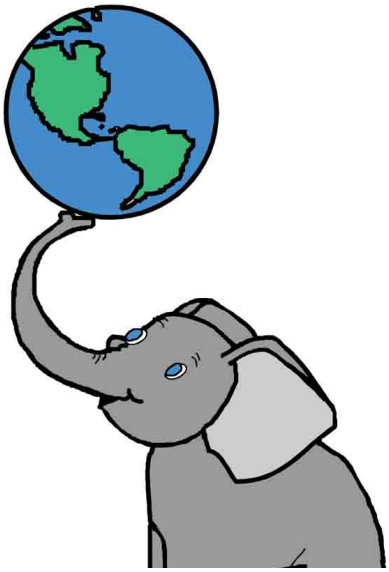
Aller plus loin



Version 1.4

Ministère de la Transition Ecologique
Licence ETALAB

Janvier 2022

Table des matières

Objectifs

Ce module va vous permettre de :

- découvrir l'utilisation des fonctions et triggers ;
- apprendre à optimiser les requêtes et à utiliser les tables et vues systèmes ;
- savoir intégrer un géostandard dans la base de données

Introduction

Le temps d'apprentissage de ce module est estimé à 3 heures, plus une heure pour la réalisation de l'exercice tutoré.

Il comporte :

- un exercice pas-à-pas ;
- un exercice en auto-correction ;
- un exercice "tutoré" dont vous devrez communiquer le résultat à vos tuteurs.

I fonctions et triggers

A. Triggers et fonctions

Un **trigger** ou déclencheur est associé à une table ou une vue. Il permet d'exécuter automatiquement des actions lorsque certains événements surviennent sur sa table ou sa vue de rattachement. Il peut :

- s'enclencher avant que l'événement ait lieu (**BEFORE**), pour par exemple vérifier ou modifier les données qui seront insérées ou mises à jour.
- s'enclencher après que l'événement ait lieu (**AFTER**), permettant par exemple de propager des mises à jour vers d'autres tables ou pour réaliser des tests de cohérence.
- Ou effectuer d'autres événements à la place de celui prévu initialement (**INSTEAD OF**) permettant par exemple de simuler une vue « modifiable » (voir cependant le paragraphe sur les vues modifiables dans la documentation de PostgreSQL). INSTEAD OF ne s'applique que pour les vues.

Un trigger exécute une fonction qui doit être créée au préalable.

PostgreSQL propose quatre types de fonction :

- *fonctions écrites en SQL*[1]
- *fonctions en langage procédural*[2] (PL/pgSQL, PL/Tcl, PL/Perl et PL/Python)
- *fonctions internes*[3]
- *fonctions en langage C*[4].

Nous allons voir quelques exemples d'utilisation en pgpsql.

La commande pour créer un trigger est :

```
CREATE [ CONSTRAINT ] TRIGGER nom { BEFORE | AFTER | INSTEAD OF }
{ événement [ OR ... ] }
```

1 - http://docs.postgresql.fr/current/xfunc-sql.html
2 - http://docs.postgresql.fr/current/xfunc-pl.html
3 - http://docs.postgresql.fr/current/xfunc-internal.html
4 - http://docs.postgresql.fr/current/xfunc-c.html

```
ON nom_table
[ FROM nom_table_referencee ]
[ NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY DEFERRED } ]
[ FOR [ EACH ] { ROW | STATEMENT } ]
[ WHEN ( condition ) ]
EXECUTE PROCEDURE nom_fonction ( arguments )
```

Cette commande permet :

- d'associer le trigger à sa table ou sa vue
- d'indiquer son type (BEFORE, AFTER, ou INSTEAD OF)
- de définir les événements déclenchant (INSERT, UPDATE, DELETE, ou TRUNCATE, + CREATE, ALTER et DROP à partir de PostgreSQL 9.3)
- d'indiquer s'il se déclenche pour chaque ligne (FOR EACH ROW) ou une fois pour l'ensemble (FOR EACH STATEMENT).
- d'associer la fonction qui devra se déclencher.

Complément : Pour en savoir plus…

Il n'est pas possible de détailler dans ce cours le langage PL/pgSQL. On trouvera, par exemple *ici*[5] une introduction.

Le *support de formation sur PL/pgSQL*[6] , complété par *PL/pgSQL avancé*[7], proposé par la société Dalibo est également complet et une bonne source pour qui veut s'investir sur le sujet.

A compléter bien sûr *par la documentation officielle*[8].

B. Pas à pas : création d'un trigger

Dans la base *stageXX* (remplaçer XX par votre numéro de stagiaire), créer la table *zonage* dans le schema *travail*, avec les spécifications suivantes

gid	serial not null (pk)
libelle	varchar(50)
surface_km2	float
geom	geometry(polygon,2154)

Sous PgAdmin taper :

```
CREATE TABLE travail.zonage (gid SERIAL NOT NULL PRIMARY KEY, libelle VARCHAR(50), surface_km2 FLOAT, geom geometry(polygon, 2154))
```

Nous souhaitons créer un trigger qui renseigne le champ *surface_km2* et mette en majuscule le libelle à chaque insertion ou modification d'une donnée de la table.

On trouvera *ici*[9] une description de l'utilisation de psql pour les triggers.

On retiendra en particulier l'existence des variables de type **RECORD** (enregistrement):

- **NEW** : contenant la nouvelle ligne de la base de données dans les triggers de

5 - https://blog.developpez.com/sqlpro/p10060/langage-sql-norme/postgresql_syntaxe_basique_des_fonctions
6 - https://public.dalibo.com/exports/formation/manuels/formations/devpg/devpg.handout.html#plpgsql-les-bases
7 - https://public.dalibo.com/exports/formation/manuels/modules/p2/p2.handout.html
8 - https://docs.postgresql.fr/current/plpgsql.html
9 - http://docs.postgresql.fr/9.4/plpgsql-trigger.html

niveau ligne (FOR EACH ROW). Elle est NULL dans un déclencheur de niveau instruction.

- **OLD** : contenant l'ancienne ligne de la base de données pour UPDATE ou DELETE. Elle est NULL dans un déclencheur de niveau instruction.

