (Virtual Layers)

QGIS dispose d'une notion de **couche virtuelle (Virtual Layers).**

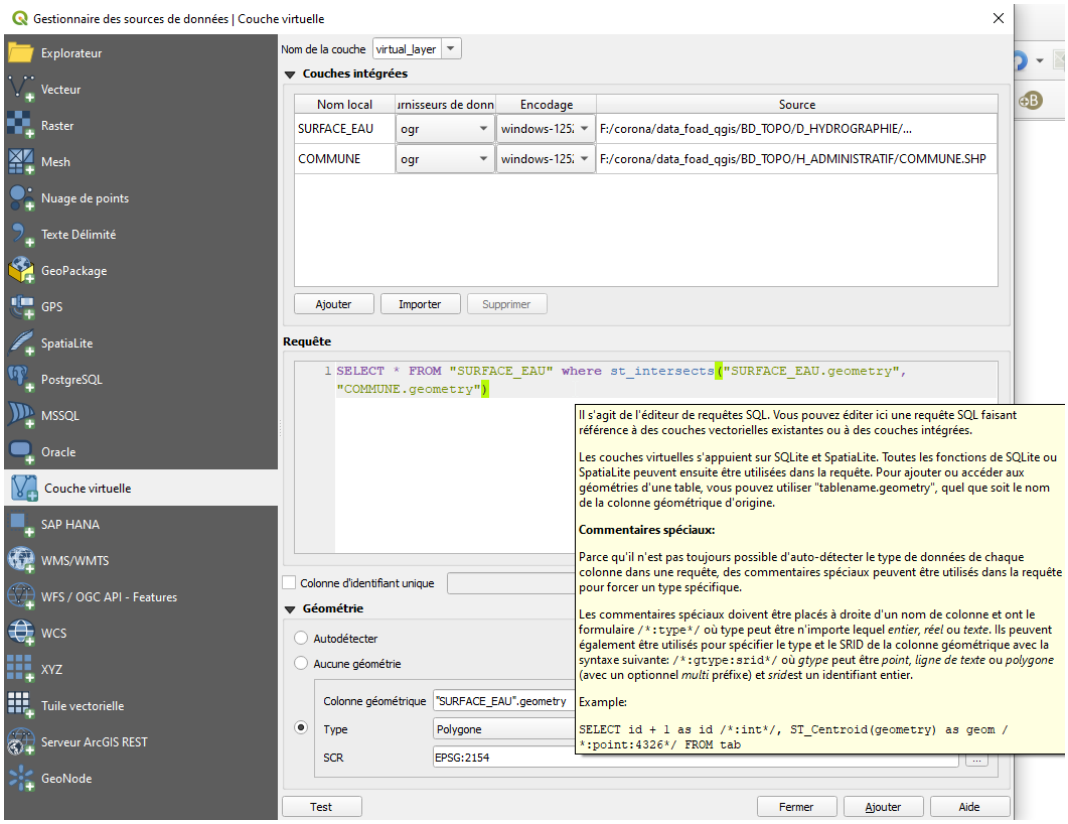
Une couche virtuelle ne contient pas de données, c'est une requête SQL qui est stockée (équivalent à une **Vue** en SQL).

De façon sous-jacente c'est le mécanisme des tables virtuelles de spatialite qui est utilisé. Les couches virtuelles sont stockées dans les fichiers projets de QGIS.

Il est possible de créer une couche virtuelle directement sous QGIS avec le bouton 

le bouton **importer** permet de choisir les couches ouvertes dans QGIS qui serviront dans la requête SQL.

Il est également possible d'ajouter n'importe quelle autre ressource avec le bouton **Ajouter**.

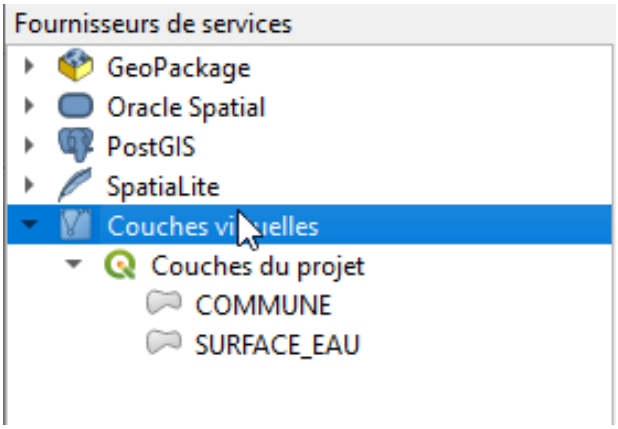


Toutefois la création de couche virtuelle explicite est réservée à des utilisateurs expérimentés.

