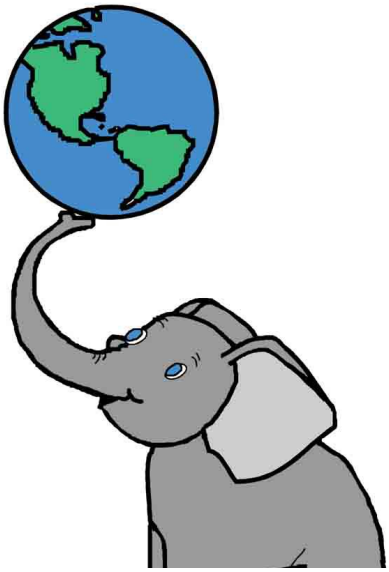
Compléments SQL



Version 1.4

Ministère de la Transistion Ecologique
Licence ETALAB

Janvier 2022

Table des matières

Objectifs

Ce module va vous permettre de compléter votre connaissance du SQL (par rapport au module SQL de QGIS perfectionnement) avec :

- Le langage de définition de données (LDD)
- Des rappels et des compléments sur les différents types de jointure
- Les sous-requêtes et le common table expression (CTE)
- Une introduction aux corrections de géométrie
- Les requêtes inter-bases (DbLink et FDW).

Introduction

Le temps d'apprentissage de ce module est estimé à 4 heures, plus deux heures pour la réalisation des trois exercices tutorés.

Il comporte :

- 4 exercices auto-corrigés ;
- 3 exercices "tutorés" dont vous devrez communiquer le résultat à vos tuteurs.

I Rappels sur SQL

SQL (Structured Query Language qui signifie langage de requêtes structuré) est un langage destiné à la manipulation des bases de données au sein d'un SGBD.

SQL est composé de trois sous-ensembles :

Le Langage de Définition de Données (**LDD**) qui permet de créer et supprimer des objets dans la base de données et que nous allons aborder.

Le Langage de Contrôle de Données (**LCD**) pour gérer les droits sur les objets et les transactions et que nous avons abordés partiellement dans le module 'Administration' avec les commandes GRANT et REVOKE. Nous n'aborderons pas dans ce cours la notion de *transaction*[1].

Le Langage de Manipulation de Données (**LMD**) pour la recherche, l'insertion, la mise à jour et la suppression de données déjà largement abordé dans le module SQL de QGIS Perfectionnement en ce qui concerne le SELECT.

A. Rappels sur des règles ou conventions de bon usage

Quelques règles ou conseils :

- Chaque requête doit se terminer par un point virgule (;)
- Tous les mots obligatoires doivent être présents dans le bon ordre
- Plusieurs requêtes peuvent s'enchaîner
- Les requêtes importantes devraient être sauvegardées
- Les mots obligatoires doivent être écrits en majuscule (lisibilité)
- La requête doit être mise en forme : espaces, tabulations
- La requête doit être commentée (- - en début de ligne ou par /* et */ sur plusieurs lignes)

1 - http://docs.postgresql.fr/9.4/tutorial-transactions.html

Exemple (PgAdmin et DBManager propose une coloration syntaxique):

```
--Commentaire sur 1 ligne : ma requête sur les suf de type 01
SELECT
    idcom,
    sum(dcnt01) AS "Surface de suf de type 01" -- mot clé AS pour "alias"
FROM
    ff_d80_2011.d80_2011_pnb10_parcelle
GROUP BY
    idcom
/* Commentaire
sur 2 lignes : groupement par commune */
ORDER BY
    idcom
```

Exemple de formatage d'une requête SQL

La colonne d'une table est identifiable de la manière suivante :

<nom_schema>.<nom_table>.<nom_colonne>

Exemple :

`ff_d80_2011.d80_2011_pnb_parcelle.idpar`

Pour le nommage :

- Eviter les accents, espaces et caractères spéciaux, sauf éventuellement pour les alias
- N'utilisez que les caractères [az], [09] ou underscore (_)
- Commencer par une lettre (exemple : ff_d59_2009)

II

Le Langage de Définition de Données (LDD)

Le **LDD** permet de créer, modifier, supprimer des objets. Il permet également de définir le domaine des données (nombre, chaîne de caractères, date, booléen...) et d'ajouter des contraintes de valeur sur les données.

Les principales instructions du LDD sont : **CREATE, ALTER, DROP, RENAME**.

Elles peuvent porter sur les objets : **TABLE, INDEX, VIEW, SEQUENCE, USER**.

A. CREATE TABLE

CREATE TABLE crée une nouvelle table initialement vide dans la base de données courante. La table appartient à l'utilisateur qui exécute cette commande.

Si un nom de schéma est donné (par exemple, `CREATE TABLE monschema.matable ...`), alors la table est créée dans le schéma spécifié. Dans le cas contraire, elle est créée dans le schéma courant. Les tables temporaires existent dans un schéma spécial, il n'est donc pas nécessaire de fournir un nom de schéma lors de la création d'une table temporaire. Le nom de la table doit être distinct du nom des autres tables, séquences, index, vues ou tables distantes dans le même

schéma.

Le synopsis général de la commande est :

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] nom_table ( [
{ nom_colonne type_donnees [ COLLATE collation ] [ contrainte_colonne [ ... ] ]
| contrainte_table
| LIKE table_source [ option_like ... ] }
[, ... ]
] )
[ INHERITS ( table_parent [, ... ] ) ]
[ WITH ( parametre_stockage [= valeur] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE nom_tablespace ]
```

Voir *la documentation de PostgreSQL*[2] pour le détail des paramètres.

Il est possible de créer des tables temporaires qui sont supprimées à la fin de la session avec le paramètre TEMPORARY ou TEMP.

Remarque

Les tables temporaires sont placées dans un schéma temporaire `pg_temp_#' (# est un numéro).

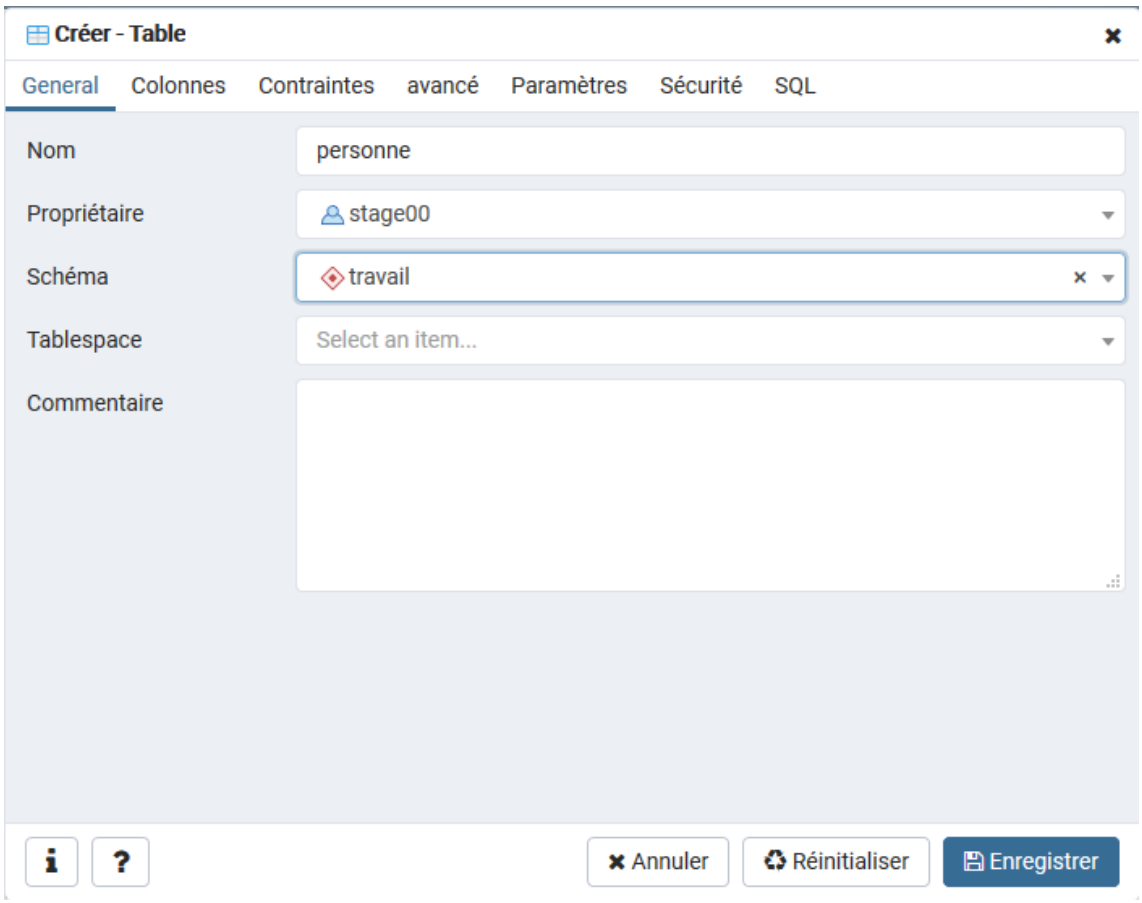
Exemple de création de table :

```
CREATE TABLE personnes (id SERIAL PRIMARY KEY, nom VARCHAR(50), prenom VARCHAR(50)) ;
```

permet de créer une table des personnes avec un identifiant qui est un numéro automatiquement incrémenté comme clef primaire.

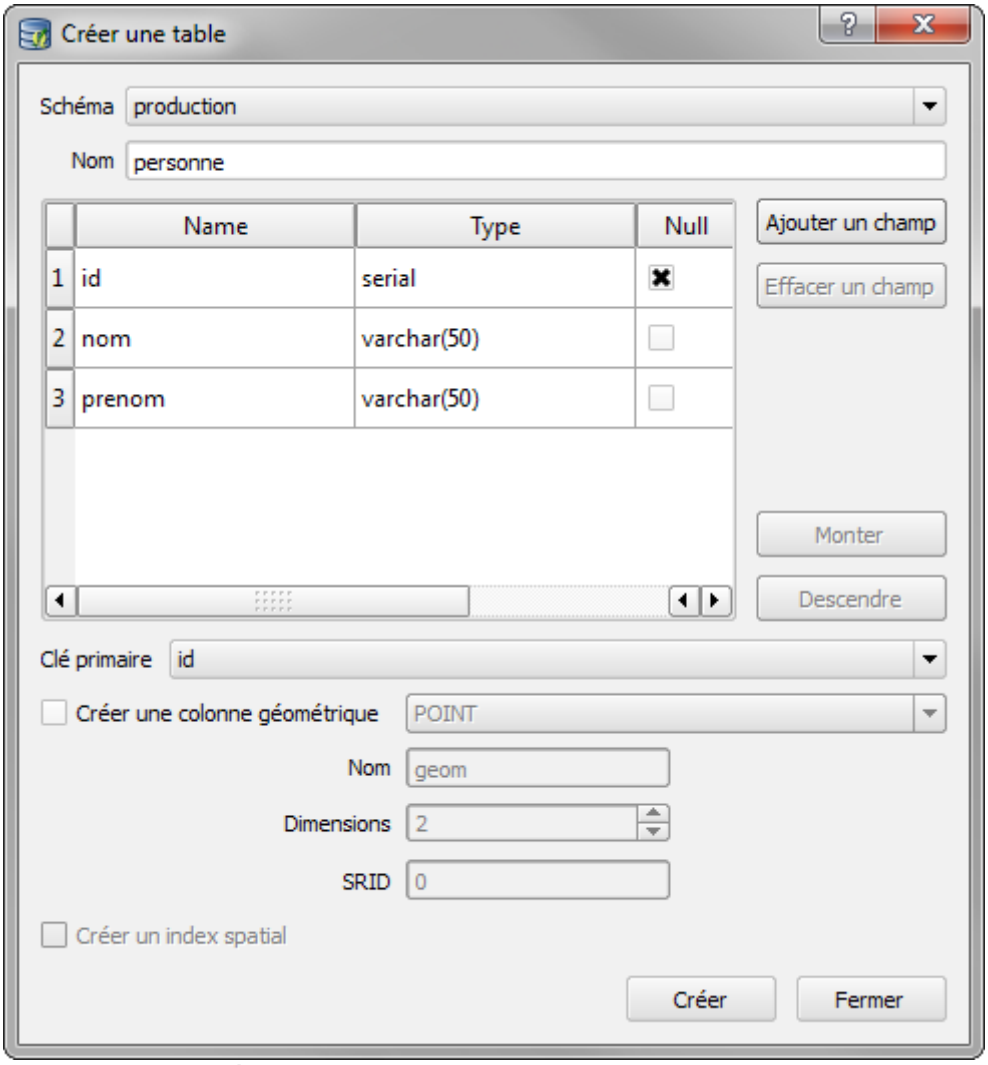
Comme nous l'avons vu au module 2, Sous PgAdmin il est possible de créer une table par clic droit sur un schéma → ajouter un objet → Ajouter une table. Les différents onglets permettent de fixer les paramètres.

2 - http://docs.postgresql.fr/9.4/sql-createtable.html



Ajouter (créer) une table sous PgAdmin

Sous DBManager, comme nous l'avons également vu, il est possible de passer par le menu Table :



Créer une table avec DBManager sous QGIS

Fondamental : Le type du champ de géométrie

Il est important de typer les champs de géométrie, ce qui permet également en option de préciser le système de projection utilisé pour le champ.

On utilise l'instruction `GEOMETRY( [type], [srid])`

Exemple : `CREATE TABLE personnes (id SERIAL PRIMARY KEY, nom VARCHAR(50), geom GEOMETRY(POINT, 2154))`

Le `[type]` peut être :

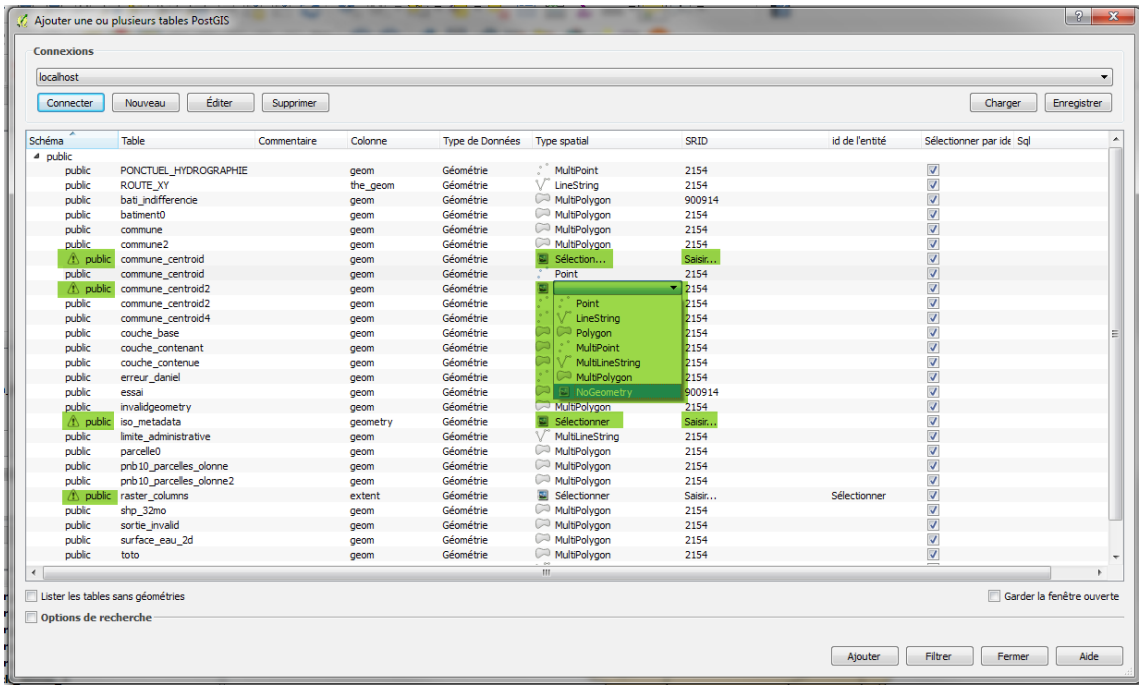
`POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION, POINTM, LINESTRINGM, POLYGONM, MULTIPOINTM, MULTILINESTRINGM, MULTIPOLYGONM, GEOMETRYCOLLECTIONM.`

Le type `GEOMETRY` est utilisé en cas de géométrie hétérogène. Il n'est pas recommandé pour les besoins courants de stocker des géométries hétérogènes dans une même table.

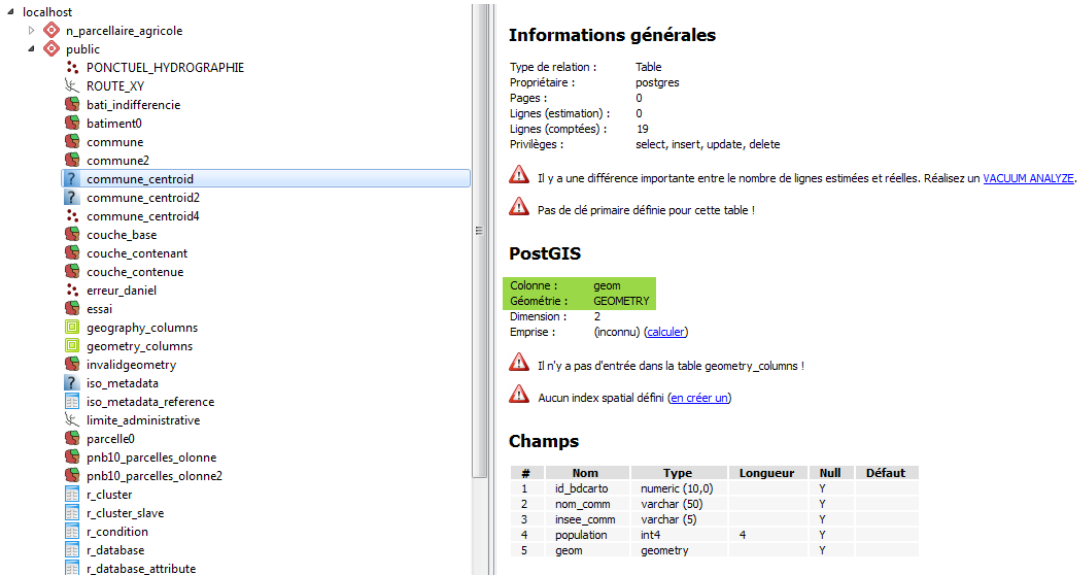
Le `[srid]` defini est le code EPSG (2154 pour le RGF93/Lambert 93).

Une table peut contenir plusieurs champs de géométrie avec potentiellement des types et des SRID différents.

En ajout de couche PostGIS sous QGIS la boîte d'ouverture indique les couches pour lesquelles la géométrie et le type ne sont pas précisés.
(il est possible de le préciser à ce moment pour la session)



Sous DBManager on identifie rapidement le type de géométrie d'une couche. Attention aux couches de type GEOMETRY qui peuvent contenir des données hétérogènes.



Complément

Il est possible de créer une table qui soit le résultat d'une requête SQL :

exemple :