La fonction peut retourner NULL ce qui annulera l'insertion, la mise à jour ou la suppression de l'enregistrement ou OLD ou NEW ce qui validera la modification, l'ajout ou la suppression.

sous PgAdmin taper :

```
CREATE OR REPLACE FUNCTION travail.maj_zonage()
RETURNS "trigger" AS
$BODY$
BEGIN
NEW.surface_km2:= (ST_Area(NEW.geom)*0.000001);
NEW.libelle = UPPER(NEW.libelle);
RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql;
```

Conseil

Une astuce consiste à écrire systématiquement `CREATE or REPLACE` pour recharger rapidement la fonction en cas de changement du code.

taper maintenant :

```
CREATE TRIGGER maj_surface BEFORE INSERT OR UPDATE
ON travail.zonage
FOR EACH ROW
EXECUTE PROCEDURE travail.maj_zonage();
```

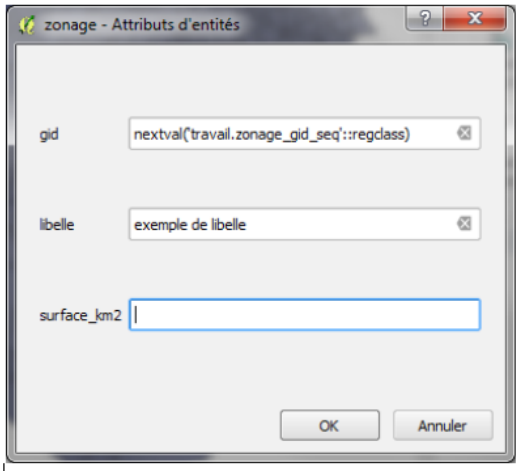
Constater que le déclencheur apparaît maintenant dans l'onglet info de la table travail.zonage sous DbManager dans QGIS (ou il peut être activé ou désactivé).

Déclencheurs

Nom	Fonction	Type	Activé
maj_surface (delete)	maj_zonage	Before INSERT UPDATE for each row	Oui (disable)

Activer tous les déclencheurs / Désactiver tous les déclencheurs

Sous QGIS ajouter la table *travail.zonage* dans le canevas, passer en mode modification et saisir des polygones sans renseigner *surface_km2* et en tapant un libelle en minuscule. Enregistrer et constater les changements dans la table des attributs.



	gid	libelle	surface_km2
0	17	EXEMPLE DE LIBELLE	151.1798252404...
1	18	ALAIN	404.5595960156...

C. 12 (tutoré) - Vue modifiable

Vous allez maintenant faire un exercice un peu plus compliqué (Vue modifiable avec l'utilisation de **INSTEAD OF**)

Question

Nous souhaitons gérer le positionnement des étiquettes de la table *zonage* et pouvoir adopter une étiquette personnalisée différente du libellé dans une nouvelle table *zonage_etiq* spécifique à cette fonctionnalité (pour ne pas mélanger les données de présentation des étiquettes dans les données attributaires de la table zonage).

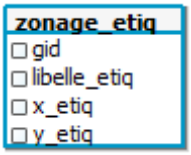
Cette nouvelle table aura pour attributs :

- gid : INTEGER (clef primaire)
- libelle_etiq : TEXT
- x_etiq : FLOAT
- y_etiq : FLOAT

Un zonage pourra avoir de 0 à 1 étiquette.

L'objectif est de proposer à l'utilisateur une vue que l'on appellera *vue_zonage* sur laquelle il pourra modifier le libellé et la position des étiquettes et à l'aide de trigger sur cette vue d'intercepter l' événement **UPDATE** pour faire en réalité les modifications dans la table *zonage_etiq*.

1) Créer la table *zonage_etiq* dans le schema *travail*.



2) créer la vue vue_zonage par jointure avec les spécifications suivantes :

SI zonage_etiq.gid est NULL ALORS vue_zonage.libelle_etiq = zonage.libelle SINON vue_zonage.libelle_etiq = zonage_etiq.libelle_etiq

SI zonage_etiq.gid est NULL ALORS vue_zonage.x_etiq = « Coord X de centroide de zonage » SINON vue_zonage.x_etiq = zonage_etiq.x_etiq

SI zonage_etiq.gid est NULL ALORS vue_zonage.y_etiq = « Coord Y de centroide de zonage » SINON vue_zonage.y_etiq = zonage_etiq.y_etiq

3) Créer une fonction maj_zonage_etiq() qui réalise l'algorithme suivant :

```
SI OLD.gid « existait déjà dans la liste des » zonage_etiq.gid ALORS
--> « mettre à jour » zonage_etiq
-----> zonage.libelle_etiq avec le nouveau libelle en majuscule
-----> zonage.x_etiq avec le nouveau x_etiq
----->zonage.y_etiq avec le nouveau y_etiq
SINON
--> « insérer » dans zonage_etiq un nouvel enregistrement (avec les nouvelles valeurs)
```

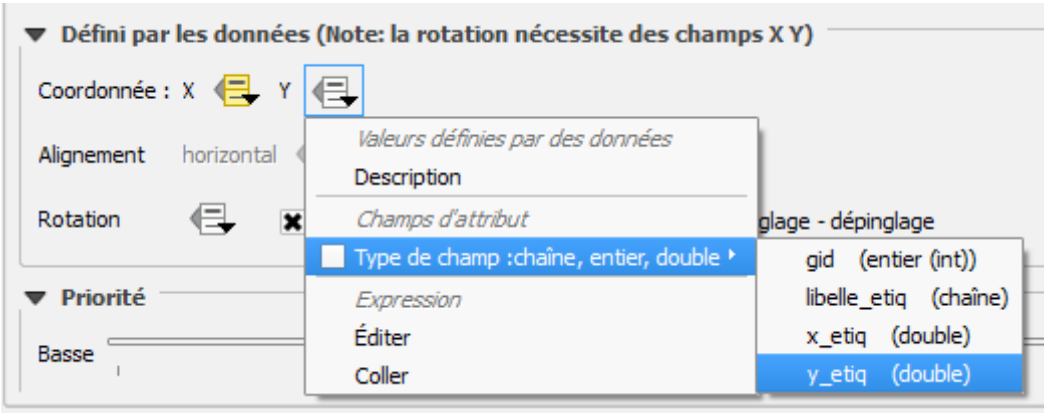
4) créer le trigger sur la vue vue_zonage.

5) faire un essai en modification d'étiquette en ajoutant la vue vue_zonage au canva sous QGIS.

vérifier les attributs de la table :

	gid	libelle_etiq	x_etiq	y_etiq
0	17	EXEMPLE DE LIBE...	199270.07085484	6274830.781156...
1	18	ALAIN	166298.2968326...	6267408.563627...

paramétrer l'emplacement des étiquettes dans les propriétés en indiquant les champs x_etiq et y_etiq



étiqueter la couche avec *libel_etiq*

basculer en mode édition et personnaliser des étiquettes avec les boutons 

enregistrer les modifications et vérifier le contenu de la table attributaire.