30. http://www.gaia-gis.it/gaia-sins/spatialite-cookbook-fr/index.html
31. https://www.gaia-gis.it/fossil/libspatialite/index
32. http://www.1keydata.com/fr/sql/
33. http://sqlpro.developpez.com/

Ceux qui désirent en savoir plus peuvent consulter la documentation QGIS[34] (nous attirons l'attention sur les commentaires spéciaux[35], que l'on retrouve dans l'aide contextuelle lorsqu'on positionne le curseur dans l'espace de saisie de la requête cf.ci-dessus).

Nous allons, dans la suite, mettre en œuvre le mécanisme au travers de **DBManager** qui l'utilise pour rendre disponibles toutes les couches ouvertes dans QGIS dans le fournisseur 'Virtual Layers'.



6. Exercice : Exercice 8 : requêtes SQL avec les couches virtuelles

Réaliser des requêtes SQL en utilisant les couches virtuelles

Objectif :Réaliser des requêtes SQL directement avec les couches ouvertes dans QGIS

Question 1

[solution n°13 p. 43]

Q1 : Charger les couches suivantes dans QGIS :

BD_TOPO/I_ZONE_ACTIVITE/**PAI_SANTE.SHP**

/BD_TOPO/H_ADMINISTRATIF/**COMMUNE.SHP**

/BD_TOPO/E_BATI/**BATI_INDUSTRIEL.SHP**

/BD_TOPO/F_VEGETATION/**ZONE_VEGETATION.SHP**

Avec **DBManager**, en utilisant les virtuals layers (QGIS Layers),

créer une nouvelle couche BATI_INDUSTRIEL10 et la charger dans QGIS en sélectionnant dans la table BATI_INDUSTRIEL les 'Bâtiment industriel' (attention à la majuscule!) dont la hauteur est d'au moins 10 m

Indice :

La table résultat doit contenir 8 enregistrements.

Question 2

[solution n°14 p. 43]

Q2 : Afficher dans la fenêtre des résultats la liste des 'Forêt fermée de conifères' de la commune de La Flèche. Ne pas oubliez de mettre une condition de jointure entre les deux couches... qui devra être ici spatiale.

Le résultat doit contenir 55 enregistrements.

Question 3

[solution n°15 p. 43]

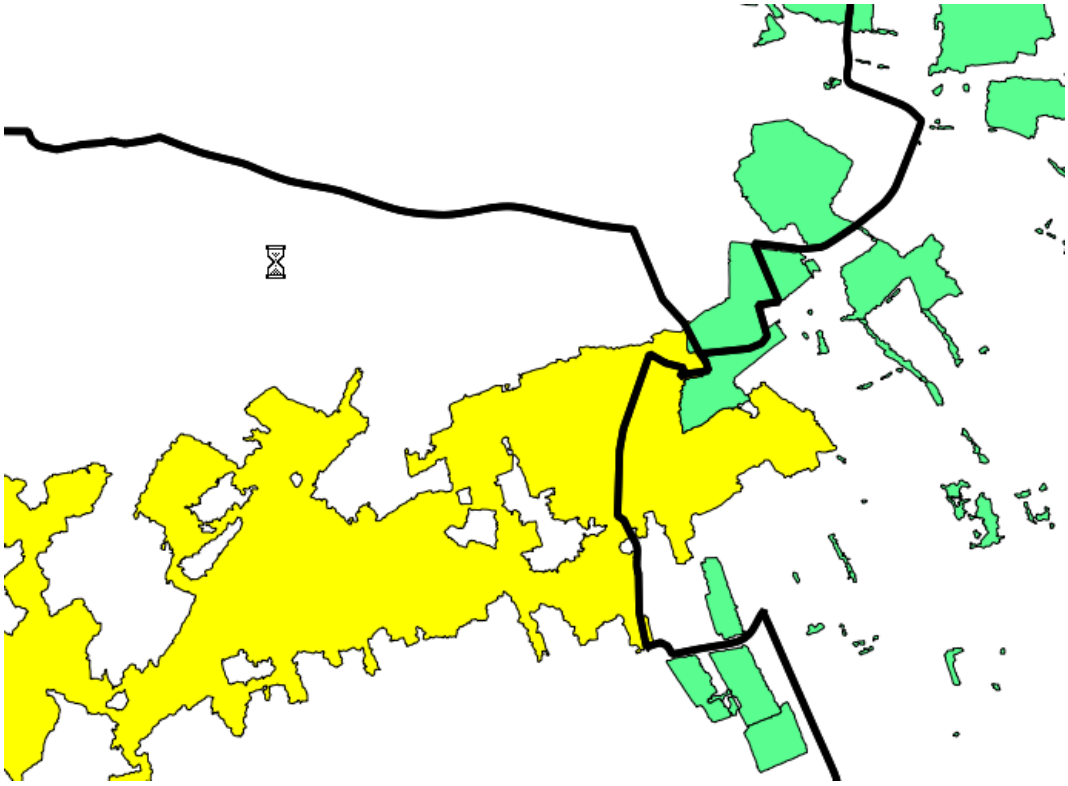
Q3 : Plus difficile...