```
CREATE TABLE exemple AS (SELECT * FROM commune) ;
```

Le cas échéant il faudra typer explicitement le résultat d'une fonction donnant une nouvelle géométrie.

Exemple `st_buffer(geom)` retourne par défaut un objet de type `GEOMETRY`, si on veut indiquer en création d'une couche qu'il s'agit d'un POLYGON on utilisera :

```
CREATE TABLE exemple AS (SELECT id,..., st_buffer(geom) : : geometry(POLYGON, 2154))
```

B. 05 - création d'une table en SQL sous DBManager

L'objectif est de mettre en application l'ordre *CREATE TABLE AS*[3]

Question

[*Solution n°1 p 51*]

Avec le rôle *garyXX* (base '*droitXX*')

Dans le *schema* production, En SQL sous QGIS, Créer une nouvelle table *route_xy2* qui sera la duplication de la table *route_xy*.

Indice :

on utilisera une requête SQL

*SELECT *...*

pour renvoyer les enregistrements de la table route_xy:

C. CREATE VIEW (création de vues)

Une ***vue***[4] dans une base de données est une synthèse d'une requête d'interrogation de la base. On peut la voir comme une table virtuelle, définie par une requête (équivalent sous MapInfo des 'tables requêtes').

Elles présentent différents avantages :

- Eviter la réécriture des mêmes commandes SQL plusieurs fois dans les programmes
- Faciliter le travail de requêtes pour des utilisateurs sur des modèles complexes. QGIS accède et visualise les vues
- Réaliser une jointure, une projection (sélection de colonnes) ... systématique
- Faire une mise en forme de données (convert, extract, fonctions de dates...)

Inconvénient :

- Temps de chargement dans QGIS dans le cas de requêtes complexes ou de tables au volume important (voir cependant la création de vues matérialisées un peu plus loin qui permet de mémoriser le résultat et de le rafraîchir à la demande).

3 - http://docs.postgresqlfr.org/9.4/sql-createtableas.html
4 - https://fr.wikipedia.org/wiki/Vue_(base_de_données)

Le syntaxe générale est la suivante :

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW nom
[ ( nom_colonne [, ...] ) ]
[ WITH ( nom_option_vue [= valeur_option_vue] [, ... ] ) ]
AS requête
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Si un nom de schéma est donné (par exemple `CREATE VIEW monschema.mavue ...`), alors la vue est créée dans ce schéma. Dans le cas contraire, elle est créée dans le schéma courant. Les vues temporaires existent dans un schéma spécial. Il n'est donc pas nécessaire de fournir de schéma pour les vues temporaires. Le nom de la vue doit être différent du nom de toute autre vue, table, séquence, index ou table distante du même schéma.

Complément : Modification des vues

Sous PostgreSQL (à partir de 9.4) les vues simples sont *automatiquement modifiables*[5] : le système autorise l'utilisation des commandes INSERT, UPDATE et DELETE sur les vues comme sur les tables. Une vue est modifiable automatiquement si elle satisfait les conditions suivantes :

- La vue doit avoir exactement une entrée (une table ou une autre vue modifiable) dans la liste FROM.
- La définition de la vue ne doit pas contenir de clauses WITH, DISTINCT, GROUP BY, HAVING, LIMIT ou OFFSET au niveau le plus haut.
- La définition de la vue ne doit pas contenir d'opérations sur des ensembles (UNION, INTERSECT ou EXCEPT) au niveau le plus haut.
- La liste de sélection de la vue ne doit pas contenir d'agrégats, de fonctions de fenêtrage ou de fonctions renvoyant des ensembles de lignes.

Si la vue est modifiable automatiquement, le système convertira automatiquement toute commande INSERT, UPDATE ou DELETE sur la vue dans la commande correspondante sur la relation sous-jacente.

l'option **CHECK OPTION** contrôle le comportement des vues automatiquement modifiables. Quand cette option est spécifiée, les commandes INSERT et UPDATE sur la vue seront vérifiées pour s'assurer que les nouvelles lignes satisfont la condition définie dans la vue (autrement dit, les nouvelles lignes sont vérifiées pour s'assurer qu'elles sont visibles par la vue). Dans le cas contraire, la mise à jour est rejetée. Si l'option CHECK OPTION n'est pas indiquée, les commandes INSERT et UPDATE sur la vue sont autorisées à créer des lignes qui ne sont pas visibles avec la vue.

Nous verrons plus loin dans la formation comment réaliser des mises à jour sur des vues plus complexes à l'aide de triggers.

Vues matérialisées

Une vue peut-être matérialisée.
[6]

La requête est exécutée et utilisée pour peupler la vue à l'exécution de la commande (sauf si WITH NO DATA est utilisé) et peut être rafraîchi plus tard en utilisant REFRESH MATERIALIZED VIEW.

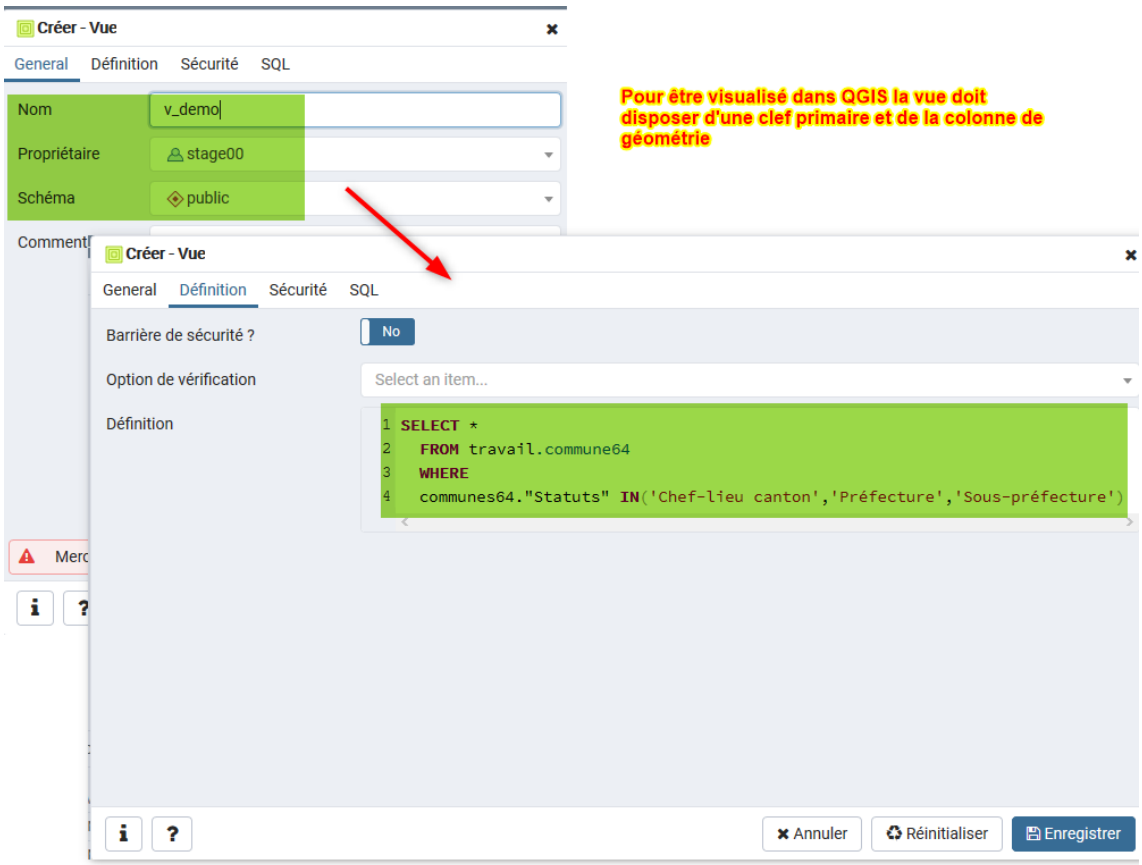
`CREATE MATERIALIZED VIEW` est similaire à `CREATE TABLE AS`, sauf que PostGreSQL se rappelle aussi de la requête utilisée pour initialiser la vue pour qu'elle puisse être

5 - http://docs.postgresql.fr/9.4/sql-createview.html
6 - http://docs.postgresql.fr/9.4/sql-creatematerializedview.html

rafraîchie à la demande. Une vue matérialisée a plusieurs propriétés communes avec une table, mais il n'y a pas de support pour les vues matérialisées temporaires.

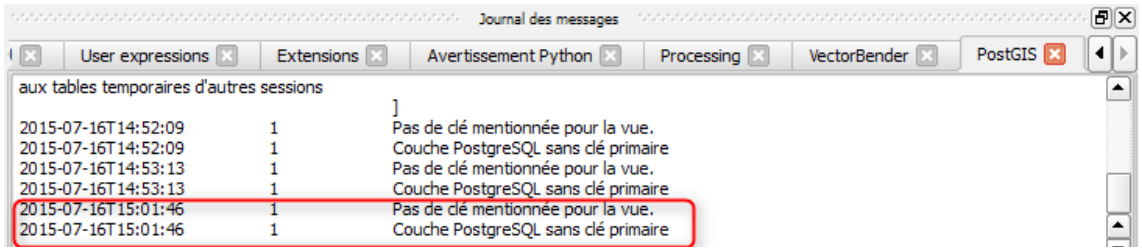
D. créer une vue avec PgAdmin ou DBManager sous QGIS

Il est possible de créer une vue sous PgAdmin :
Clic droit → Ajouter un objet → Ajouter une vue (ou Vues → Ajouter une vue)



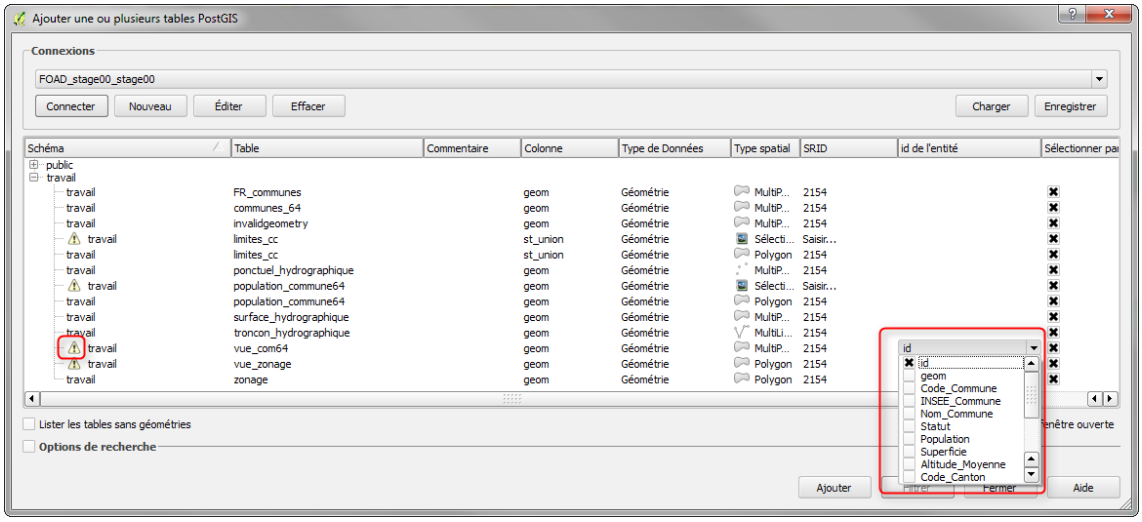
créer une vue sous PgAdmin

Si la couche n'a pas de clef primaire en cas de tentative de chargement dans le canvas QGIS affichera dans le journal des messages dans l'onglet PostGIS :

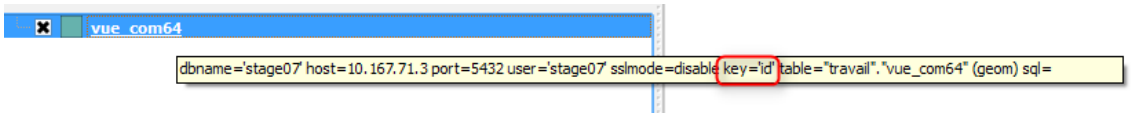


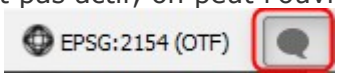
erreur si pas de clef primaire

Ceci si on passe par 'Ajouter une couche PostGIS' (en 2.12 le message est 'Vous devez sélectionner une table afin de pouvoir ajouter une couche'). Il faut alors désigner l'id de l'entité pour la vue.

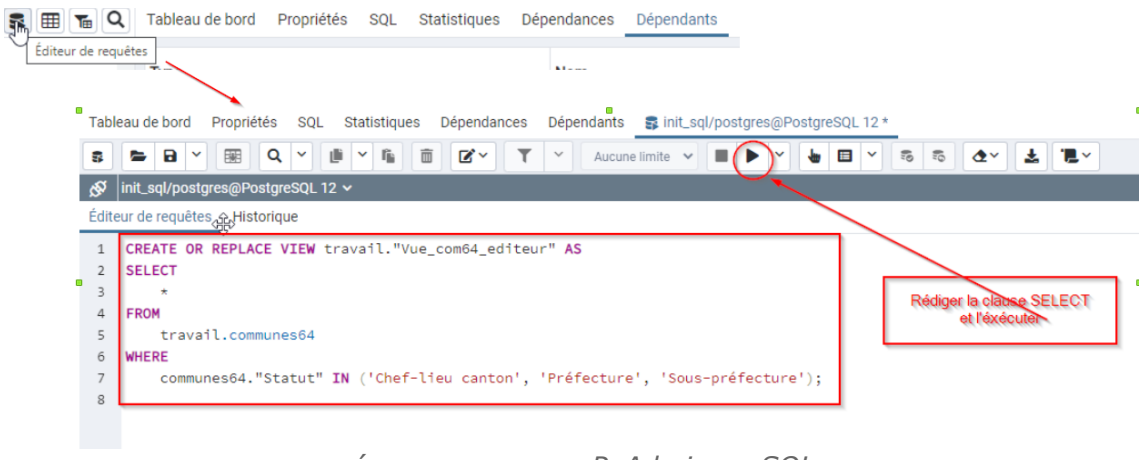


A noter que DbManager ajoute automatiquement la désignation d'une clef ce que l'on peut voir après coup avec l'info-bulle sur la couche.

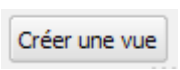


Si le journal des messages n'est pas actif, on peut l'ouvrir à l'aide du bouton à droite du code EPSG, en bas à droite : 

Il est également possible sous PgAdmin de créer la vue en tapant directement le SQL de création dans l'éditeur SQL (Exemple sous pgadmin).



créer une vue sous PgAdmin en SQL

Après avoir réalisé une requête SQL, DBManager propose un bouton 

E. 06 - création d'une vue

L'objectif est de mettre en application l'ordre CREATE VIEW

Question

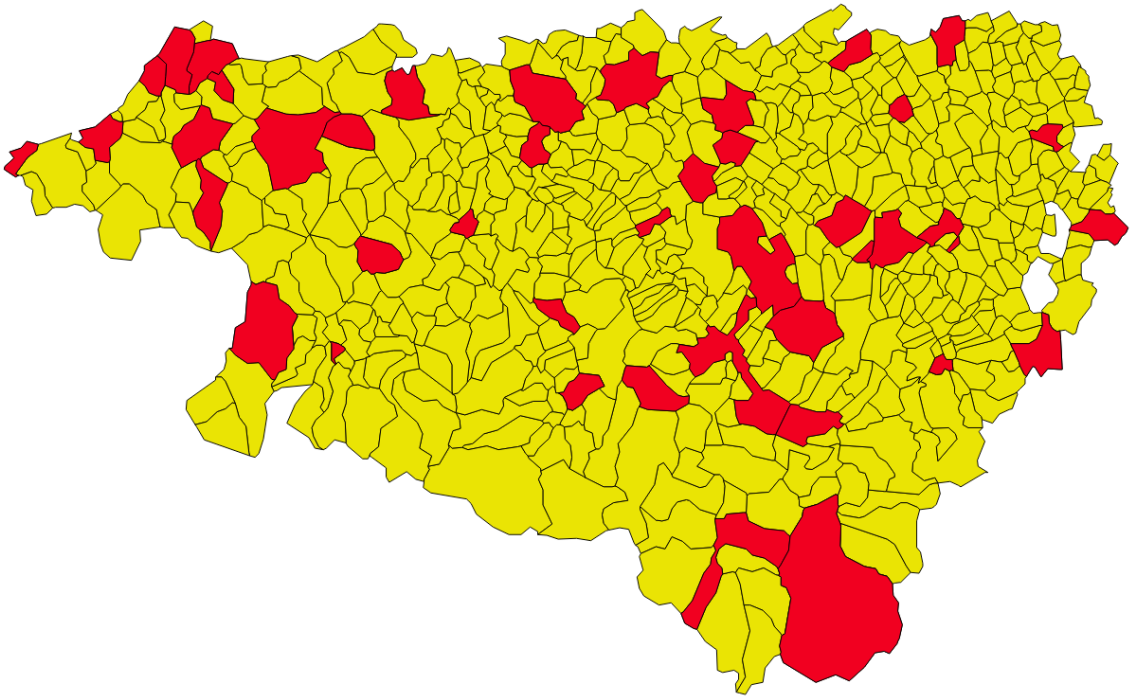
[*Solution n°2 p 51*]

Avec le rôle *stageXX* sur le schéma travail de la base *stageXX* sélectionner à partir de la table *communes64* les communes qui ont comme statut :

- Préfecture
- Sous-préfecture
- Chef-lieu de canton

Créer une vue (que l'on choisira pour cet exercice comme étant matérialisée) *vue_com64* dans le schéma *travail*.

Visualiser le résultat dans QGIS :



F. ALTER

La commande **ALTER** permet de modifier un objet.

ALTER TABLE permet par exemple de modifier la définition d'une table. Par exemple pour ajouter une nouvelle colonne ou réaliser une modification dans le typage d'un champ avec **ALTER TYPE**.

A noter que dans certains cas le transtypage ne peut être automatique il faut alors utiliser une clause **USING** permettant d'expliciter la conversion par une expression.

Méthode : Exercice guidé

Nous souhaitons transformer le type de la colonne *importance* de

production.route_xy2 (base *droitXX*) de VARCHAR(10) en INTEGER...

Il faut cependant noter que cette colonne contient également 'NC' lorsque l'attribut n'est pas rempli. Nous souhaitons mettre 0 lorsque l'attribut n'est pas connu.

la commande