Envoyer vos différentes requêtes SQL aux tuteurs :

1. requête de création de la table zonage_etiq
2. requête de création de la vue vue_zonage
3. requête de création de la fonction maj_zonage_etiq
4. requête de création du trigger

Important : On ne cherche pas ici à pouvoir faire de la création de nouvelles entités directement dans *vue_zonage*. C'est pourquoi on n'intercepte pas l'événement **INSERT**.

La création de nouvelles entités doit toujours se faire dans la couche *zonage*.

Indices :

*Les spécifications de la jointure seront réalisées par utilisation de **CASE WHEN... THEN ... ELSE** dans la clause SELECT.*

*la fonction inclura un **IF...THEN... ELSE... END IF**. Le test portera sur le fait que OLD.gid (l'ancien gid avant modification) est ou non présent dans la liste des gid existants dans zonage_etiq.gid. Cette liste sera obtenue par* `SELECT gid FROM travail.zonage_etiq`*. Pour vérifier l'appartenance on utilisera donc le test.*

```
IF OLD.gid IN (SELECT gid FROM travail.zonage_etiq)
```

*Le trigger utilisera l'événement **INSTEAD OF UPDATE***

D. 13 - triggers

Cet exercice complémentaire va vous permettre d'utiliser les triggers pour gérer la mise à jour des contours des communautés de communes.

Question

[*Solution n°1 p 43*]

Nous souhaitons créer une table *limites_cc* qui représente les contours des communautés de communes à partir des limites des communes de la table *commune64* et qui soit entièrement régénérée à chaque modification (INSERT, UPDATE, ou DELETE) dans les tables *structures* et *delegation*.

On pourra décomposer le problème en étapes :

1) trouver la requête qui créée un table permettant de visualiser le contours de CC à partir des limites de la table commune64 et des tables *structures* et *delegation*

2) créer une fonction *actualise_geofla* qui supprime la table si elle existe et la recrée en utilisant la requête du point 1

3) créer une procédure qui appelle la fonction *actualise_geofla* en cas de modification dans les tables *structures* et *delegation*.

Optimisation

II

A. L'optimisation des requêtes avec EXPLAIN

La performance des requêtes peut être affectée par un grand nombre d'éléments. Certains peuvent être contrôlés par l'utilisateur, d'autres sont fondamentaux au concept sous-jacent du système.

PostgreSQL réalise un plan de requête pour chaque requête qu'il reçoit. Choisir le bon plan correspondant à la structure de la requête et aux propriétés des données est absolument critique pour de bonnes performances, donc le système inclut un planificateur complexe qui tente de choisir les bons plans.

Optimiser les requêtes peut-être un art difficile, mais il est bon de connaître les rudiments.

Vous pouvez utiliser la commande [10] pour voir quel plan de requête le planificateur crée pour une requête particulière.

On pourra également lire le chapitre sur l'utilisation des *index*[11].

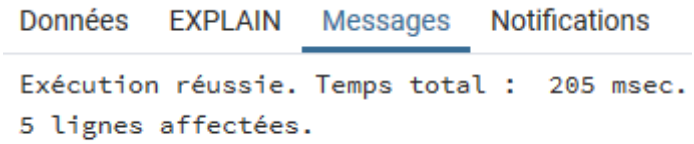
Le cours de Stéphane CROZAT[12] constitue également une bonne introduction non orienté sur la géomatique.

Exemple d'optimisation avec création d'index

Sous PgAdmin, exécuter l'ordre SQL suivant sur la table *FR_communes* du schéma *travail* de la base *stageXX* :

```
select * from travail."FR_communes" where "Code_Commune" = '023' and
"Statut" = 'Commune simple' and "Nom_Région" = 'AQUITAINE'
```

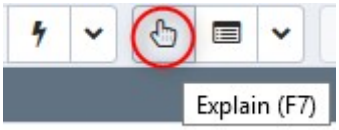
noter le temps qui apparaît en bas à droite ou dans l'onglet '*messages*' :



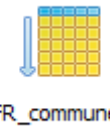
En fait si on re-execute plusieurs fois la requête on s'aperçoit d'une assez grande dispersion des résultats, mais l'ordre de grandeur reste le même.

10 - http://docs.postgresql.fr/9.4/performance-tips.html#using-explain
11 - http://docs.postgresql.fr/current/indexes-examine.html
12 - https://stph.scenari-community.org/bdd/opt1/co/optAC03.html

Dans notre cas le temps d'exécution moyen est d'environ 140 ms.
Lancer maintenant la requête avec l'option EXPLAIN :

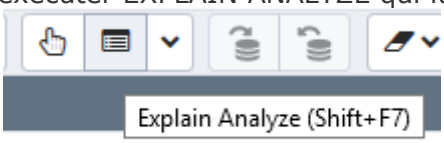


vous devez voir apparaître dans l'onglet 'EXPLAIN'



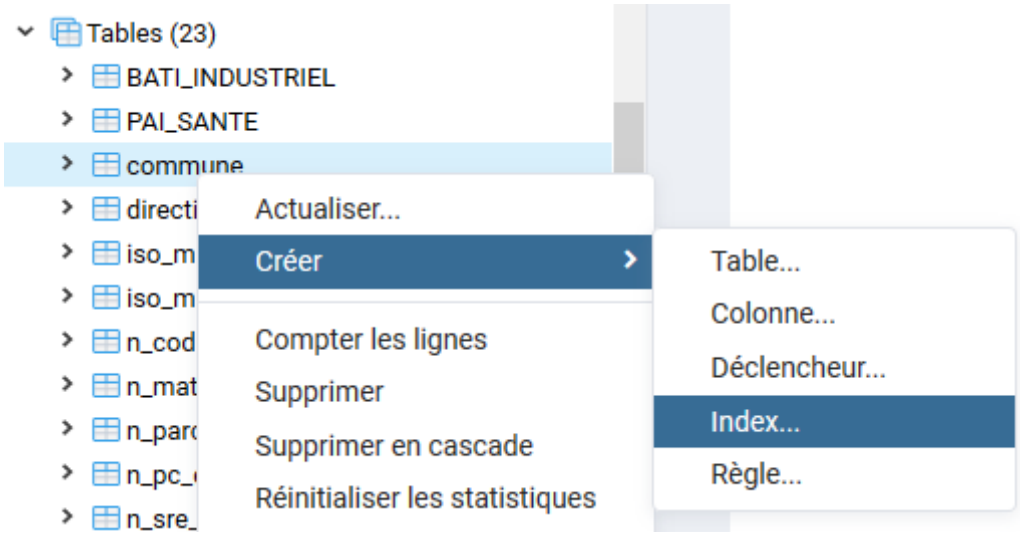
En positionnant le curseur sur l'icône des explications détaillées apparaissent :