Calculer la somme des surfaces des 'Forêt fermée de feuillus' de la commune de la Flèche en ha (1ha = 10 000 m2), en faisant attention à ne prendre en compte que les parties de surfaces des polygones réellement situées à l'intérieur de la commune.

---

34. https://docs.qgis.org/latest/fr/docs/user_manual/managing_data_source/create_layers.html#creating-virtual-layers

35. https://docs.qgis.org/latest/fr/docs/user_manual/managing_data_source/create_layers.html#special-comments



attention aux limites des polygones

Ainsi dans l'exemple ci-dessus il ne faut prendre en compte que la partie du polygone jaune qui est à l'intérieur de la commune de la Flèche... on pourra penser à la fonction st_Intersection() qui retourne un objet géométrique intersection de deux objets...le résultat est 565 ha

Question 4

[solution n°16 p. 43]

Q4 : Construire une nouvelle couche dans QGIS (non graphique) de nom ETABLIS_PLUS_PROCHE qui pour chaque établissement hospitalier de la couche PAI_SANTE donne l'identifiant (ID) de l'établissement industriel de la couche BATI_INDUSTRIEL le plus proche, ainsi que la distance

ATTENTION : Cet exercice fait appel pour sa solution à l'utilisation d'une sous-requête[36], il peut être jugé complexe, dans ce cas essayez de comprendre la solution. Son objectif est de montrer la puissance du SQL pour la résolution de problèmes parfois complexes...

Le résultat est :

	ID	ID	distance
1	PAISANTE0000000244723937	BATIMENT0000000214067387	124,649
2	PAISANTE0000000244723936	BATIMENT0000000214076737	150,308

exo8 Q4 résultat

Indice :

Il est conseillé de décomposer un problème complexe en problèmes plus simples pour arriver à la solution...

On pourra dans un premier temps construire une table qui donne les distances de tous les établissements industriels de la couche BATI_INDUSTRIEL pour chaque établissement hospitalier. Il faut pour cela utiliser les deux tables PAI_SANTE et BATI_INDUSTRIEL. On notera qu'on ne peut donner une condition de jointure, ni attributaire (pas de champ commun), ni géographique (les objets ne se superposent pas). Dans ce cas on peut construire le produit des deux tables (produit cartésien) sans condition.

```
SELECT * FROM "Pai_SANTE","BATI_INDUSTRIEL"
```

36. http://sqlpro.developpez.com/cours/sqlaz/sousrequetes/

Il reste à ajouter la colonne donnant les distances.

ATTENTION : Faire un produit cartésien sur deux tables sans condition de jointure doit être réservé à des tables de petite dimension.

nb : Pour éviter de faire le produit cartésien complet, on pourrait penser à utiliser sous PostGIS la fonction ST_DWithin() avec un rayon de recherche maximum, fonction qui est disponible que dans spatialite sous le nom de PtDistWithin().