```
ALTER TABLE production.route_xy2 ALTER COLUMN importance TYPE integer;
```

renvoi un message d'erreur indiquant d'utiliser une expression avec USING.

essayez donc :

```
ALTER TABLE production.route_xy2 ALTER COLUMN importance TYPE integer
USING (translate(importance, 'NC', '0')::integer);
```

USING permet de préciser l'expression renvoyant un integer.

Nous utilisons ici la fonction [7] pour remplacer 'NC' par '0'.

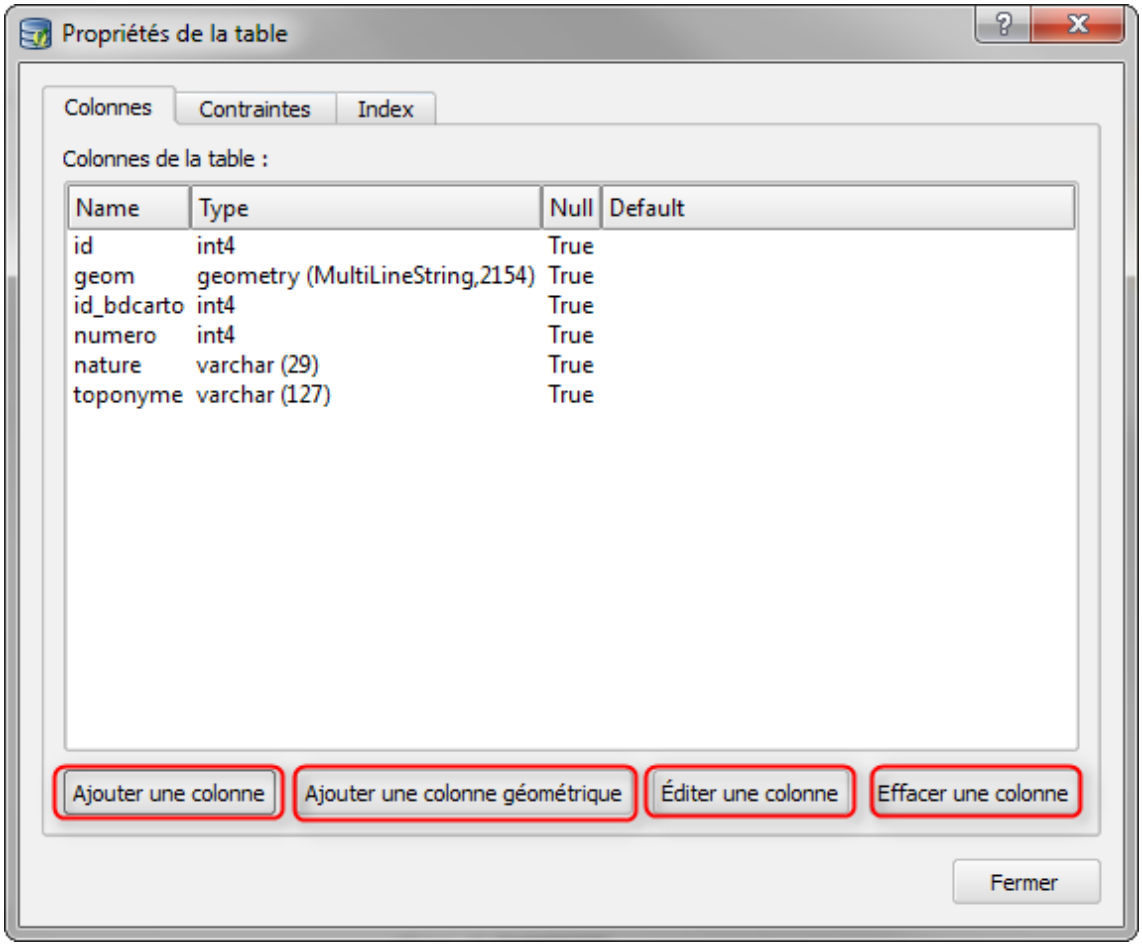
nb : si on relance la commande ALTER..., on aura un message d'erreur avec la fonction translate car le champ importance est maintenant de type integer et la fonction translate attend une chaîne de caractères comme premier paramètre.

[8] permet de modifier la définition d'une vue.

7 - http://docs.postgresqlfr.org/9.4/functions-string.html
8 - http://docs.postgresql.fr/9.4/sql-alterview.html

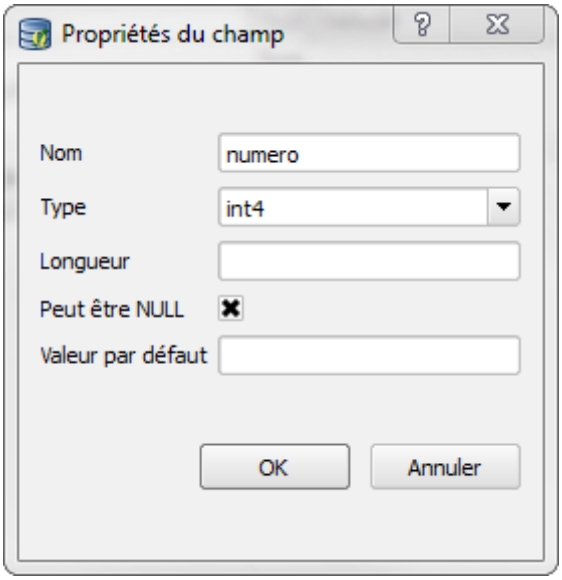
Modification d'un table sous DBManager

Le menu **`Table → editer Table`** de DBManager permet des modifications interactives :



Modification d'une table sous DBManager

exemple 'éditer une colonne' permet le transtypage.



transtypage sous DBManager

G. DROP

DROP permet de supprimer des objets.

DROP TABLE supprime des tables de la base de données. Seul son propriétaire (ou un superuser) peut détruire une table. DROP TABLE supprime tout index, règle, déclencheur ou contrainte qui existe sur la table cible. Néanmoins, pour supprimer une table référencée par une vue ou par une contrainte de clé étrangère d'une autre table, **CASCADE** doit être ajouté.

Pas défaut une table n'est pas supprimée si un objet en dépend (une vue par exemple).

Sous PgAdmin, clic droit sur un objet → supprimer....

Sous DBManager, clic droit sur une table → Effacer.

H. RENAME

Permet de renommer des objets.

Exemple : renommer une table :

```
ALTER TABLE production.itineraire2 RENAME TO itineraire3 ;
```

Sous PgAdmin ; **clic droit → propriétés**, puis changer le nom dans l'onglet propriétés.

Sous DBManager ; **clic droit sur une table → renommer**.

I. INSERT et COPY

La commande [9] permet d'ajouter de nouvelle lignes dans une table.

Le synopsis simplifié de la commande est

```
INSERT INTO <nom_schema>.<nom_table> ([nom_champs] VALUES ([<valeur champ>]
```

exemple :

```
INSERT INTO "Structure" VALUES (38, '200036614', 'AGGLOMERATION SUD PAYS BASQUE', 'CA', 12, NULL);
```

Exemple d'extrait de script par sauvegarde en fichier à plat (SQL) avec commande INSERT explicite :

```
CREATE TABLE "Structure" (
    id integer NOT NULL,
    n_siren character varying(25),
    nom_struct character varying(150),
    nature_jur character varying(10),
    nb_membres integer,
    responsabl character varying(35)
);


ALTER TABLE travail."Structure" OWNER TO utilisateur;

--
-- TOC entry 3293 (class 0 OID 44158)
-- Dependencies: 225
-- Data for Name: Structure; Type: TABLE DATA; Schema: travail; Owner: utilisateur
--

INSERT INTO "Structure" VALUES (38, '200036614', 'AGGLOMERATION SUD PAYS BASQUE', 'CA', 12, NULL);
INSERT INTO "Structure" VALUES (39, '246401723', 'COMMUNAUTE D''AGGLOMERATION PAU-PYRENEES', 'CA', 14, NULL);
INSERT INTO "Structure" VALUES (40, '246400030', 'AGGLOMERATION COTE BASQUE-ADOUR', 'CA', 5, NULL);
INSERT INTO "Structure" VALUES (41, '246400337', 'COMMUNAUTE DE COMMUNES DE LA VALLEE D''OSSAU', 'CC', 18, NULL);
INSERT INTO "Structure" VALUES (42, '246401517', 'COMMUNAUTE DE COMMUNES DU PAYS DE MORLAAS', 'CC', 28, NULL);
INSERT INTO "Structure" VALUES (43, '246401525', 'COMMUNAUTE DE COMMUNES GAVE ET COTEAUX', 'CC', 7, NULL);
```

Exemple utilisation INSERT

Dans la pratique cette commande est surtout utilisée dans les scripts SQL générés par les sauvegardes (dump) et par les applications métiers. Dans le cas d'une utilisation SIG, la saisie des données se fera dans QGIS (en ajoutant la couche au canvas et en passant en mode édition).

La commande **INSERT** est toutefois beaucoup plus lente que la commande [10].

COPY transfère des données entre les tables de PostgreSQL™ et les fichiers du système de fichiers standard. **COPY TO** copie le contenu d'une table vers un fichier tandis que **COPY FROM** copie des données depuis un fichier vers une table (ajoutant les données à celles déjà dans la table). COPY TO peut aussi copier le résultat d'une requête SELECT. Attention le système de fichier est celui du serveur, on ne peut copier ainsi des données sur le poste de travail distant.

On trouvera *ici*[11] des recommandations pour optimiser les performances lors d'insertion massive.

9 - http://docs.postgresql.fr/9.4/sql-insert.html
10 - http://docs.postgresql.fr/9.4/sql-copy.html
11 - http://docs.postgresql.fr/9.4/populate.html

J. UPDATE et DELETE

UPDATE

La commande [12] permet de mettre à jour les lignes d'une table.

Mise à jour sans condition :

```
UPDATE Matable SET Macolonne = NULL ;
```

(mise à NULL de Macolonne pour toutes les lignes de Matable).

Mise à jour avec reprise de valeur :