Seq scan : indique que le moteur SQLde PostgreSQL execute un balayage séquentiel sur la table FR_communes
startup cost : est le coût estimé de lancement
Total cost : est le coût estimé du scan (les coûts sont estimés en *unité arbitraire*[13])
Plan rows : indique que 5 lignes sont retournées
Plan width : longueur estimée en octets
EXPLAIN fourni des données estimées.
On peut exécuter EXPLAIN ANALYZE qui lui va exécuter la requête et fournir les coûts mesurés



Ajoutons maintenant un index (de type btree) sur le champs"Statut"

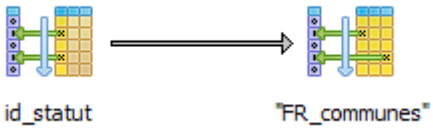
13 - http://docs.postgresql.fr/9.6/runtime-config-query.html#guc-seq-page-cost



puis, réexécuter la requête SQL

Normalement vous devez constater un temps d'exécution plus court (le cas échéant exécuter plusieurs fois la requête pour avoir un temps moyen).

Relancer le EXPLAIN



Le moteur de PostgreSQL commence par balayer la table d'index sur le statut pour un coût de 4.94, puis balaye uniquement les enregistrements de la table FR_communes correspondant à la condition sur statut pour un coût de 299.15 (coût total) - 4.94 (coût du balayage de la table d'index).

Le coût total est de 299.15 et donc très inférieur au coût sans l'index.

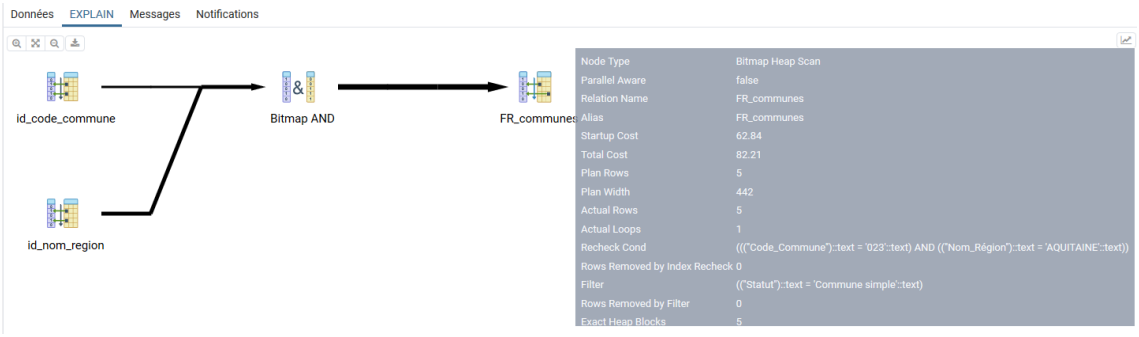
Il se peut cependant que le planificateur n'utilise pas ce nouvel index dans votre cas particulier.

en effet les index ne sont pas systématiquement utilisés. Voir *la documentation PostgreSQL*[14].

Ajouter maintenant un index supplémentaire sur le champ "Code_Commune" et un autre sur le champ "Nom_Région"

14 - http://docs.postgresql.fr/current/indexes-examine.html

relancer la requête et le EXPLAIN :



Si on survole l'icône finale à droite, on constate que le coût global est maintenant de 82.21

Le planificateur a choisi une stratégie :

1. balayage de l'index sur le champ *Nom_Région* en 57.66
2. balayage de l'index *id_commune* en 4.93
3. ET (Bitmap And) entre les deux résultats ci-dessus en 0
4. balayage des lignes de la table *FR_Communes* correspondantes en ajoutant le filtre sur le Statut en (82.21- 62.84) = 19.77

Pour un total de 82.21.

Le planificateur aurait pu choisir une autre stratégie en inversant le rôle d'un ou de deux des index. Il base sa stratégie sur les données statistiques qu'il maintient sur les tables.

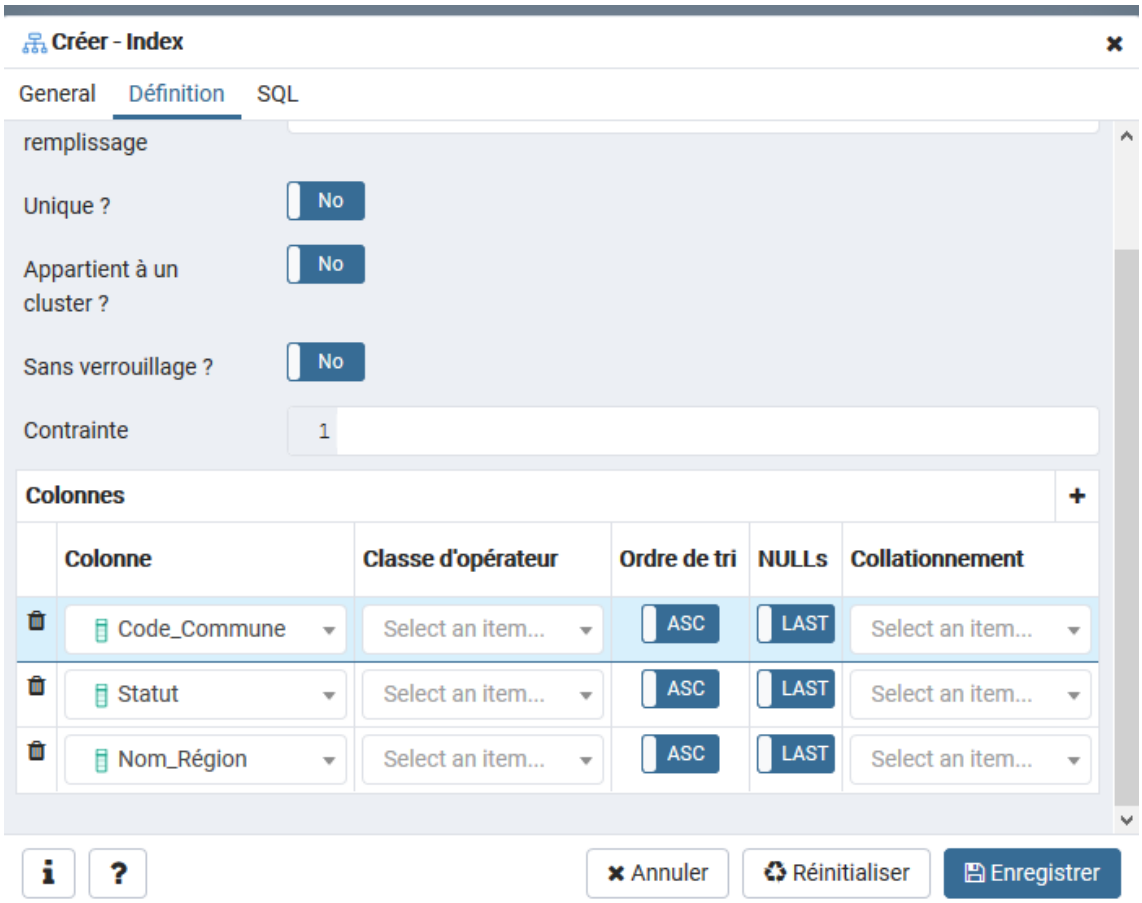
Une commande **VACCUM ANALYZE** que l'on peut lancer sur la base *stageXX* à partir du menu *Outils -> Maintenance*