Indice :

La table précédent peut nous donner accès pour chaque PAI_SANTE à la distance minimum de l'établissement le plus proche avec un GROUP BY

```
SELECT PAI_SANTE.ID, min(st_distance(PAI_SANTE.Geometry, BATI_INDUSTRIEL.Geometry)) AS distance_min from PAI_SANTE,BATI_INDUSTRIEL GROUP BY PAI_SANTE.ID
```

On pourrait penser à demander dans le tableau BATI_INDUSTRIEL.ID... mais le résultat serait faux, car il ne faut pas oublier lorsqu'on utilise un GROUP BY que chaque colonne en sortie (dans la clause SELECT) doit être, soit le critère de rupture (celui du GROUP BY), soit être le résultat d'une fonction d'agrégation... (sous PostGIS vous aurez d'ailleurs un message du type

```
ERREUR : la colonne "bati_industriel.id" doit apparaître dans la clause GROUP BY ou être utilisée dans une fonction d'agrégation
```

Spatialite est plus tolérant, mais il vaut mieux prendre les bonnes habitudes !

Nous voila donc avec le tableau suivant :

	ID	distance_min
1	PAISANTE0000000244723936	150,308
2	PAISANTE0000000244723937	124,649

résultat sous-selection

Il faut maintenant trouver les couples (PAI_SANTE.ID BATI_INDUSTRIEL.ID) pour lesquels la distance est l'une ou l'autre des distances de la table précédente... autrement dit exécuter une requête du type

SELECT...st_distance(...) as distance FROM PAI_SANTE, BATI_INDUSTRIEL WHERE distance **IN** (... résultat de la requête donnant les deux distances minimum)

Solutions des exercices

[exercice p. 21] **Solution n°1**

[exercice p. 21] **Solution n°2**

[exercice p. 21] **Solution n°3**

[exercice p. 21] **Solution n°4**

[exercice p. 21] **Solution n°5**

[exercice p. 22] **Solution n°6**

[exercice p. 22] **Solution n°7**

[exercice p. 22] **Solution n°8**

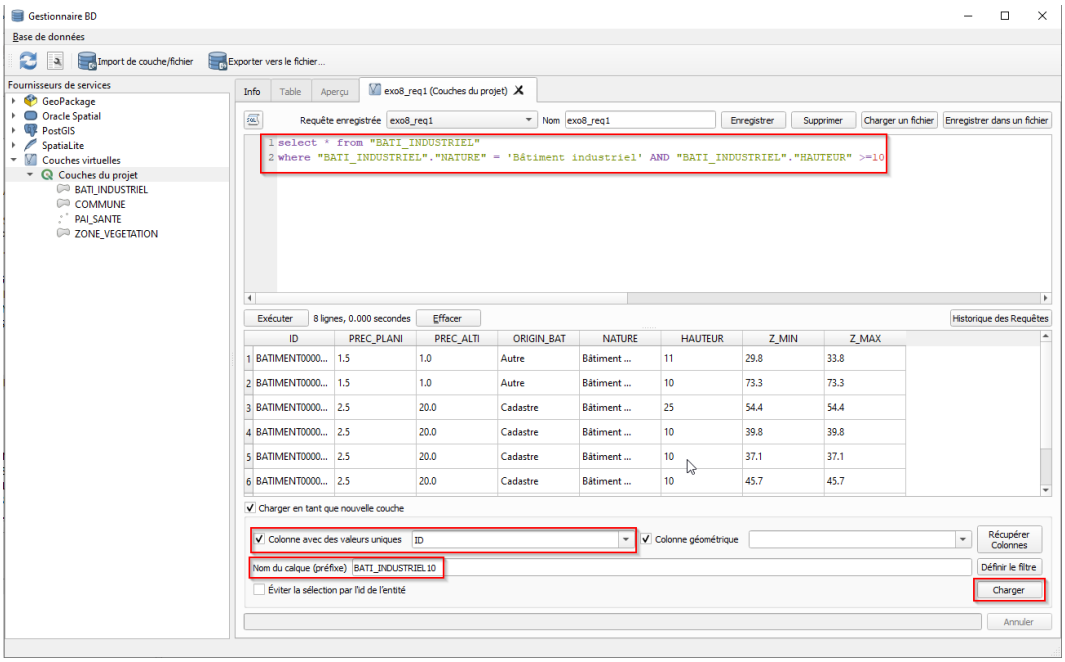
[exercice p. 26] **Solution n°9**

[exercice p. 26] **Solution n°10**

[exercice p. 27] **Solution n°11**

[exercice p. 27] **Solution n°12**

[exercice p. 39]
Solution n°13



solution

[exercice p. 39]
Solution n°14

SELECT *

FROM "ZONE_VEGETATION", "COMMUNE"