```
UPDATE Matable SET Macolonne = 2 * Macolonne ;
```

(on double les valeurs de Macolonne)

Mise à jour avec filtrage (WHERE) :

```
UPDATE Matable SET Macolonne = 2 * Macolonne WHERE Macolonne < 10 ;
```

Mise à jour de plusieurs colonnes :

UPDATE Matable SET Macolonne1 = Macolonne1 *2, Macolonne2 = Macolonne2 * 2 ;

Mise à jour à partir des valeurs d'une autre table :

```
UPDATE table1 SET table1.col1 = table2.col1 WHERE table1.id = table2.id ;
```

DELETE

La commande [13] permet de supprimer des lignes d'une table.

Pour supprimer toutes les lignes d'une table on utilisera

```
DELETE FROM Matable ;
```

On peut ajouter une condition :

```
DELETE FROM Matable WHERE population > 5000 ;
```

K. 07 (tutoré) - CREATE TABLE, SQL UPDATE et ALTER TABLE

Utilisation des commandes CREATE TABLE, UPDATE et ALTER TABLE

Question

Sous QGIS.

A partir de la table *travail.commune64* (base *stageXX*) créer une table *population_commune64* ayant pour attribut *id, INSEE_Commune, Nom_Commune, Population* et dont la géométrie un point au centroide de chaque commune.

pour créer la table on utilisera **CREATE TABLE AS...**

On fera attention à typer correctement la géométrie obtenue.

On utilisera ensuite **UPDATE** avec la fonction **initcap()** pour transformer les noms de commune avec seulement la 1ère lettre de chaque mot en majuscule (ex : Mareuil-Sur-Loire)

Puis on utilisera **ALTER TABLE... ALTER COLUMN....USING** pour modifier la géométrie et remplacer les POINTS par des POLYGONES qui seront des buffers des

12 - http://docs.postgresql.fr/9.4/sql-update.html
13 - http://docs.postgresql.fr/9.4/sql-delete.html

centroides obtenu par l'expression 'Population*100'.

On utilisera la fonction **st_buffer()** et on fera attention à modifier le type de la géométrie avec le **ALTER COLUMN geom TYPE....**

Envoyez votre requête aux tuteurs.

Compléments

III

A. Utilisation des alias

Dans le langage SQL il est possible d'utiliser des alias avec le mot clef **AS**, pour renommer temporairement une colonne ou une table dans une requête.

Cette astuce est particulièrement utile pour faciliter la lecture des requêtes.

Exemple, Alias sur une colonne :

```
SELECT macolonne as c1 FROM matable ;
```

Exemple, Alias sur des tables :

```
SELECT mt1.champ_commun, mt2.champ13, mt1.champ6
FROM nom_schema.ma_table1 AS mt1,
JOIN nom_schema.ma_table2 AS mt2
ON mt1.champ_commun = mt2.champ_commun
WHERE mt1.population > 5000;
```

Remarque

Il est plus pratique de mettre des alias significatifs (tout en restant courts). On ne recommande pas d'utiliser des alias peu évocateurs comme t1, t2,... ou a,b,c.

AS n'est pas obligatoire.

Ainsi dans l'exemple ci-dessus on aurait pu écrire :

```
SELECT mt1.champ_commun, mt2.champ13, mt1.champ6
FROM nom_schema.ma_table1 mt1,
```

```
JOIN nom_schema.ma_table2 mt2
ON mt1.champ_commun = mt2.champ_commun
WHERE mt1.population > 5000;
```

B. Rappels et compléments sur les jointures

Jointure naturelle

Cette jointure s'effectue à la condition qu'il y ait des colonnes du même nom et de même type dans les 2 tables.

exemple :

```
SELECT t1.champ_commun, t2.champ13, t1.champ6
FROM nom_schema.ma_table1
NATURAL JOIN nom_schema.ma_table2 ;
```

L'avantage d'un NATURAL JOIN c'est qu'il n'y a pas besoin d'utiliser la clause ON qui permet de préciser la condition de jointure.

A utiliser cependant avec précaution, car il peut y avoir des tables qui ont un champ commun (par exemple timestamp pour l'horodatage) sans que ce soit un champ de jointure.

Complément : Préciser le champ de jointure commun avec USING...

Si il y a un champ commun ayant le même nom dans les deux tables, il est préférable, plutôt que d'utiliser NATURAL JOIN de le préciser avec la syntaxe :

```
SELECT t1.champ_commun, t2.champ13, t1.champ6
FROM nom_schema.ma_table1
JOIN nom_schema.ma_table2 USING(champ_commun)
```

Jointure explicite

Dans tous les autres cas on parle de **jointure explicite**.

Utilisation de JOIN...ON

La syntaxe utilisant la clause **WHERE** pour indiquer la condition de jointure n'est pas la plus recommandée. En effet, elle sert à la fois pour écrire la condition de jointure (ex : t1.champ_commun = t2.champ_commun) et pour un éventuel filtrage des données.

exemple si on souhaite ne conserver que les lignes pour lesquelles le champ population de t1 est > 5000 on écrira :

```
SELECT t1.champ_commun, t2.champ13, t1.champ6
FROM
nom_schema.ma_table1 AS t1,
nom_schema.ma_table2 AS t2
WHERE t1.champ_commun = t2.champ_commun AND t1.population > 5000;
```

Pour éviter cette confusion entre filtrage et condition de jointure il est donc préférable d'utiliser l'opérateur de jointure normalisé **JOIN** :

```
SELECT t1.champ_commun, t2.champ13, t1.champ6
FROM nom_schema.ma_table1 AS t1,
JOIN nom_schema.ma_table2 AS t2
```

```
ON t1.champ_commun = t2.champ_commun
WHERE t1.population > 5000;
```

Conseil

Cette syntaxe est plus lisible (séparation de la condition de jointure de la condition de filtrage).

De plus il est possible de contrôler jusqu'à un certain point le planificateur en utilisant la syntaxe explicite (JOIN). Voir *la documentation PostgreSQL sur ce sujet*[14].

A partir de maintenant nous utiliserons systématiquement cette syntaxe.

C. Les jointures internes, externes, croisées et qualifiées

Dans PostgreSQL, il existe plusieurs types de jointures très bien détaillés dans *la documentation*[15].

Une jointure croisée (**cross join**) créé le produit cartésien des deux tables. c'est-à-dire que pour chaque combinaison de ligne (ex : t1 CROSS JOIN t2) provenant de t1 et t2, la table jointe contiendra une ligne disposant de toutes les colonnes de t1 suivies par toutes les colonnes de t2.

Les autres jointures (FULL, RIGHT, LEFT, INNER JOIN) sont des **jointures qualifiées**.

Une **jointure externe** (OUTER) permet de faire en sorte que le résultat comprenne toutes les lignes des tables (ou d'au moins une des tables de la jointure), même s'il n'y a pas correspondance des lignes entre les différentes tables mise en œuvre dans la jointure. Le mot clef OUTER est optionnel avec LEFT, RIGHT, FULL (voir ci-dessous).

14 - http://docs.postgresql.fr/current/explicit-joins.html
15 - http://docs.postgresql.fr/current/queries-table-expressions.html

L'exemple ci-dessous montre des exemples de différents types de jointure :

<table>
<tr><th>Tables en entrée</th><th>Type de jointure</th><th>Requête SQL</th><th>Résultat</th></tr>
<tr><td rowspan="4">
<table><tr><th colspan="2">Table 1</th></tr><tr><th>id</th><th>nom</th></tr><tr><td>1</td><td>a</td></tr><tr><td>2</td><td>b</td></tr><tr><td>3</td><td>c</td></tr></table>
<table><tr><th colspan="2">Table 2</th></tr><tr><th>id</th><th>valeur</th></tr><tr><td>1</td><td>xxx</td></tr><tr><td>3</td><td>yy</td></tr><tr><td>3</td><td>yyyy</td></tr><tr><td>5</td><td>zzz</td></tr></table>
</td><td>Croisée
CROSS JOIN
Pas de critère de jointure</td><td>SELECT *
FROM table1 AS t1
CROSS JOIN table2 AS t2 ;</td><td>
<table><tr><th>t1.id</th><th>t1.nom</th><th>t2.id</th><th>t2.valeur</th></tr><tr><td>1</td><td>a</td><td>1</td><td>xxx</td></tr><tr><td>1</td><td>a</td><td>3</td><td>yy</td></tr><tr><td>1</td><td>a</td><td>3</td><td>yyyy</td></tr><tr><td>1</td><td>a</td><td>5</td><td>zzz</td></tr><tr><td>2</td><td>b</td><td>1</td><td>xxx</td></tr><tr><td>2</td><td>b</td><td>3</td><td>yy</td></tr><tr><td>2</td><td>b</td><td>3</td><td>yyyy</td></tr><tr><td>2</td><td>b</td><td>5</td><td>zzz</td></tr><tr><td>3</td><td>c</td><td>1</td><td>xxx</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yy</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yyyy</td></tr><tr><td>3</td><td>c</td><td>5</td><td>zzz</td></tr></table>
</td></tr>
<tr><td>Complète
FULL JOIN
Nécessité d'avoir un critère de jointure</td><td>SELECT *
FROM table1 AS t1
FULL JOIN table2 AS t2
ON t1.id = t2.id ;</td><td>
<table><tr><th>t1.id</th><th>t1.nom</th><th>t2.id</th><th>t2.valeur</th></tr><tr><td>1</td><td>a</td><td>1</td><td>xxx</td></tr><tr><td>2</td><td>b</td><td></td><td></td></tr><tr><td>3</td><td>c</td><td>3</td><td>yy</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yyyy</td></tr><tr><td></td><td></td><td>5</td><td>zzz</td></tr></table>
</td></tr>
<tr><td>À gauche
LEFT JOIN
Nécessité d'avoir un critère de jointure</td><td>SELECT *
FROM table1 AS t1
LEFT JOIN table2 AS t2
ON t1.id = t2.id ;</td><td>
<table><tr><th>t1.id</th><th>t1.nom</th><th>t2.id</th><th>t2.valeur</th></tr><tr><td>1</td><td>a</td><td>1</td><td>xxx</td></tr><tr><td>2</td><td>b</td><td></td><td></td></tr><tr><td>3</td><td>c</td><td>3</td><td>yy</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yyyy</td></tr></table>
</td></tr>
<tr><td>À droite
RIGHT JOIN
Nécessité d'avoir un critère de jointure</td><td>SELECT *
FROM table1 AS t1
RIGHT JOIN table2 AS t2
ON t1.id = t2.id ;</td><td>
<table><tr><th>t1.id</th><th>t1.nom</th><th>t2.id</th><th>t2.valeur</th></tr><tr><td>1</td><td>a</td><td>1</td><td>xxx</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yy</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yyyy</td></tr><tr><td></td><td></td><td>5</td><td>zzz</td></tr></table>
</td></tr>
<tr><td></td><td>Naturelle
JOIN
Nécessité d'avoir un critère de jointure</td><td>SELECT *
FROM table1 AS t1
JOIN table2 AS t2
ON t1.id = t2.id ;</td><td>
<table><tr><th>t1.id</th><th>t1.nom</th><th>t2.id</th><th>t2.valeur</th></tr><tr><td>1</td><td>a</td><td>1</td><td>xxx</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yy</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yyyy</td></tr></table>
</td></tr>
</table>

Conseil : visualisation des types de jointure (en anglais)

Ce *site*[16] permet de générer un exemple de code SQL en fonction du type de résultat désiré...

16 - http://sql-joins.leopard.in.ua/

D. Les expressions conditionnelles

L'opérateur **CASE WHEN** permet de construire une colonne dont la valeur dépend d'autres colonnes (mise à jour de colonne) : il est très utile pour les groupements et aussi pour construire des tableaux croisés

```
CASE WHEN condition THEN résultat
[WHEN ...]
[ELSE résultat]
END
```

Les clauses **CASE** peuvent être utilisées partout où une expression est valide. Chaque condition est une expression qui renvoie un résultat de type boolean. Si le résultat de la condition est vrai, alors la valeur de l'expression CASE est le résultat qui suit la condition. Si le résultat de la condition n'est pas vrai, toutes les clauses WHEN suivantes sont parcourues de la même façon.

exemple :

select a,
CASE
WHEN a=1 THEN 'un'
WHEN a=2 THEN 'deux'
ELSE 'autres'
END
FROM test ;
va renvoyer :

a	case
1	un
2	deux
3	autres

E. 08 - CASE ... WHEN (1)

Première mise en pratique du CASE...WHEN

Question

[*Solution n°3 p 52*]

Sous PgAdmin, avec la table *commune64* du schema *travail*, afficher un tableau indiquant pour chaque arrondissement avec le nom de l'arrondissement en clair la population, sachant que les codes arrondissement sont les suivants :

1 : Bayonne

2 : Oloron Sainte-Marie

3 : Pau

Le résultat doit être :

	Arrondissement text	Population en milliers double precision
1	Pau	297.7
2	Bayonne	266.7
3	Oloron Sainte-Marie	74

F. 09 (tutoré) - CASE ... WHEN (2)

Autre mise en application du **CASE...WHEN...**

Question

Réaliser le tableau croisé ci-dessous donnant le nombre de communes par arrondissement selon le statut de la commune, toujours à partir de *commune64* :

	statut commune character varying(20)	Bayonne bigint	Oloron Sainte-Marie bigint	Pau bigint
1	Chef-lieu canton	14	10	15
2	Commune simple	108	144	253
3	Préfecture	0	0	1
4	Sous-préfecture	1	1	0

Envoyer votre requête aux tuteurs.

G. Les sous-requêtes

Avec SQL il est possible d'imbriquer des requêtes. On parlera de 'sous-requêtes' ou 'requêtes imbriquées' ou encore 'requêtes en cascades'.

Une sous-requête peut-être utilisée pour renvoyer un résultat dans chacune des clauses de la requête principale SELECT, WHERE, FROM.

Ceci permet de résoudre élégamment des problèmes complexes ou d'optimiser les performances des requêtes.

A noter qu'il est en général possible de décomposer un problème complexe en problèmes intermédiaires plus simple en créant par exemple des tables temporaires, mais les performances risquent d'être dégradées.

Méthode : Sous-requêtes renvoyant une valeur unique

Une requête renvoyant une valeur **unique** peut-être imbriquée partout ou une constante peut figurer dans la requête principale.

Exemple dans la clause SELECT:

```
SELECT "Nom_Commune", round(("Population" / (SELECT avg(a."Population")
FROM travail.communes64 AS a)) ::numeric ,2) AS "Population/Moyenne"
FROM travail.communes64
```

renvoie le rapport de la population à la moyenne du département pour chaque commune :

	Nom_Commune	Population/Moyenne
1	ANOS	0.17
2	ORAAS	0.17
3	MONEIN	3.77
4	HAUT-DE-BOS...	0.26
5	SAINT-ARMOU	0.51
6	GABASTON	0.51
7	BAYONNE	38.04
8	BEYRIE-SUR-JO...	0.43

Exemple dans la clause WHERE (le plus usité) :

On recherche les communes dont la population est supérieure à la moyenne des populations des communes.