permet de remettre à jour les statistiques sur les tables.

Tentons maintenant une autre stratégie d'optimisation en utilisant plutôt un index composite.

Supprimer les index créés sur la table (clic droit sur l'index -> supprimer)

créer un index qui porte sur les champs *statut* et *Nom_Région* et *Code_commune* :



Relancer l'analyse sur la requête :

La stratégie est maintenant de balayer l'index composite, puis de rechercher les enregistrements correspondant dans la table *FR_communes* pour un coût total de 18.13

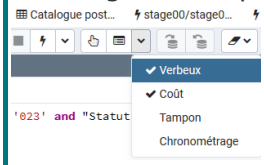
A comparer avec les résultats précédents.

Bien entendu il ne faut pas abuser de la création des index (occupation disque et ralentissement des requêtes en modifications), mais les utiliser a bon escient après avoir diagnostiqué les requêtes coûteuses les plus fréquentes.

Complément

EXPLAIN fourni des temps estimés, EXPLAIN ANALYZE fourni les temps observés (exécute la requête), ce qui est beaucoup plus précis mais coûteux en temps.

Il est également possible d'avoir plus de détails avec EXPLAIN ANALYZE VERBOSE



ou directement en SQL :

```
explain analyze verbose select * from travail."FR_communes" where "Code_Commune" = '023' and "Statut" = 'Commune simple' and "Nom_Région" = 'AQUITAINE'
```

qui renvoi :

"Index Scan using id_composite on travail."FR_communes" (cost=0.41..18.13 rows=5 width=442) (actual time=0.024..0.032 rows=5 loops=1)"

" Output: id, geom, "Code_Commune", "INSEE_Commune", "Nom_Commune", "Statut", "Population", "Superficie", "Altitude_Moyenne", "Code_Canton", "Code_dpt_arr_can", "Code_Arrondissement", "Code_dpt_arr", "Code_Département", "Nom_Département", "Code_Région", "Nom_Région""

" Index Cond: ((("FR_communes"."Statut")::text = 'Commune simple'::text) AND (("FR_communes"."Nom_Région")::text = 'AQUITAINE'::text) AND (("FR_communes"."Code_Commune")::text = '023'::text))"

"Planning time: 0.104 ms"

"Execution time: 0.056 ms"

Complément

PostgreSQL permet de mettre dans le fichier de log, les requêtes ayant plus pris plus d'un certain temps. Cette option se configure via l'entrée *log_min_duration_statement* du fichier de configuration PostgreSQL.

A noter également un *site* [15]qui permet de mettre en exergue les parties les plus coûteuses d'un plan d'analyse.

Dalibo propose *PEV2*[16] qui est utilisable en *ligne*[17] ou en téléchargement.

On trouvera également des explications très détaillées sur ce *site*[18] avec par exemple la description des *opérations*[19] des plans d'analyse.

Complément : Indexation des géométries dans des requêtes complexes.

Si on utilise des sous-requêtes qui calculent des géométries (st_intersection, st_buffer,...) pour ensuite les croiser avec des grosses tables (ex : département entier), il faut savoir que les géométries calculées ne sont pas indexées. Une solution d'optimisation peut alors être de créer une table résultat temporaire qui pourra être elle, être indexée :

`CREATE TEMP TABLE Matable AS...` , puis CREATE INDEX monIndex ON MaTable USING GIST (the_geom),

15 - http://explain.depesz.com/
16 - https://blog.dalibo.com/2021/01/26/pev2_whats_new.html
17 - https://explain.dalibo.com/
18 - http://use-the-index-luke.com/fr
19 - http://use-the-index-luke.com/fr/sql/plans-dexecution/postgresql/operations

au lieu de `WITH Matable AS (SELECT...)` qui ne permet pas de réaliser la phase d'indexation.

Exemple :

Si on créé un index sur « *Matable* »

```
---creation d'un index sur la géométrie de "matable"
CREATE INDEX matable_the_geom_idx ON matable USING gist (the_geom);
```

Puis qu'on utilise, par exemple, un syntaxe CTE :

WITH tampons AS (SELECT st_buffer(the_geom) as geom from ma_table)

SELECT * from tampons where ST_area(geom) <100 ;

L'index **n'est pas mobilisé** puisque le WHERE n'utilise pas la géométrie indexée the_geom, mais la géométrie calculée geom=st_buffer(the_geom)

Il pourrait donc être opportun de créer une table temporaire « *tampons* » plutôt que d'utiliser le WITH.

III

Utilisez la tables et vues systèmes

A. Présentation

Pour l'administrateur système il est utile de surveiller l'activité de la base de données. On trouvera une introduction dans la documentation de PostgreSQL sur le *monitoring*[20]. Le récupérateur de statistique propose des *vues statistiques standards*[21].

PostgreSQL dispose également d'un certains nombres de tables et vues systèmes dans les *catalogues*[22].

Dans cette formation nous explorerons que deux cas de figure simples :

- Lister les tables d'un schéma d'une base et changer le propriétaire
- Arrêter une requête longue

B. Modifier les propriétés des tables d'un schéma

La vue [23] que l'on peut trouver dans PgAdmin pour chaque base dans

catalogue → catalogue PostgreSQL → Vues → pg_tables

donne les informations sur chaque table de la base de données.

On peut par exemple extraire les tables du schema *travail* de la base *stageXX* avec la requête suivante :

```
select tablename from pg_tables where schemaname = 'travail';
```

Récupérer la liste des tables d'un schéma peut-être utile pour par exemple mettre à jour les droits sur toutes les tables ou changer de propriétaires.