WHERE "ZONE_VEGETATION"."NATURE' = 'Forêt fermée de conifères' **and** "COMMUNE"."NOM' = 'La Flèche' **and** st_intersects("ZONE_VEGETATION"."Geometry', "COMMUNE"."Geometry'

[exercice p. 39]
Solution n°15

SELECT
round(sum(st_area(st_intersection("ZONE_VEGETATION"."Geometry","COMMUNE"."Geometry'))) / 10000) **as** surface_ha

FROM "ZONE_VEGETATION", "COMMUNE"

WHERE st_intersects("ZONE_VEGETATION"."Geometry","COMMUNE"."Geometry') **and** "ZONE_VEGETATION"."NATURE' = 'Forêt fermée de feuillus' **and** "COMMUNE"."NOM' = 'La Flèche'

[exercice p. 40]
Solution n°16

La solution est :

```
SELECT PAI_SANTE.ID, BATI_INDUSTRIEL.ID, st_distance(PAI_SANTE.Geometry, BATI_INDUSTRIEL.Geometry) AS distance FROM PAI_SANTE,BATI_INDUSTRIEL WHERE distance IN (SELECT min(st_distance(PAI_SANTE.Geometry, BATI_INDUSTRIEL.Geometry)) AS distance_min FROM PAI_SANTE,BATI_INDUSTRIEL GROUP BY PAI_SANTE.ID)
```

	ID	ID	distance
1	PAISANTE0000000244723937	BATIMENT0000000214067387	124,649
2	PAISANTE0000000244723936	BATIMENT0000000214076737	150,308

exo8 Q4 résultat

Sous PostGIS le `WHERE distance IN...` ne marche pas. Il faut re-écrire explicitement :

`st_distance(PAI_SANTE.Geometry, BATI_INDUSTRIEL.Geometry) IN...`

soit

```
SELECT PAI_SANTE.ID, BATI_INDUSTRIEL.ID, st_distance(PAI_SANTE.Geometry, BATI_INDUSTRIEL.Geometry) AS distance FROM PAI_SANTE,BATI_INDUSTRIEL WHERE st_distance(PAI_SANTE.Geometry, BATI_INDUSTRIEL.Geometry) IN (SELECT min(st_distance(PAI_SANTE.Geometry, BATI_INDUSTRIEL.Geometry)) AS distance_min FROM PAI_SANTE,BATI_INDUSTRIEL GROUP BY PAI_SANTE.PKUID)
```

Complément

Le but de l'exercice est de montrer l'intérêt et la syntaxe d'une requête complexe.

Cependant pour répondre à la question posée nous aurions pu utiliser l'outil **Vecteur -> Outils d'analyse -> Matrice des distances.**

Cet outil ne fonctionnant que sur des couches de points, il faut au préalable créer la couche BATI_INDUSTRIEL_CENTROID avec la fonction **Vecteur -> Outil de géométrie -> Centroïdes de polygones.**

On utilise ensuite l'outil 'matrice de distances' en choisissant 'Utiliser uniquement les points cibles les plus proches' avec k=1, Ceci génère un fichier csv qui donne le résultat cherché. Les distances sont un peu différentes qu'avec la fonction st_distance qui utilise le contour des polygones au lieu du centroïde.

On pourrait également utiliser le plugin **NNjoin** [37]qui permet de calculer pour chaque objet de la couche (input layer), l'objet le plus proche de la couche 'Join vector Layer', ainsi que sa distance.

Il est possible pour la couche 'input layer' d'utiliser '*approximate geometries by centroids*' si la couche n'est pas une couche de points et pour calculer les objets les plus proches d'utiliser '*Approximate geometries*' qui utilise les rectangles englobant plutôt que la géométrie exacte ce qui permet un calcul beaucoup plus rapide. On retrouve dans la couche crée une colonne donnant la distance.

37. http://arken.umb.no/~havatv/gis/qgisplugins/NNJoin/