```
SELECT "Nom_Commune", "Population" FROM travail.communes64 WHERE "Population" > (SELECT avg("Population") FROM travail.communes64) ORDER BY "Population"
```

renvoi (extrait):

	Nom_Commune character varying(50)	Population double precision
1	BIDOS	1.2
2	SOURAIDE	1.2
3	CHERAUTE	1.2
4	LAGOR	1.2
5	NAVARRENX	1.2
6	BIDACHE	1.2
7	SOUMOULOU	1.2
8	GUETHARY	1.3
9	GARLIN	1.3
10	NAVAILLES-ANGOS	1.3
11	ARTIGUELOUVE	1.4
12	SAUVETERRE-DE-BEARN	1.4
13	OUSSE	1.4

Méthode : L'instruction WITH (Common Table expressions)

WITH fournit une façon d'écrire les sous-requêtes pour utilisation dans une requête SELECT plus étendue. Les sous-requêtes, qui sont souvent appelées des expressions communes de tables (**Common Table Expressions**) ou CTE, peuvent être considérées comme la déclaration d'une table temporaire n'existant que pour la requête. Une utilisation de cette fonctionnalité est de découper des requêtes

complexes en parties plus simples.

Exemple :

Sans WITH

```
SELECT * FROM (SELECT * FROM MatableA) AS Monsubset
```

Avec WITH :

```
WITH Monsubset AS (SELECT * FROM MatableA)
SELECT * FROM Monsubset
```

Chaque ordre auxiliaire dans une clause WITH peut être un SELECT, INSERT, UPDATE, ou DELETE; et la clause WITH elle même est attachée à un ordre primaire qui peut lui aussi être un SELECT, INSERT, UPDATE, ou DELETE.

On trouvera des exemples plus pertinents dans *la documentation de PostgreSQL*[17]

Méthode : Sous-requête renvoyant une liste

Une sous-requête peut renvoyer une liste de valeurs (c'est-à-dire une colonne).

Dans ce cas elle ne peut être utilisée que comme critère de comparaison avec certains opérateurs comme IN (ou ALL, ou ANY).

Exemple : On recherche les N° INSEE des communes qui sont dans des structures de plus de 20 membres.

```
select code_insee FROM travail."Delegation" WHERE code_siren IN (SELECT n_siren FROM travail."Structure" WHERE nb_membres > 20)
```

beaucoup de requêtes utilisant le IN (comme le NOT IN) peuvent être simplifiées en utilisant des jointures. En général les performances seront meilleures en utilisant une jointure que dans le cas d'une sous-requête avec [NOT] IN.

Dans notre exemple :

```
SELECT
"Delegation".code_insee
FROM
travail."Delegation",
travail."Structure"
WHERE
"Structure".n_siren = "Delegation".code_siren AND
"Structure".nb_membres > 20;
```

Pour en savoir plus on pourra par exemple consulter *le SQL de A à Z sur les sous-requêtes*[18].

Méthode : Sous-requête vide ou non vide

Une requête renvoyant des valeurs ou pas peut-être imbriquée dans un prédicat EXISTS, UNIQUE ou MATCH.

Par exemple, Le prédicat **EXISTS** permet de tester l'existence ou l'absence de données dans la sous-requête. Si la sous-requête renvoie au moins une ligne, même remplie de marqueurs **NULL**, le prédicat est vrai. Dans le cas contraire le prédicat à pour valeur faux.

Exemple :

```
SELECT sum(a."Population")FROM travail.communes64 AS a WHERE EXISTS(SELECT * FROM travail.communes64 AS b WHERE b."Population" > 1000)
```

renvoi

17 - http://docs.postgresql.fr/current/queries-with.html
18 - http://sqlpro.developpez.com/cours/sqlaz/sousrequetes/

	sum double precision
1	

```
SELECT sum(a."Population")FROM travail.communes64 AS a WHERE
EXISTS(SELECT * FROM travail.communes64 AS b WHERE b."Population" > 10)
```

renvoi

	sum double precision
1	638.400000000002

car dans ce cas le select sous EXISTS renvoie au moins une valeur.

On obtient le même résultat que

```
SELECT sum(a."Population")FROM travail.communes64 AS a
```

Notre exemple est donc peu pertinent, mais montre le principe de l'utilisation du EXISTS.

H. 10 (tutoré) - sous-requêtes

Exercice permettant de mettre en pratique les sous-requêtes

Question

A partir des tables travail.structure, travail.Delegation et travail.communes64

trouver la liste des communes qui appartiennent à la "COMMUNAUTE DE COMMUNES NIVE-ADOUR"

Pour réaliser cet exercice

1) utiliser des sous-requêtes sans WITH

2) Utiliser WITH

3) utiliser une jointure.

Comparer les temps de traitement entre 1,2 et 3. Commenter.

Le résultat à obtenir :

	Nom_Commune character varying(50)
1	URCUIT
2	URT
3	MOUGUERRE
4	LAHONCE
5	SAINT-PIERRE-D'IRUBE
6	VILLEFRANQUE

Envoyer vos requêtes et commentaires sur les temps de traitement au tuteur.

nb : Ne pas utiliser code_siren='246401855', mais utiliser struc.nom_struct ='COMMUNAUTE DE COMMUNES NIVE-ADOUR' dans les requêtes.

I. Correction de géométrie

Notion de validité de la géométrie des entités

Une géométrie non valide est la cause principale de l'échec des requêtes spatiales.

Dans 90% des cas la réponse à la question "pourquoi mes requêtes me renvoient un message d'erreur du type 'TopologyException error' est : "un ou plusieurs des arguments passés sont invalides". Ce qui nous conduit à nous demander : que signifie invalide et pourquoi est-ce important ?

La validité est surtout importante pour les polygones, qui définissent des surfaces et requièrent une bonne structuration.

Certaines des règles de validation des polygones semblent évidentes, et d'autres semblent arbitraires (et le sont vraiment) :

- Les contours des polygones doivent être fermés.
- Les contours qui définissent des trous doivent être inclus dans la zone définie par le contour extérieur.
- Les contours ne doivent pas s'intersecter (ils ne doivent ni se croiser ni se toucher).
- Les contours ne doivent pas toucher les autres contours, sauf en un point unique.

Les deux dernières règles font partie de la catégorie arbitraire.

PostGIS est conforme au standard OGC OpenGIS Specifications.

Les entités géométriques des bases PostGIS doivent ainsi être à la fois **simples** et **valides**.

Par exemple, calculer la surface d'un polygone comportant un trou à l'extérieur ou construire un polygone à partir d'une limite non simple n'a pas de sens.

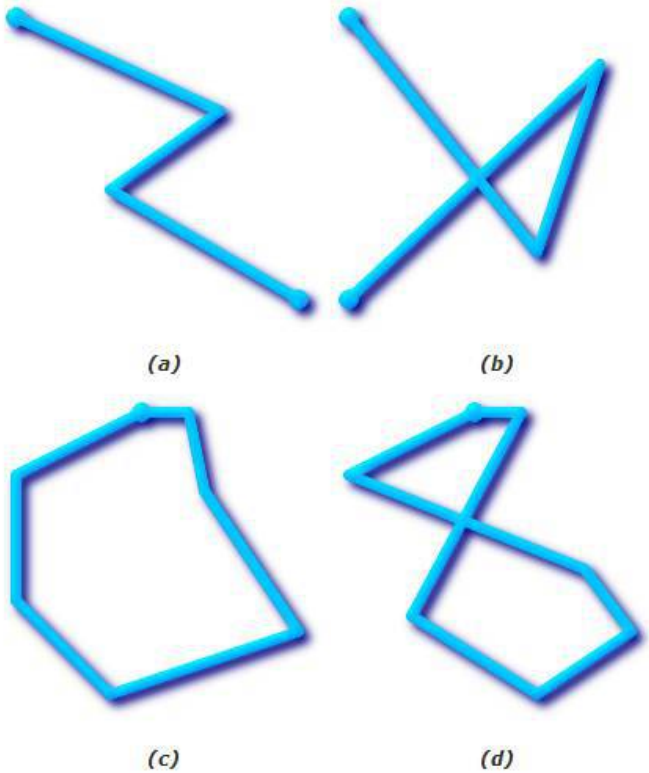
Selon les spécifications de l'OGC, une **géométrie simple** est une géométrie qui ne comporte pas de points géométriques anormaux, comme des auto-intersections ou des auto-tangences, ce qui concerne essentiellement les points, les multi-points, les polylignes et les multi-polylignes.

La notion de **géométrie valide** concerne principalement les polygones et les multi-polygones et le standard définit les caractéristiques d'un polygone valide.

Un point est par nature simple, ayant une dimension égale à 0.

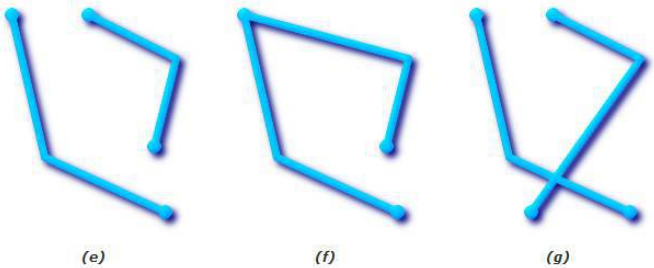
Un objet multi-points est simple si tous les points le composant sont distincts.

Une polyligne est simple si elle ne se recroise pas (les extrémités peuvent être confondues, auquel cas c'est un anneau et la polyligne est close).



Les polylignes (a) et (c) sont simples, mais pas les polylignes (b) et (d)

Une multi-polyligne est simple si toutes les polylignes la composant sont elles-mêmes simples et si les intersections existant entre 2 polylignes se situent à une extrémité de ces éléments :



(e) et (f) sont des multipolylignes simples, pas (g)

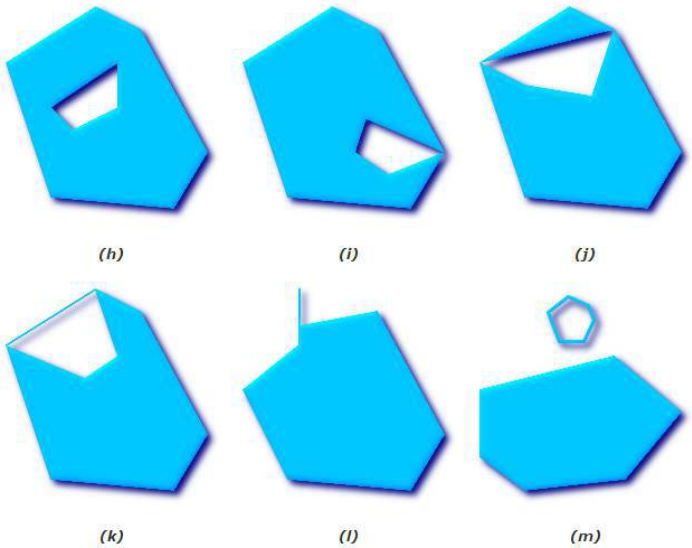
Un polygone est valide si ses limites, qui peuvent être constituées par un unique anneau extérieur (polygone plein) ou par un anneau extérieur et un ou plusieurs anneaux intérieurs (polygone à trous), ne comporte pas d'anneaux se croisant.

Un anneau peut intersecter la limite mais seulement en un point (pas le long d'un segment).

Un polygone ne doit pas comporter de lignes interrompues (les limites doivent être

continues) ou de point de rebroussement (pic).

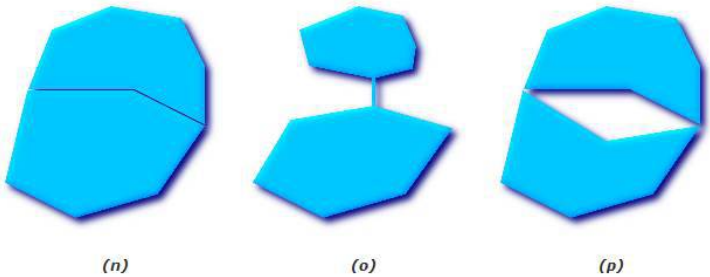
Les anneaux intérieurs doivent être entièrement contenus dans la limite extérieure du polygone.



(h) et (i) sont des polygones valides, (j), (k), (l), (m) sont des polygones ni simples ni valides mais (j) et (m) sont des multi-polygones valides

Un multi-polygone est valide si et seulement si tous les polygones le composant sont valides et si aucun intérieur d'un polygone ne croise celui d'un autre.

Les limites de 2 polygones peuvent se toucher, mais seulement par un nombre fini de points (pas par une ligne).



(n) et (o) ne sont pas des multi-polygones valides, par contre (p) est valide

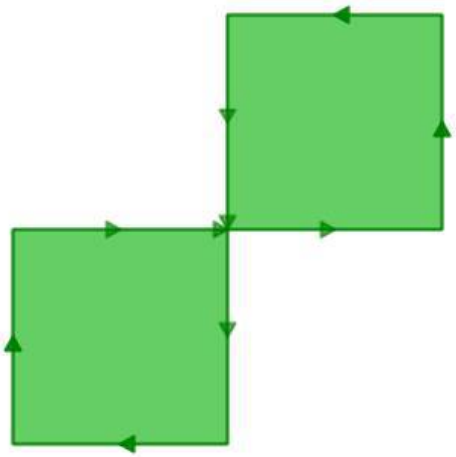
Par défaut, PostGIS n'applique pas le test de validité géométrique lors de l'import d'entités géométriques, parce que le test de validité géométrique consomme beaucoup de temps processeur pour les géométries complexes, notamment les polygones.

Il faut donc mettre en œuvre différentes méthodes pour vérifier la validité de la géométrie des entités.

Exemple

Exemple :

Le polygone POLYGON((0 0, 0 1, 2 1, 2 2, 1 2, 1 0, 0 0)) n'est pas valide



Le contour externe est exactement en forme en 8 avec une intersection au milieu.

Notez que la fonction de rendu graphique est tout de même capable d'en afficher l'intérieur, donc visuellement cela ressemble bien à une "aire" : deux unités carré, donc une aire couplant ces deux unités.

Essayons maintenant de voir ce que pense la base de données de notre polygone en calculant sa surface :

```
SELECT ST_Area(ST_GeometryFromText('POLYGON((0 0, 0 1, 1 1, 2 1, 2 2, 1 2, 1 1, 1 0, 0 0))'));
```

```
 st_area
---------
       0
```

Que ce passe-t-il ici ?

L'algorithme qui calcule la surface suppose que les contours ne s'intersectent pas.

Un contour normal devra toujours avoir une surface qui est bornée (l'intérieur) dans un sens de la ligne du contour (peu importe le sens).

Cependant, dans notre figure en 8, le contour externe est à droite de la ligne pour un lobe et à gauche pour l'autre.

Cela entraîne que les surfaces qui sont calculées pour chaque lobe annulent la précédente (l'une vaut 1 et l'autre -1) donc le résultat est une surface égale à 0.

- Détecter la validité

Dans l'exemple précédent nous avions un polygone que nous savions non-valide.

Comment déterminer les géométries non valides dans une table d'un million d'enregistrements ?

Avec la fonction ST_IsValid(geometry) utilisée avec notre polygone précédent, nous obtenons rapidement la réponse :

SELECT ST_IsValid(ST_GeometryFromText('POLYGON((0 0, 0 1, 1 1, 2 1, 2 2, 1 2, 1 1, 1 0, 0 0))'));

résultat : f (false)

Maintenant nous savons que la géométrie de l'entité n'est pas valide mais nous ne savons pas pourquoi.

Nous pouvons utiliser la fonction ST_IsValidReason(geometry) pour trouver la cause de non validité :

SELECT ST_IsValidReason(ST_GeometryFromText('POLYGON((0 0, 0 1, 1 1, 2 1, 2 2, 1 2, 1 1, 1 0, 0 0))'));

résultat : Self-intersection[1 1]

En plus de la cause d'invalifdité (auto-intersection), la localisation de la non validité (coordonnée (1 1)) est aussi renvoyée.

Nous pouvons aussi utiliser la fonction ST_IsValid(geometry) pour tester les tables comme dans l'exemple suivant :

SELECT ST_IsValidReason(geom) FROM *monschema.table* WHERE Not ST_IsValid (geom);

Détecter et corriger les erreurs géométriques par la pratique

Nous ébauchons ici la problématique des géométries invalides et des corrections envisageables.

Pour créer une table de géométries invalides dans le schéma travail:

exécuter le script SQL ci-dessous :

```
CREATE TABLE travail.invalidgeometry (id serial, type varchar(20), geom geometry(MULTIPOLYGON, 2154), PRIMARY KEY(id));