Pour mémoire on peut par exemple changer le propriétaire d'un table avec la

20 - http://docs.postgresql.fr/current/monitoring.html
21 - http://docs.postgresql.fr/current/monitoring-stats.html#monitoring-stats-views-table
22 - http://docs.postgresql.fr/current/catalogs.html
23 - http://docs.postgresql.fr/current/view-pg-tables.html

commande

```
ALTER TABLE matable OWNER nouveauproprietaire ;
```

La requête :

```
select CONCAT('ALTER TABLE travail.',tablename, ' OWNER TO gary00;')
FROM pg_tables where schemaname = 'travail';
```

renvoi :

Données | EXPLAIN | Messages | Notifications

	concat text
1	ALTER TABLE travail.test3d OWNER TO gary00;
2	ALTER TABLE travail.surface_eau OWNER TO gary00;
3	ALTER TABLE travail.surface_eau_2d OWNER TO gary00;
4	ALTER TABLE travail.toto3d OWNER TO gary00;
5	ALTER TABLE travail.val_etat_eolienne OWNER TO gary00;
6	ALTER TABLE travail.val_etat_pc OWNER TO gary00;
7	ALTER TABLE travail.val_etat_demande_zde OWNER TO gary00;
8	ALTER TABLE travail.FR_communes OWNER TO gary00;
9	ALTER TABLE travail.invalidgeometry OWNER TO gary00;
10	ALTER TABLE travail.communes64 OWNER TO gary00;
11	ALTER TABLE travail.localisation_geographique_des_mares_2012 OW...

Que l'on peut exporter en script SQL en supprimant les double-quotes :

Le rechargement de ce script dans l'interface SQL donne :

stage00/stage00@DOIP FOAD PostgreSQL

Éditeur de requêtes | Historique

```
concat
ALTER TABLE travail.test3d OWNER TO gary00;
ALTER TABLE travail.surface_eau OWNER TO gary00;
ALTER TABLE travail.surface_eau_2d OWNER TO gary00;
ALTER TABLE travail.toto3d OWNER TO gary00;
ALTER TABLE travail.val_etat_eolienne OWNER TO gary00;
ALTER TABLE travail.val_etat_pc OWNER TO gary00;
ALTER TABLE travail.val_etat_demande_zde OWNER TO gary00;
ALTER TABLE travail.FR_communes OWNER TO gary00;
ALTER TABLE travail.invalidgeometry OWNER TO gary00;
ALTER TABLE travail.communes64 OWNER TO gary00;
ALTER TABLE travail.localisation_geographique_des_mares_2012 OWNER TO gary00;
ALTER TABLE travail.makevalidgeometry OWNER TO gary00;
ALTER TABLE travail.makevalidgeometry2 OWNER TO gary00;
ALTER TABLE travail.n_pc_eolien_r52 OWNER TO gary00;
ALTER TABLE travail.n_sre_eolien_s_r52 OWNER TO gary00;
ALTER TABLE travail.qgis_menubuilder_metadata OWNER TO gary00;
ALTER TABLE travail.n_parc_eolien_s_r52 OWNER TO gary00;
```

Il suffit de supprimer la première ligne et d'exécuter le script.

Attention à remettre des guillemets ("") pour les noms de table qui contiendraient des majuscules

Complément : Utilisation de fonctions utilitaires d'ASGARD

ASGARD[24] propose des *fonctions utilitaires*[25]. Par exemple la fonction *asgard_admin_proprietaire*[26] permet d'attribuer un schéma et tous les objets qu'il contient à un [nouveau] propriétaire

C. Arrêter une requête

Normalement vous devriez pour la mise au point des requêtes faire des essais sur des jeux limités, par exemple avec utilisation de **LIMIT 10**.

Il vaut mieux également avant de lancer une requête exécuter un **EXPLAIN** qui vous indiquera le temps estimé de la requête.

Il faut peut-être rajouter des index, peut-être revoir la requête, peut-être même si elle est trop complexe la décomposer pour la mettre dans une fonction.

Si toutefois vous souhaitez arrêter une requête trop longue. Il existe une table système qui recense toutes les requêtes en cous dans la base. C'est la table *pg_stat_activity*.

```
select * from pg_stat_activity;
```

24 - https://snum.scenari-community.org/Asgard/Documentation/#SEC_Introduction
25 - https://snum.scenari-community.org/Asgard/Documentation/#SEC_FonctionsUtilitaires
26 - https://snum.scenari-community.org/Asgard/Documentation/#CON_AsgardAdminProprietaire

va renvoyer un tableau avec une ligne par processus serveur, montrant les informations liées à l'activité courante du processus, comme l'état et la requête en cours

Extrait :

Données | EXPLAIN | Messages | Notifications

	datid oid	datname name	pid integer	usesysid oid	usename name	application_name text	client_addr inet	client_hostname text	client_port integer	backend_start timestamp with time zone	xact_start timestamp with time zone	query_start timestamp with time zone	state_change timestamp wit
1	134284	stage01	27448	16426	geoadm	pgAdmin 4 - DB:stag...	172.26.49.27	[null]	19971	2019-08-29 11:51:01.413856+02	[null]	2019-08-29 12:04:12.993564+02	2019-08-29 12
2	12407	postgres	27447	16426	geoadm	pgAdmin 4 - DB:post...	172.26.49.27	[null]	19970	2019-08-29 11:51:01.236618+02	[null]	2019-08-29 12:04:16.396233+02	2019-08-29 12
3	12407	postgres	1568	16472	stage00	pgAdmin 4 - DB:post...	172.26.49.157	[null]	56154	2019-08-29 13:33:40.665118+02	[null]	2019-08-29 13:33:40.829232+02	2019-08-29 13
4	18021	stage00	1545	16472	stage00	pgAdmin 4 - DB:stag...	172.26.49.157	[null]	56132	2019-08-29 13:33:21.380822+02	[null]	2019-08-29 13:35:09.916614+02	2019-08-29 13
5	18021	stage00	1548	16472	stage00	pgAdmin 4 - CONN:4...	172.26.49.157	[null]	56137	2019-08-29 13:33:24.378553+02	2019-08-29 13:40:55.696667+02	2019-08-29 13:40:55.696667+02	2019-08-29 13
6	18021	stage00	27241	16472	stage00	pgAdmin 4 - CONN:4...	172.26.49.157	[null]	51135	2019-08-29 11:47:19.040055+02	[null]	2019-08-29 11:54:16.966322+02	2019-08-29 11
7	18021	stage00	27209	16472	stage00	pgAdmin 4 - DB:stag...	172.26.49.157	[null]	51106	2019-08-29 11:47:08.882878+02	[null]	2019-08-29 11:54:17.025652+02	2019-08-29 11
8	12407	postgres	27204	16472	stage00	pgAdmin 4 - DB:post...	172.26.49.157	[null]	51104	2019-08-29 11:47:07.707636+02	[null]	2019-08-29 11:47:07.829365+02	2019-08-29 11

Le détail des colonnes est donné *ici* [27]:

pid nous permet d'identifier le processus serveur. La colonne *query* permet de vérifier la requête.