INSERT INTO travail.invalidgeometry (type, geom) VALUES ('Hole Outside Shell', ST_multi(ST_GeomFromText('POLYGON((465000 6700000, 465010 6700000, 465010 6700010, 465000 6700010, 465000 6700000), (465015 6700015, 465015 6700020, 465020 6700020, 465020 6700015, 465015 6700015))',2154)));

INSERT INTO travail.invalidgeometry (type, geom) VALUES ('Nested Holes', ST_multi(ST_GeomFromText('POLYGON((465030 6700000, 465040 6700000, 465040 6700010, 465030 6700010, 465030 6700000), (465032 6700002, 465032 6700008, 465038 6700008, 465038 6700002, 465032 6700002), (465033 6700003, 465033 6700007, 465037 6700007, 465037 6700003, 465033 6700003))',2154)));
```

```
INSERT INTO travail.invalidgeometry (type, geom) VALUES ('Dis. Interior', ST_Multi(ST_GeomFromText('POLYGON((465060 6700000, 465070 6700000,465070 6700010, 465060 6700010, 465060 6700000), (465065 6700000, 465070 6700005, 465065 6700010, 465060 6700005, 465065 6700000))', 2154)));
INSERT INTO travail.invalidgeometry (type, geom) VALUES ('Self Intersect.', ST_multi(ST_GeomFromText('POLYGON((465090 6700000, 465100 6700010, 465090 6700010, 465100 6700000, 465090 6700000))',2154)));
INSERT INTO travail.invalidgeometry (type, geom) VALUES ('Ring Self Intersect.', ST_multi(ST_GeomFromText('POLYGON((465125 6700000, 465130 6700000, 465130 6700010, 465120 6700010, 465120 6700000, 465125 6700000, 465123 6700003, 465125 6700006, 465127 6700003, 465125 6700000))',2154)));
INSERT INTO travail.invalidgeometry (type, geom) VALUES ('Nested Shells', ST_multi(ST_GeomFromText('MULTIPOLYGON(((465150 6700000, 465160 6700000, 465160 6700010, 465150 6700010, 465150 6700000)),(( 465152 6700002, 465158 6700002, 465158 6700008, 465152 6700008, 465152 6700002)))',2154)));
```

Visualiser la table dans QGIS :

Chacun de ces objets est invalide.

Vérifions-le avec la requête suivante :

```
SELECT id, type, ST_IsValidReason(geom) FROM travail.invalidgeometry WHERE NOT ST_IsValid(geom);
```

qui nous renvoi :

	id integer	type character varying(20)	st_isvalidreason text
1	1	Hole Outside Shell	Hole lies outside shell[465015 6700015]
2	3	Nested Holes	Holes are nested[465033 6700003]
3	4	Discon. Interior	Interior is disconnected[465070 6700005]
4	5	Self Intersect.	Self-intersection[465095 6700005]
5	6	Ring Self Intersect.	Ring Self-intersection[465125 6700000]
6	7	Nested Shells	Nested shells[465152 6700002]

Méthode : Correction avec ST_MakeValid()

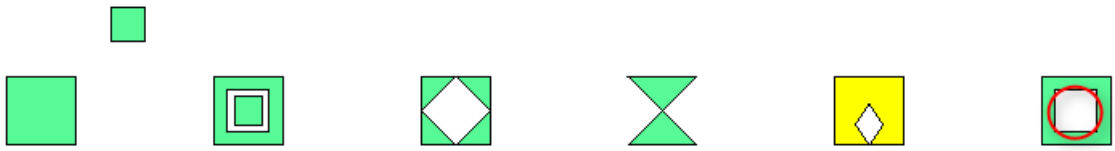
Executer le script SQL :

```
CREATE TABLE travail.makevalidgeometry AS
(SELECT id, type, ST_MULTI(ST_MakeValid(geom))::geometry(MULTIPOLYGON, 2154) AS geom FROM travail.invalidgeometry WHERE NOT St_IsValid(geom));
```

qui renvoi :

```
NOTICE:  Hole lies outside shell at or near point 465015 6700015
NOTICE:  Holes are nested at or near point 465033 6700003
NOTICE:  Interior is disconnected at or near point 465070 6700005
NOTICE:  Self-intersection at or near point 465095 6700005
NOTICE:  Ring Self-intersection at or near point 465125 6700000
NOTICE:  Nested shells at or near point 465152 6700002
La requête a été exécutée avec succès : 6 lignes modifiées. La requête a été exécutée en 142 ms.
```

Charger la nouvelle table dans QGIS :



Seul le dernier élément est différent :

constater que la requête :

```
SELECT id, type, ST_IsValidReason(geom) FROM travail.makevalidgeometry
WHERE NOT ST_IsValid(geom);
```

Ne trouve plus d'erreur.

Exécuter :

```
SELECT st_AsText(geom) from travail.makevalidgeometry ;
```

qui renvoi :

```
"MULTIPOLYGON(((465000 6700000,465000 6700010,465010 6700010,465010
6700000,465000 6700000)),((465015 6700015,465015 6700020,465020
6700020,465020 6700015,465015 6700015)))"
"MULTIPOLYGON(((465030 6700000,465030 6700010,465040 6700010,465040
6700000,465030 6700000),(465032 6700002,465038 6700002,465038
6700008,465032 6700008,465032 6700002)),((465033 6700003,465033
6700007,465037 6700007,465037 6700003,465033 6700003)))"
"MULTIPOLYGON(((465065 6700000,465060 6700000,465060 6700005,465065
6700000)),((465060 6700005,465060 6700010,465065 6700010,465060
6700005)),((465065 6700010,465070 6700010,465070 6700005,465065
6700010)),((465070 6700005,465070 6700000,465065 6700000,4650 (...)"
"MULTIPOLYGON(((465090 6700000,465095 6700005,465100 6700000,465090
6700000)),((465095 6700005,465090 6700010,465100 6700010,465095
6700005)))"
"MULTIPOLYGON(((465125 6700000,465120 6700000,465120 6700010,465130
6700010,465130 6700000,465125 6700000),(465125 6700000,465127
6700003,465125 6700006,465123 6700003,465125 6700000)))"
"MULTIPOLYGON(((465150 6700000,465150 6700010,465160 6700010,465160
6700000,465150 6700000),(465152 6700002,465158 6700002,465158
6700008,465152 6700008,465152 6700002)))"
```

On peut comparer avec la table de départ.

```
SELECT st_AsText(geom) from travail.invalidgeometry
```

renvoi :

```
"MULTIPOLYGON(((465000 6700000,465010 6700000,465010 6700010,465000
6700010,465000 6700000),(465015 6700015,465015 6700020,465020
6700020,465020 6700015,465015 6700015)))"
"MULTIPOLYGON(((465030 6700000,465040 6700000,465040 6700010,465030
```

```
6700010,465030     6700000),(465032     6700002,465032     6700008,465038
6700008,465038     6700002,465032     6700002),(465033     6700003,465033
6700007,465037 6700007,465037 6700003,465033 6700003)))"
"MULTIPOLYGON(((465090   6700000,465100   6700010,465090   6700010,465100
6700000,465090 6700000)))"
"MULTIPOLYGON(((465125   6700000,465130   6700000,465130   6700010,465120
6700010,465120     6700000,465125     6700000,465123     6700003,465125
6700006,465127 6700003,465125 6700000)))"
"MULTIPOLYGON(((465150   6700000,465160   6700000,465160   6700010,465150
6700010,465150    6700000)),((465152    6700002,465158    6700002,465158
6700008,465152 6700008,465152 6700002)))"
"MULTIPOLYGON(((465060   6700000,465070   6700000,465070   6700010,465060
6700010,465060     6700000),(465065     6700000,465070     6700005,465065
6700010,465060 6700005,465065 6700000)))"
```

'Hole Outside Shell' (trou extérieur à l'enveloppe) : un seul polygone composé d'un trou en dehors du polygone de départ est devenu un multi-polygone composé de deux polygones.

'Nested Holes' (trous imbriqués) : est devenu un multipolygone composé d'un polygone avec trou et d'un deuxième polygone qui est au centre (le plus petit).

'Disconnected Interior' : (le trou touche le polygone en plus de 1 point) : est devenu un multi-polygone composé de 4 polygones en triangle.

'Self Intersection' (auto-intersection) : un multi-polygone composé de deux polygones en triangle.

'Ring Self Intersection'(anneau auto-intersectant, avec ici réduction de l'anneau en un point) : est devenu un polygone à trou (trou en contact en 1 point avec l'enveloppe ce qui est correct au sens de geos).

'Nested shell' (polygones imbriqués) : est devenu un polygone à trou.

Complément

A la recherche d'une requête universelle...

La requête suivante permet de prendre en compte des cas de figure où ST_MakeVAlid() transforme des objets en collection de (multi)polygones, (multi)polyliges, ou (multi)point.

C'est le cas si, par exemple on part d'une couche avec un polygone comprenant un arc qui deviendra avec ST_Makevalid() une collection d'un polygone et d'une polyligne.

On peut utiliser :

update ma_table SET geom = ST_Multi(ST_CollectionExtract(st_simplify(ST_ForceCollection(ST_MakeValid(geom)), 0),3))

st_simplify(geometry,0) permet de supprimer les nœuds en double.

ST_ForceCollection() permet de forcer le résultat comme une collection, même si St_MakeValid() ne produit pas de collection, afin de pouvoir utiliser après St_CollectionExtract()

On utilise ST_CollectionExtract(geometry,3) pour ne retenir dans **cet exemple** que les **polygones** de la collection.

et ST_Multi() pour forcer le résultat en multipolygones, même s'il n'y a qu'un seul polygone extrait.

Enfin pour ne pas simplifier inutilement des objets (par exemple linéaire dans notre exemple ou l'on recherche des polygones) on utilisera finalement la requête suivante :

update ma_table SET geom = st_multi(ST_Simplify(ST_Multi(ST_CollectionExtract(ST_ForceCollection(ST_MakeValid(geom)),3)),0)) WHERE ST_GeometryType(geom) like ' %Polygon' and St_invalid(geom)

Méthode : Correction par ST_Buffer(geom,0)

Une autre solution souvent proposée sur les forums est de corriger les erreurs de géométrie en réalisant un buffer nul.

Executer le script suivant :

```
CREATE TABLE travail.geometryvalidbuffer AS
(SELECT id, type, ST_Multi(ST_Buffer(geom,0))::geometry(MULTIPOLYGON, 2154) FROM travail.invalidgeometry WHERE NOT ST_IsValid(geom));
```

Charger cette couche dans QGIS :

Constater que les parties de polygones en jaune ont disparu en particulier pour les polygones en 'papillons'.

La méthode avec st_buffer(geom, 0) est donc une alternative, si la méthode st_makevalid() échoue (gros jeu de données), mais elle ne doit être utilisée que si on a déjà corrigé les polygones en 'papillons'

Il est fortement conseillé de noter le nombre d'objets de la couche avant et après traitement pour une première vérification de pertinence.

Complément

Il existe l'algorithme **Réparer les géométries** dans Processing (menu traitement).

Nous n'avons abordé ici que la problématique des géométries invalides au sens de l'osgeo. Mais il peut exister d'autres motif de correction comme des incohérences entres polygones d'une même couche (dépend des spécifications de saisie), comme des recouvrements (partiels ou non), des trous, des petits polygones (scories),...

Il existe d'autres solutions pour experts, comme l'utilisation de fonctions de GRASS (v.in.ogr et v.clean) . Le plugin *vérificateur de géométrie*[19] qui est une extension principale de QGIS est désormais relativement stable (QGIS 3 et plus) et intéressant. Voir la *documentation*[20] de QGIS.

Complément : Pour en savoir plus...

consulter la *page spécifique*[21] sur le site Géoinformation.

Philippe Desboeufs a également créé un plugin *'Le Nettoyeur (de polygones)*[22]',

19 - http://www.qgis.ch/de/ressourcen/anwendertreffen/2015/geometry-cleaning-plugins
20 - https://docs.qgis.org/3.16/fr/docs/user_manual/plugins/core_plugins/plugins_geometry_checker.html
21 - http://www.geoinformations.developpement-durable.gouv.fr/verification-et-corrections-des-geometries-a3522.html
22 - http://piece-jointe-carto.developpement-durable.gouv.fr/NAT002/QGIS/plugins3/nettoyeur.zip

disponible dans le répertoire des *plugins du MTE*[23].

J. 11 - Correction de géométries

Mise en pratique de corrections de géométrie avec PostGIS sur un exemple

Question

[*Solution n°4 p 52*]

Charger sous QGIS le fichier `localisation_geographique_des_mares_2012.SHP`

Dans le gestionnaire de couches, afficher le décompte des entités de cette couche.

Utiliser l'algorithme '*vérifier la validité*' de la boîte à outils de traitement pour évaluer les erreurs avec la méthode de contrôle GEOS, puis QGIS.

Dans le gestionnaire de couche procéder à un regroupement des couches produites par chaque méthode et décompter les entités non valides dans chaque cas.

Identifier les types d'erreurs détectés par chaque méthode dans la table d'attribut.

Procéder à l'import dans le schéma `travail` de la base `stageXX` (remplacer XX par votre numéro de stagiaire) en utilisant l'algorithme '`import vector into PostGIS database` ou '`Exporter vers PostgreSQL`' dans QGIS 3 et +

Réaliser une requête SQL pour Décompter ; les géométries null, les géométries invalides,

Sous PostGIS corriger à l'aide de requêtes SQL les entités avec la fonction *ST_Makevalid()*

puis corriger les problèmes de *collection* (st_makevalid() renvoi parfois des objets collection au lieu de multipolygon) avec *st_multi(st_CollectionExtract(ST_Force_Collection(st_makevalid(geom)),3))* en remplacement du seul *st_makevalid()*

Ensuite corriger les problèmes de nœuds doubles avec la combinaison de fonction *st_multi(st_simplify(geom,0))*

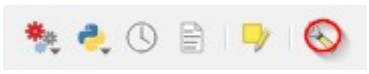
et enfin supprimer les entités n'ayant pas de géométrie associée.

recharger la table sous QGIS et re-analyser la table avec `vérifier la validité` selon les deux méthodes.

Qu'en concluez-vous ?

Comme on souhaite importer dans PostgreSQL une couche qui présente des erreurs de géométrie, il convient de paramétrer le module de traitement pour qu'il autorise l'exécution des algorithmes sans tenir compte des erreurs de géométrie.

Ouvrir les options du module de traitement :



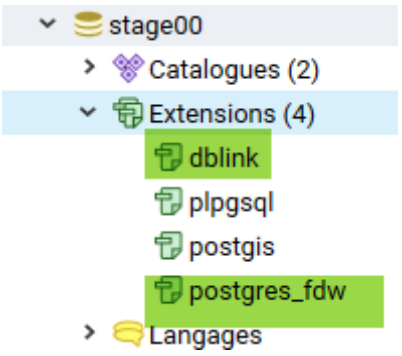
Positionner l'option 'filtrage des éléments invalides' à 'ne pas filtrer'

23 - http://piece-jointe-carto.developpement-durable.gouv.fr/NAT002/QGIS/plugins/plugins.xml

K. Requêtes inter-bases

Il peut être nécessaire de réaliser des requêtes inter-bases de données. PostgreSQL propose deux extensions permettant des liaisons avec des bases autre que celle sur laquelle la connexion est établie. Il s'agit de **Dblink** et **Postgres_FDW** (PostgreSQL > 9.3). Les bases peuvent être sur un serveur distant.

Ces extensions ne sont pas nativement installées dans PostgreSQL ; il faut donc les créer si elle ne sont pas disponibles. Pour voir si une extension est disponible examiner la partie extensions de la base sous PgAdmin :



Méthode : DBLINK

Si l'extension n'est pas installée il faut utiliser la commande :

```
CREATE EXTENSION dblink ;
```

Il se peut que vous rencontriez le message :

ERREUR: n'a pas pu ouvrir le fichier de contrôle d'extension « /usr/share/postgresql/9.3/extension/dblink.control » : Aucun fichier ou dossier de ce type

Voir dans ce cas avec l'administrateur système pour installer les paquets de contributions complémentaires (`sudo apt-get install postgresql-contrib`).

Une fois le module installé, il est possible d'utiliser [24] pour établir une connexion.

Pour faire un exercice, nous souhaitons faire une requête entre la table *production.route_xy* de la base *droitsXX* et la table *consultation.commune* de la base *newdroitXX* (créée dans le module 2 sur l'administration de PostgreSQL).

Dans pgAdmin, se connecter à la base *droitsXX* avec le rôle *stageXX*.

Exécuter :

```
select * from dblink_connect('demo','host=localhost user=stageXX password=stageXX dbname=newdroitXX') ;
```

(demo est le nom de la connexion qui permettra de ne pas renseigner tous les paramètres dans le prochain appel)

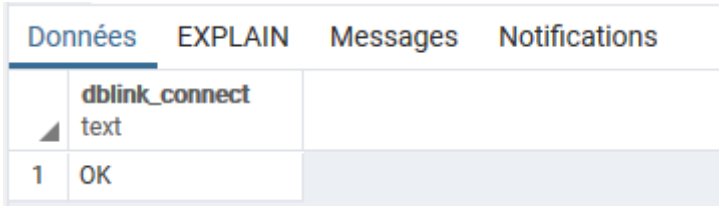
nb : On utilise *localhost* car on est déjà connecté sur le serveur et que l'on souhaite se reconnecter dans une autre base du **même** serveur.

Le cas échéant il faut préciser le port d'écoute du serveur, si ce n'est pas celui par défaut (5432), exemple :

```
select * from dblink_connect('demo','host=172.XX.XX.XXX port=5632 user=postgres password=postgres dbname=stage00')
```

24 - http://docs.postgresql.fr/9.1/contrib-dblink-connect.html

Si vous avez le message OK comme indiqué ci-dessous, la connexion est établie :



Il est maintenant possible de réaliser des requête sur la table distante avec [25]:

Exemple :

```
WITH toto AS (SELECT * from dblink('demo', 'select insee_comm, geom FROM consultation.commune')
as foo(insee_comm character(5), geom geometry(MultiPolygon, 2154)))
SELECT production.route_xy.*, toto.insee_comm FROM production.route_xy, toto
WHERE st_intersects(production.route_xy.geom, toto.geom);
```

nb : *foo*[26] est en gros, l'équivalent de toto en anglais !

Pour se déconnecter on utilisera :

```
SELECT * FROM dblink_disconnect('demo')
```

Attention

Pour pouvoir se connecter sur le serveur 'distant' il faut que ce soit autorisé dans le fichier ph_hba.conf du serveur 'distant'. Si le serveur 'distant' est le même (comme dans l'exemple ci-dessus) et si l'adresse IP du serveur est 10.167.71.3 il faut avoir quelque chose comme :

```
host all all 10.167.71.3 /32 md5
```

Méthode : FDW

postgres_fdw[27] propose a peu près les mêmes possibilités tout en étant en général plus performant.

si l'extension n'existe pas la créer :

```
CREATE EXTENSION postgres_fdw ;
```

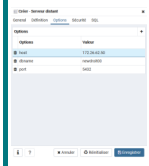
Avec le même exemple que pour Dblink on aura (remplacer XX par votre numéro de stagiaire) :

```
CREATE SERVER demo FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host '172.26.62.50', dbname 'newdroitXX', port '5432') ;
```

L'ordre SQL peut-être généré sous pgadmin à partir de l'assistant :

25 - docs.postgresql.fr/9.4/contrib-dblink-function.html
26 - https://fr.wikipedia.org/wiki/Variable_m%C3%A9tasyntaxique
27 - http://docs.postgresql.fr/current/postgres-fdw.html

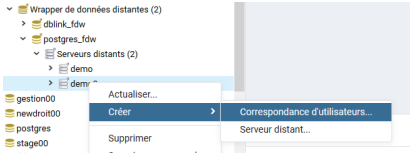
puis en entrant les différentes options :



Création d'un utilisateur habilité (remplacer XX par votre numéro de stagiaire):

```
CREATE USER MAPPING FOR stageXX SERVER demo OPTIONS (user 'stageXX', password 'stageXX') ;
```

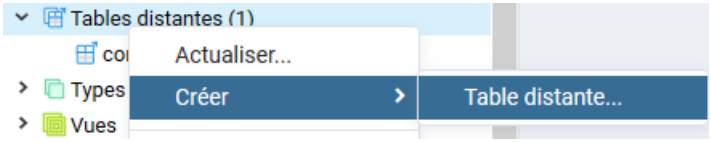
que l'on peut créer par l'assistant sous pgadmin :



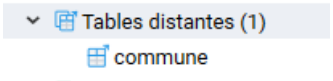
Création de la la table distante (il faut donner la liste des colonnes comme pour la création d'une table)

```
CREATE FOREIGN TABLE commune(geom geometry(MultiPolygon, 2154), nom_comm character varying(50)) SERVER demo OPTIONS (schema_name 'consultation', table_name 'commune') ;
```

Que l'on peut créer par l'assistant sous pgadmin :



La table apparaît désormais dans la liste des tables distances du schéma public :



Il est maintenant possible d'exécuter une requête comme si la table commune était locale :

```
SELECT * FROM production.route_xy, commune WHERE st_intersects(production.route_xy.geom, commune.geom);
```

Pour supprimer la connexion sur une table on utilisera :

```
DROP FOREIGN TABLE commune ;
```

pour un utilisateur :

```
DROP USER MAPPING IF EXISTS FOR geoadmin SERVER demo ;
```

et pour le serveur (il faut d'abord avoir supprimé les objets liés) :

```
DROP SERVER demo ;
```

On peut également utiliser le clic droit > *supprimer* sur les objets dans pgadmin.

Complément : FDW et optimisation

nb : A partir de PostgreSQL 9.6 Il est possible pour des raisons de performances d'ajouter `extension 'postgis'` au moment de la création du serveur ce qui donne :

```
CREATE SERVER demo FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host '172.26.62.50', dbname 'newdroitXX', port '5432', extension 'postgis') ;
```

Comme indiqué *ici*[28] , il est de la responsabilité de l'utilisateur que les extensions listées existent bien et se comportent de façon identique sur les serveurs local et distant.

28 - https://docs.postgresql.fr/13/postgres-fdw.html

Complément : FDW pour intégrer différentes ressources externes

FDW dispose en réalité de nombreux *connecteur de données distantes*[29].

Il faut contacter, le cas échéant, l'administrateur système du serveur pour faire installer ces connecteurs.

Nous donnons ci-dessous un exemple avec ogr_fdw pour se connecter à un ensemble de fichier SHP (ogr_fdw est disponible par défaut dans les paquets PostGIS à partir de la version 2.2.0).

```
-- si pas déjà fait
CREATE EXTENSION ogr_fdw ;
-- creation du FDW pour OGR
CREATE FOREIGN DATA WRAPPER ogr_fdw
HANDLER ogr_fdw_handler
VALIDATOR ogr_fdw_validator;
--
-- création du serveur
CREATE SERVER serv_shp_geobase_referentiels_bdcarto_2015_administratif
FOREIGN DATA WRAPPER ogr_fdw
OPTIONS (datasource '//SBL87-GEOIDE/GB_REF/0_REFERENTIELS/BDCARTO_087_2015/1_DONNEES_LIVRAISON/ADMINISTRATIF',format 'ESRI Shapefile');
--
-- creation du schéma d'accueil
create schema referentiels_bdcarto_2015_administratif;
--
-- création des foreign tables dans ce schéma
import foreign schema ogr_all from server serv_shp_geobase_referentiels_bdcarto_2015_administratif into referentiels_bdcarto_2015_administratif;
--
```

On pourra trouver *ici*[30] un exemple pour consulter des données externes via le protocole WFS

Complément : FDW : Import de schémas à partir de PostgreSQL 9.5

PostgreSQL (> 9.5) offre la possibilité d'importer des schémas entier avec *IMPORT FOREIGN SCHEMA*[31]

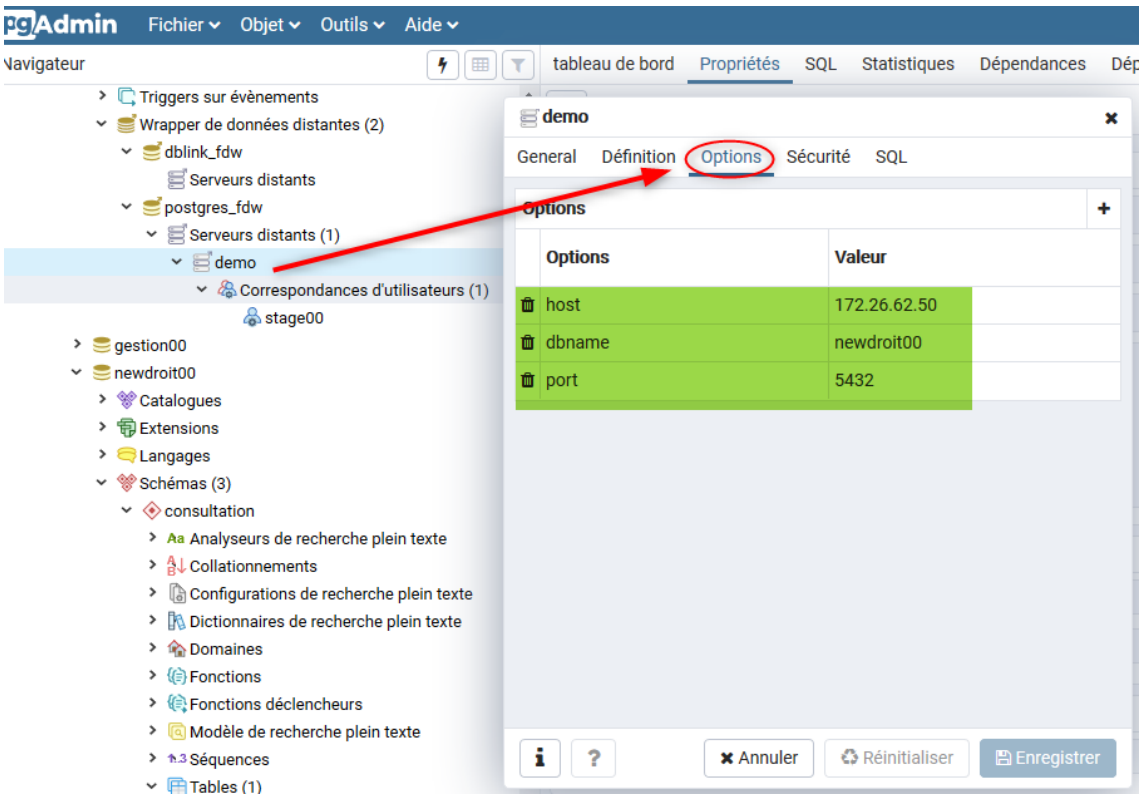
Les nouvelles tables externes sont toutes créées dans le schéma cible, qui doit déjà exister.

L'utilisateur doit avoir le droit USAGE sur l'instance distante, ainsi que le droit CREATE sur le schéma cible.

29 - https://wiki.postgresql.org/wiki/Foreign_data_wrappers
30 - https://docs.3liz.org/formation-postgis/fdw/
31 - https://docs.postgresql.fr/current/sql-importforeignschema.html

Complément

On peut voir les serveurs distants dans pgadmin, ainsi que leur propriétés (avec clic droit sur le nom du serveur) :



Solution des exercices

> Solution n°1 *(exercice p. 16)*

```
CREATE TABLE production.route_xy2 AS (SELECT * FROM production.route_xy);
```

Rafraîchir l'affichage et vérifier que la table *route_xy2* a bien été créée dans le schéma *production*.

Remarque

CREATE TABLE AS semble posséder des similitudes avec la création d'une vue mais est, en fait, assez différente : elle crée une nouvelle table et n'évalue la requête qu'une seule fois, pour le chargement initial de la nouvelle table. Les modifications ultérieures de la table source ne sont pas prises en compte. Au contraire, une vue réévalue l'instruction SELECT de définition à chaque appel.

CREATE TABLE AS ne possède pas l'option IF EXISTS.

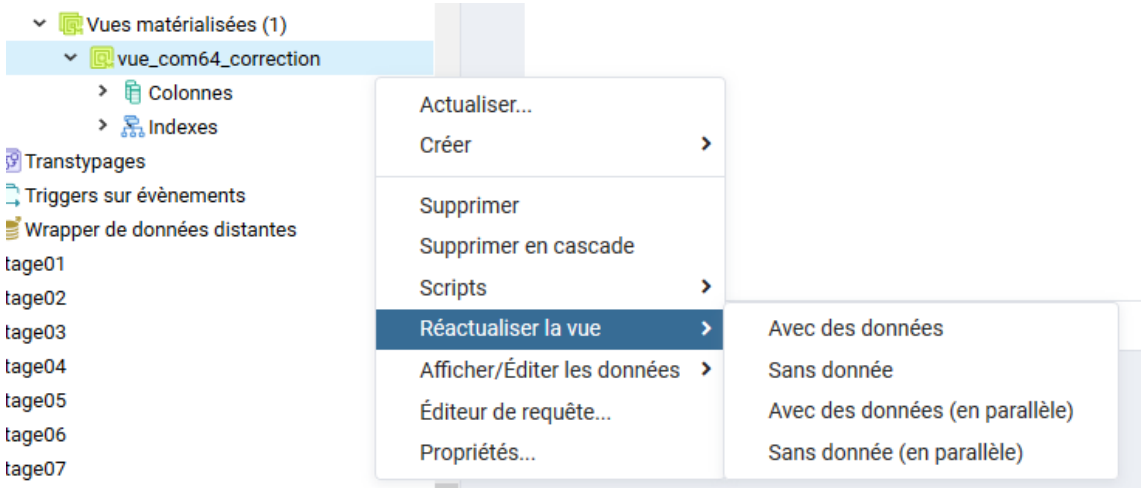
> Solution n°2 *(exercice p. 20)*

Sous DBManager :

vérifier que vous êtes connecté sur la base *stageXX* avec le rôle *stageXX*, puis créer la vue matérialisée avec la commande :