La colonne *client_addr* renvoi l'adresse IP des clients connectés (::1 signifie qu'il s'agit du poste local).

Il est possible d'avoir un sous-ensemble de ces informations, par exemple :

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
pg_stat_get_backend_activity(s.backendid) AS query
FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

va renvoyer quelque chose qui peut ressembler à :

Données | EXPLAIN | Messages | Notifications

	pid integer	query text
1	27204	
2	27209	SELECT oid, format_type(oid, NULL) AS typname FROM pg_type WHERE oid IN (25) ORDER BY oid;
3	27241	explain analyze verbose select * from travail."FR_communes" where "Code_Commune" = '023' and "Statut" = 'Commune simple' and "Nom_Région" = 'AQUITAINE'
4	1548	SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
5	1545	SELECT oid, format_type(oid, NULL) AS typname FROM pg_type WHERE oid IN (26, 19, 23, 26, 19, 25, 869, 25, 23, 1184, 1184, 1184, 1184, 25, 25, 25, 28, 28, 25) ORDER BY ...
6	1568	SELECT 1
7	27447	
8	27448	/*pga4dash*/

La fonction `pg_cancel_backend()` nous permet de tuer un processus serveur.

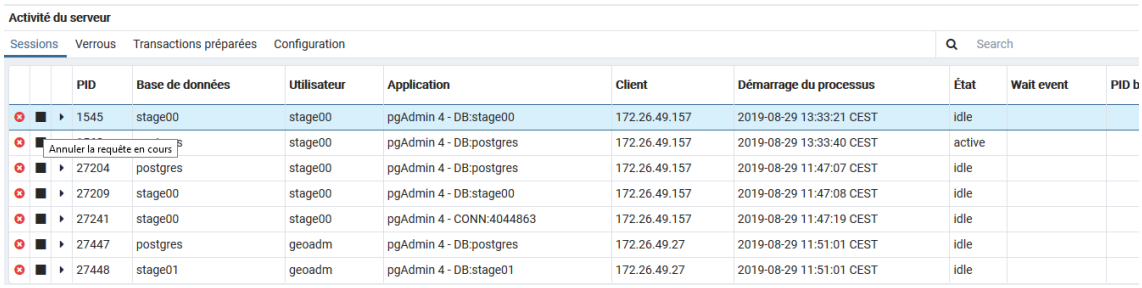
```
select pg_cancel_backend(6520) ;
```

Cette commande devrait retourner t (true). Si c'est bien le cas, dans quelques secondes, si vous relancez le `select * from pg_stat_activity;`, elle devrait avoir disparu. Si ce n'est pas le cas, c'est peut être parce qu'elle est déjà terminée, parce qu'elle a déjà été tuée par ailleurs, ou que vous n'avez pas les droits pour terminer une requête.

Complément : Utilisation de l'outil 'Etat du serveur' de PgAdmin

PgAdmin propose l'outil *tableau de bord*. Le cas échéant l'ajouter en avec un clic droit dans la barre des onglets

27 - http://docs.postgresql.fr/current/monitoring-stats.html#pg-stat-activity-view



	PID	Base de données	Utilisateur	Application	Client	Démarrage du processus	État	Wait event	PID b
	1545	stage00	stage00	pgAdmin 4 - DB:stage00	172.26.49.157	2019-08-29 13:33:21 CEST	idle		
			stage00	pgAdmin 4 - DB:postgres	172.26.49.157	2019-08-29 13:33:40 CEST	active		
	27204	postgres	stage00	pgAdmin 4 - DB:postgres	172.26.49.157	2019-08-29 11:47:07 CEST	idle		
	27209	stage00	stage00	pgAdmin 4 - DB:stage00	172.26.49.157	2019-08-29 11:47:08 CEST	idle		
	27241	stage00	stage00	pgAdmin 4 - CONN:4044863	172.26.49.157	2019-08-29 11:47:19 CEST	idle		
	27447	postgres	geoadm	pgAdmin 4 - DB:postgres	172.26.49.27	2019-08-29 11:51:01 CEST	idle		
	27448	stage01	geoadm	pgAdmin 4 - DB:stage01	172.26.49.27	2019-08-29 11:51:01 CEST	idle		

Les icône à gauche permettent :

Terminer la session : Tue le processus serveur sélectionné

Annuler la requête : annule une requête en cours

Utiliser ces fonctions avec précaution car elles peuvent conduire à perturber le fonctionnement du serveur et à devoir le redémarrer.

Complément

On trouve sur Internet des exemple de requêtes sur les tables systèmes qui peuvent être intéressantes pour un administrateur :

exemples :

Lister tous les index de toutes les tables :[28]

Lister toutes les contraintes[29]

Ce type de requête permet de procéder à des vérification après import de données en masse par exemple.

Autres exemples : *Scripts d'administration de la base partagée CeremaBase Centre-Est*[30]

28 - http://www.postgresql.org/message-id/432F450F.4080700@squiz.net
29 - http://solaimurugan.blogspot.fr/2010/10/list-out-all-forien-key-constraints.html
30 - Scripts d'administration de la base partagée CeremaBase Centre-Est

IV

Intégrer un géostandard dans la base de données

Nous allons voir comment intégrer un géostandard dans une base PostgreSQL/PostGIS.

Après un rappel sur la Covadis (désormais placée sous la commission des standards du *CNIG*[31]) et les géostandards, la manipulation va consister à, successivement :

- créer la base de données ou le schéma qui va accueillir les tables et les données ;
- créer les tables à partir du script SQL obtenu via le serveur de gabarits ;
- importer les fichiers SHP de données dans la base ;
- consulter les données importées avec QGIS.

Dans ce chapitre, nous nous appuierons sur l'exemple du **géostandard de l'éolien terrestre**.