```
CREATE MATERIALIZED VIEW travail.vue_com64_correction AS
SELECT * FROM travail.communes64
WHERE communes64."Statut" IN ('Chef-lieu canton', 'Préfecture', 'Sous-préfecture') ;
```

Noter que sous PgAdmin un clic droit sur la vue matérialisée fait apparaître la possibilité de réactualiser la vue avec les données. L'option '*en parallèle*' (CONCURRENTLY) permet de ne pas verrouiller les sélections sur la vue pendant le rafraîchissement.



En pratique on n'utilisera des vues matérialisées que lorsque l'on est confronté à des problèmes de performances.

> Solution n°3 *(exercice p. 31)*

solution :

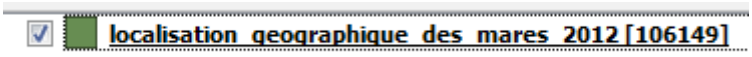
```
SELECT
        (CASE WHEN "Code_Arrondissement" = '1' THEN 'Bayonne'
        WHEN "Code_Arrondissement" = '2' THEN 'Oloron Sainte-Marie'
        WHEN "Code_Arrondissement" = '3' THEN 'Pau' END) AS "Arrondissement",
        SUM("Population") as "Population en milliers"
FROM travail.communes64
GROUP BY "Code_Arrondissement"
```

> Solution n°4 *(exercice p. 45)*

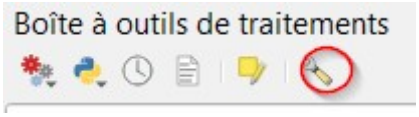
Charger `localisation_geographique_des_mares_2012.SHP`

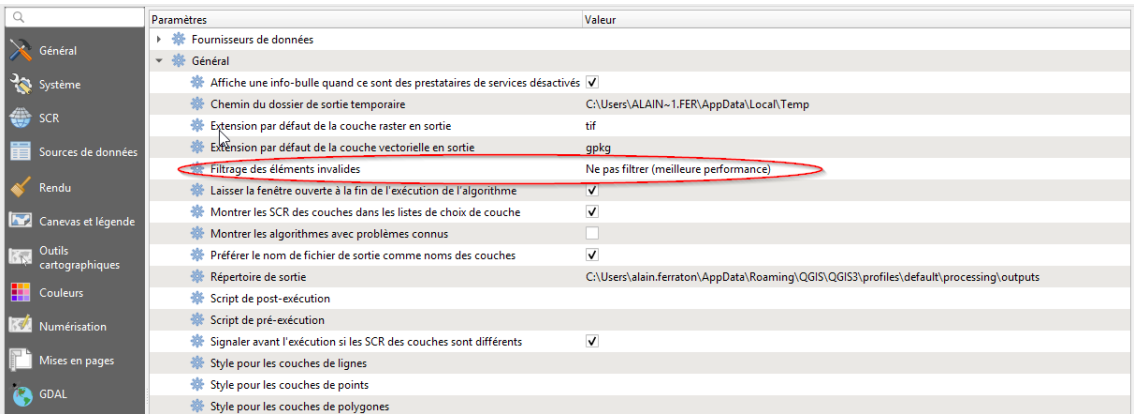
faire clic droit dans le gestionnaire de couche -> `Montrer le décompte des entités.`

Il y a 106149 entités :

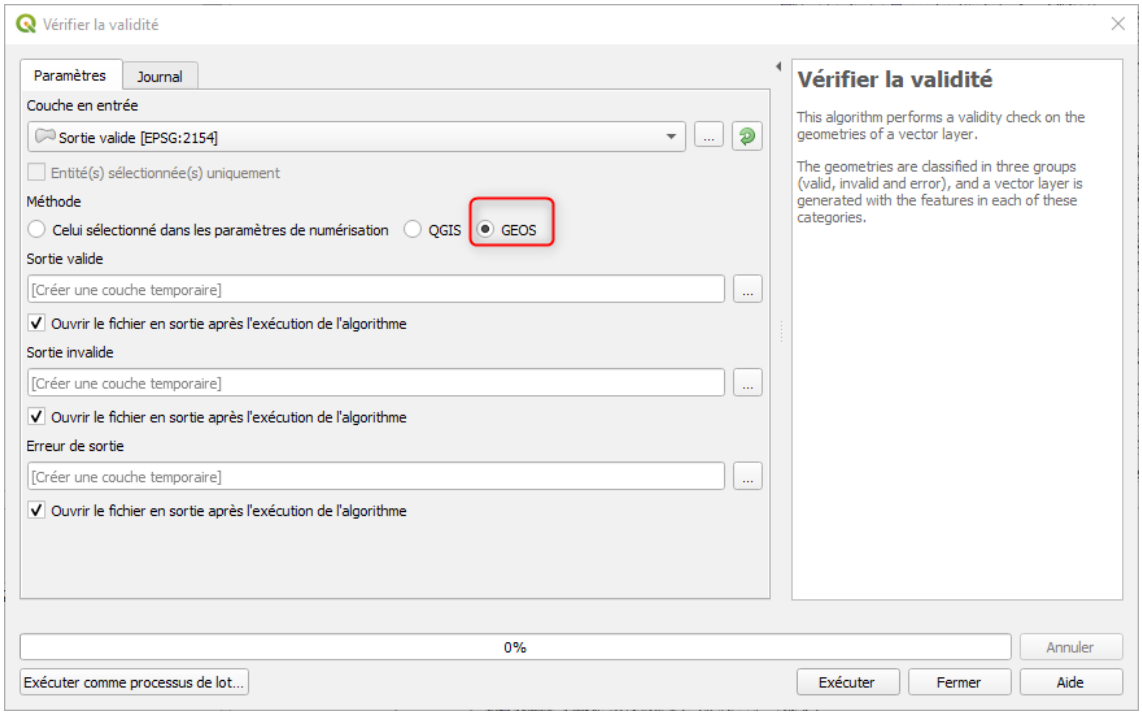


Paramétrer la boîte à outils de traitements pour ne pas filtrer les éléments invalides...

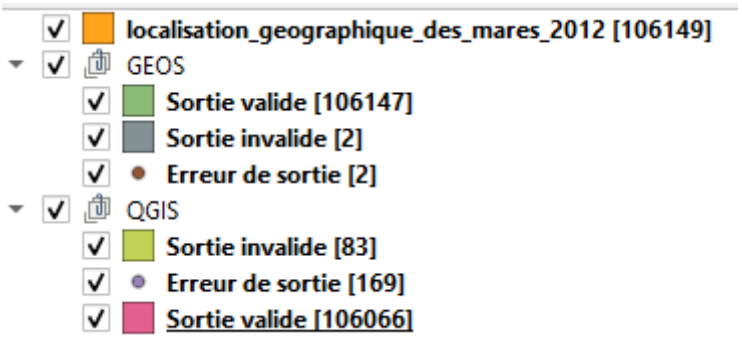




Utiliser l'algorithme *vérifier la validité* avec la méthode GEOS, puis QGIS.



Réorganiser les couches en utilisant clic droit -> `grouper la selection` et `montrer le décompte des entités`



Pour identifier le type d'erreur utilisons la table attributaire de la couche *erreur de sortie* dans les deux cas.

(faire un zoom sur l'étendue de la couche erreur de sortie et constater qu'il y a des points aberrants)

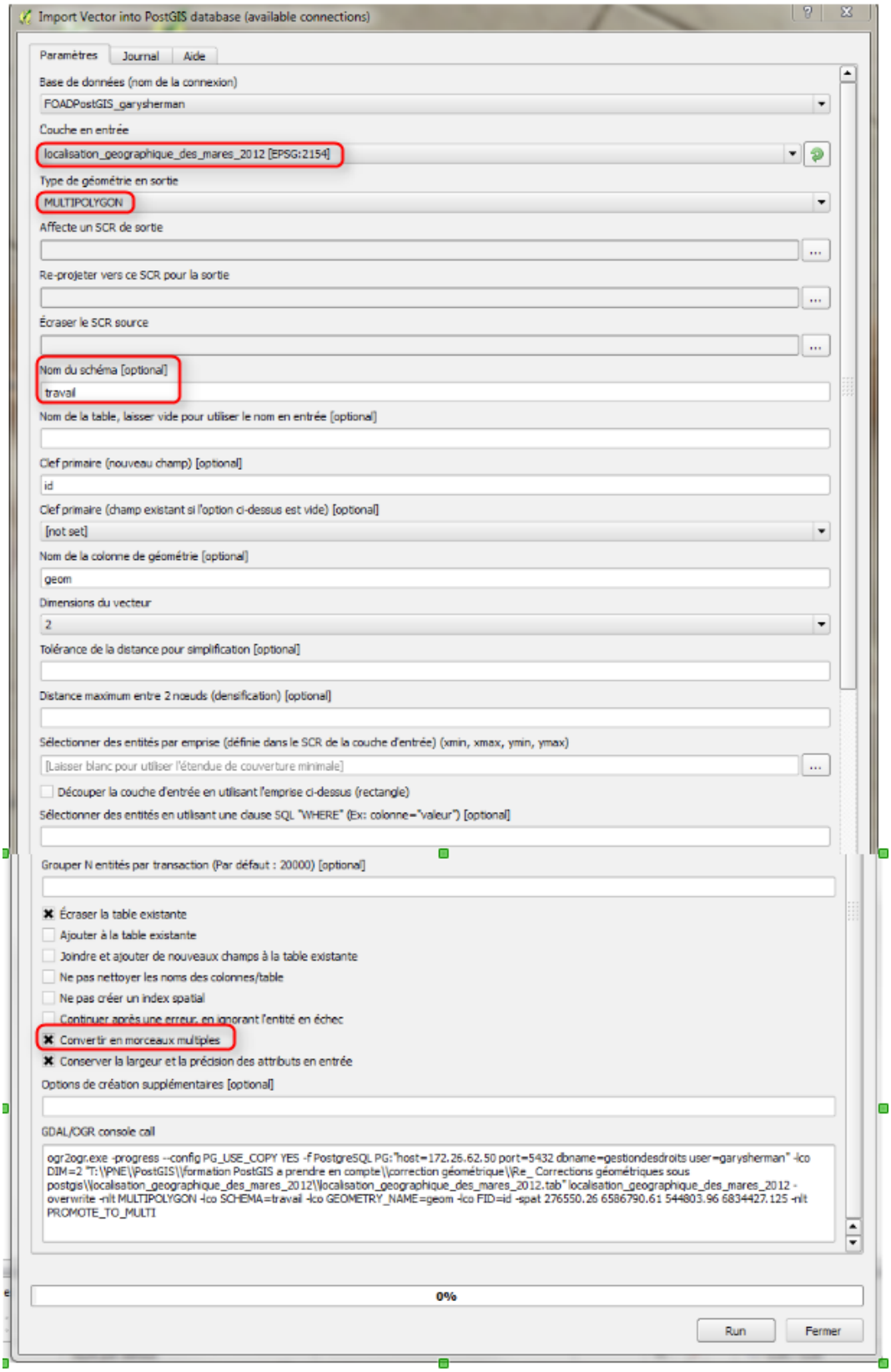
avec la méthode GEOS, les erreurs détectées sont :

auto-intersection et *trop peu de points dans la géométrie*

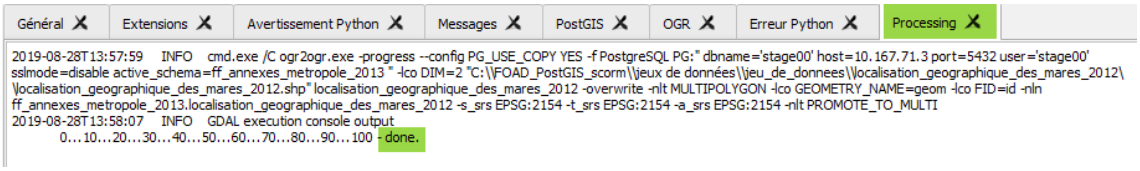
avec la méthode QGIS les erreurs détectées sont :

nœud en double et *la géométrie à 1 erreurs*.

Utilisons l'algorithme d'import de gdal (ou export avec QGIS 3) pour charger la table dans le schéma *travail* de la base *stageXX*


Import Vector into PostGIS database (available connections)
Paramètres
Journal
Aide
Base de données (nom de la connexion)
FOADPostGIS_garysherman
Couche en entrée
localisation_geographique_des_mares_2012 [EPSG:2154]
Type de géométrie en sortie
MULTIPOLYGON
Affecte un SCR de sortie
Re-projeter vers ce SCR pour la sortie
Écraser le SCR source
Nom du schéma [optional]
travail
Nom de la table, laisser vide pour utiliser le nom en entrée [optional]
Clef primaire (nouveau champ) [optional]
id
Clef primaire (champ existant si l'option ci-dessus est vide) [optional]
[not set]
Nom de la colonne de géométrie [optional]
geom
Dimensions du vecteur
2
Tolérance de la distance pour simplification [optional]
Distance maximum entre 2 noeuds (densification) [optional]
Sélectionner des entités par emprise (définie dans le SCR de la couche d'entrée) (xmin, xmax, ymin, ymax)
[Laisser blanc pour utiliser l'étendue de couverture minimale]
Découper la couche d'entrée en utilisant l'emprise ci-dessus (rectangle)
Sélectionner des entités en utilisant une clause SQL "WHERE" (Ex: colonne="valeur") [optional]
Grouper N entités par transaction (Par défaut : 20000) [optional]
Écraser la table existante
Ajouter à la table existante
Joindre et ajouter de nouveaux champs à la table existante
Ne pas nettoyer les noms des colonnes/table
Ne pas créer un index spatial
Continuer après une erreur, en ignorant l'entité en échec
Convertir en morceaux multiples
Conserver la largeur et la précision des attributs en entrée
Options de création supplémentaires [optional]
GDAL/OGR console call
ogr2ogr.exe -progress --config PG_USE_COPY YES -f PostgreSQL PG:"host=172.26.62.50 port=5432 dbname=gestiondesdroits user=garysherman" -lco DIM=2 "T:\\PNE\\PostGIS\\formation PostGIS a prendre en compte\\correction géométrique\\Re_ Corrections géométriques sous postgis\\localisation_geographique_des_mares_2012\\localisation_geographique_des_mares_2012.tab" localisation_geographique_des_mares_2012 -overwrite -nlt MULTIPOLYGON -lco SCHEMA=travail -lco GEOMETRY_NAME=geom -lco FID=id -spat 276550.26 6586790.61 544803.96 6834427.125 -nlt PROMOTE_TO_MULTI
0%
Run
Fermer

l'onglet processing du journal des messages doit renvoyer quelque chose comme :



sous Pgadmin vérifier que la couche *localisation_geographique_des_mares_2012* apparaît bien dans le schema travail de votre base.

La requête pour compter les géométries null et invalides peut s'écrire (on utilise UNION pour avoir le résultat en une seule fois) :

```
select 'invalid' as nb, count(*) from travail.localisation_geographique_des_mares_2012 where not st_isvalid(geom) union
select 'geonul' as nb, count(*) from travail.localisation_geographique_des_mares_2012 where geom is null
```

le résultat est le suivant :

Données | EXPLAIN

	nb text	count bigint
1	geon...	56
2	invalid	2

Corrigez la géométrie avec la requête suivante :

```
update travail.localisation_geographique_des_mares_2012 set geom = st_makevalid(geom) where not st_isvalid(geom)
```

constater que des messages d'erreur sont générés :

```
NOTICE: Too few points in geometry component at or near point 370462.88999999996 6673504.5749999993
ERREUR: Geometry type (GeometryCollection) does not match column type (MultiPolygon)
********** Erreur **********
ERREUR: Geometry type (GeometryCollection) does not match column type (MultiPolygon)
État SQL :22023
```

En effet dans certains st_makevalid génère un objet de type collection plutôt que multipolygon.

Pour corriger ce pb nous pouvons utiliser :

```
update travail.localisation_geographique_des_mares_2012 set geom = st_multi(st_CollectionExtract(ST_ForceCollection(st_makevalid(geom)),3)) where not st_isvalid(geom)
```

PostGIS renvoi maintenant :

```
NOTICE: Too few points in geometry component at or near point 370462.89000000001 6673504.5700000003
NOTICE: Self-intersection at or near point 359608.83000000002 6714918.75
```

Corrigeons maintenant les points doubles.

```
update travail.localisation_geographique_des_mares_2012 set geom = st_multi(st_simplify(geom,0))
```

Nous pouvons supprimer les entités avec géométrie nulle avec la requête suivante :

```
delete from travail.localisation_geographique_des_mares_2012 where geom is null;
```

Charger la couche dans QGIS et réexécuter *vérifier la validité* avec la méthode QGIS et GEOS

constater qu'il n y a plus d'erreurs au sens QGIS et GEOS.

a noter que l'algorithme '*Réparer les géométries' de QGIS*' corrige dans notre cas les anomalies détectés par la méthode GEOS mains ne corrige pas les noeuds en double (détecté par la méthode QGIS).

Remarque : Pas d'accent dans les chemins de fichiers.

Dans les algorithmes d'import sous PostgreSQL les noms des chemins ne doivent pas contenir d'accents.

Complément

Cet exercice constitue une introduction à la correction des géométries.

Vous pouvez trouver un document un peu ancien maintenant traitant du sujet *ici*[32]

Il existe plusieurs plugins intéressants dont 'Le nettoyeur 'de polygones)' de P. Desboeufs.

32 - http://www.geoinformations.developpement-durable.gouv.fr/verification-et-corrections-des-geometries-a3522.html