A. Comprendre la Covadis et ses GéoStandards

Fondamental : Rôle de la COVADIS

La Covadis, Commission de validation des données pour l'information spatialisée, était une commission interministérielle mise en place par les ministères en charge de l'écologie, du logement et de l'agriculture pour **standardiser leurs données géographiques les plus fréquemment utilisées** dans leurs métiers.

La COVADIS a produit des **GéoStandards**

Depuis septembre 2019, *La COVADIS est placée sous la commission Données du CNIG*[32] , désormais la commission des standards.

31 - http://cnig.gouv.fr/?page_id=15232
32 - http://cnig.gouv.fr/?page_id=15232

Si vous n'êtes pas familier avec la Covadis et les géostandards, nous vous invitons maintenant à **lire la ressource** *Comprendre la Covadis et ses GéoStandards*[33].

B. Intégrer un géostandard dans une base PostgreSQL/PostGIS

Travailler avec des tables issues de l'implémentation d'un géostandard dans une vraie base de données relationnelle telle que PostgreSQL/PostGIS présente des avantages :

- mettre en œuvre des contraintes qui permettent d'assurer un contrôle sur les données et éviter, entre autres, les erreurs de saisie ;
- disposer de vues qui permettent de masquer aux utilisateurs la complexité de la base de données ;
- optimiser de manière transparente les traitements sur des volumes de données importants.

Par exemple, nous verrons que le géostandard sur l'éolien terrestre, au moyen d'un script SQL, définit des contraintes qui seront automatiquement implémentées avec les tables correspondantes dans la base de données.

Ce n'est pas le cas si l'on utilise des couches SHP ou TAB qui ont été directement importées dans une base de données, puisque les contraintes définies dans le standard n'auront pas été prises en compte.

Méthode : Création de la base de données ou du schéma avec pgAdmin

Pour intégrer les tables issues d'un géostandard dans PostgreSQL/PostGIS, on peut :

- soit créer une nouvelle base de données et intégrer les tables dans un des schémas de cette nouvelle base ;
- soit utiliser une base existante et créer un schéma qui pourra être réservé aux nouvelles tables.

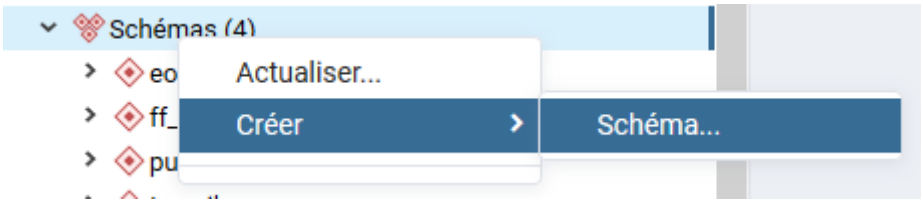
Le choix de la solution dépend de l'organisation choisie. En ce qui nous concerne, nous allons **créer un nouveau schéma qui sera dénommé *eolienterrestre***.

Lancer pgAdmin

Dans le navigateur d'objets, sélectionner votre base de données `stagexx` et avec un clic droit sur `Schémas` choisir `Créer -> Schéma...`

Le dénommer `eolienterrestre` et vous en faire le propriétaire

33 - http://www.territoires-ville.cerema.fr/comprendre-la-covadis-et-ses-geostandards-a1463.html



Méthode : Obtenir le script SQL de création des tables à partir du serveur de gabarits

Le serveur de gabarits *http://geostandards.developpement-durable.gouv.fr/*[34] permet d'obtenir des informations sur les géostandards élaborés par la Covadis.

Notamment, il est possible d'obtenir les tables de gabarit au format Mapinfo ou Shape : onglet **`Téléchargement`** du standard considéré.

34 - http://geostandards.developpement-durable.gouv.fr/

Par contre, pour les tables à intégrer dans une base de données PostgreSQL/PostGIS, il faut passer par un script SQL de création de tables qui sera exécuté avec pgAdmin.

La mise à disposition de ce script n'est pas fourni dans l'état actuel.

On utilisera le fichier de script SQL suivant :

Il est intéressant de consulter le contenu de ce script qui est composé principalement de commandes DROP, CREATE, INSERT que l'on a étudiées dans le module précédent.

Méthode : Création des tables à partir du script SQL

Dans pgadmin, connectez-vous au serveur sur la base stageXX (XX étant votre numéro de stagiaire) :

Remarquer que les ordres CREATE, INSERT n'indiquent pas le nom du schéma dans lequel les données vont être intégrées. Ce sera donc par défaut dans le premier schéma de la variable *search_path* (que nous avons vu au début de cette formation).

L'ordre SQL

```
SHOW search_path;
```

Affiche le contenu de la variable :

"$user",public

Nous n'avons pas crée de schéma portant le nom du rôle de connexion, donc par défaut les données seront créés dans le schéma *public*.

Modifions pour la durée de la session le *search_path* :

```
SET search_path TO eolienterrestre, public;
```

puis exécuter le script SQL récupéré.

Attention : ce script supprimera le cas échéant toutes les tables de même nom avant de les recréer.

La requête ne renvoie aucun résultat, mais supprime les tables existantes, les recréée, créée les contraintes et index, et peuple les tables de valeurs codées.

nb : si le nom du schéma comporte des majuscules (ce qui n'est pas recommandé) il faut mettre son nom entre "" dans l'ordre

```
SET search_path TO "MonSchema", public;
```

stage00/stage00@DOIP FOAD PostgreSQL

Éditeur de requêtes | Historique

```
1
2
3   -- Tables
4   DROP TABLE IF EXISTS n_mat_eolien_p_r52 CASCADE;
5   DROP TABLE IF EXISTS n_pc_eolien_r52 CASCADE;
6   DROP TABLE IF EXISTS n_codoa_eolien_r52 CASCADE;
7   DROP TABLE IF EXISTS n_parc_eolien_s_r52 CASCADE;
8   DROP TABLE IF EXISTS n_zde_moa_r52 CASCADE;
9   DROP TABLE IF EXISTS n_zde_com_r52 CASCADE;
10  DROP TABLE IF EXISTS n_zde_eolien_s_r52 CASCADE;
11  DROP TABLE IF EXISTS n_sre_commune_favorable_r52 CASCADE;
12  DROP TABLE IF EXISTS n_sre_eolien_s_r52 CASCADE;
13  -- types énumérés
14  DROP TABLE IF EXISTS val_classe_precision;
15  DROP TABLE IF EXISTS val_booleen_liste;
16  DROP TABLE IF EXISTS val_etat_eolienne;
17  DROP TABLE IF EXISTS val_etat_pc;
18  DROP TABLE IF EXISTS val_etat_demande_zde;
```

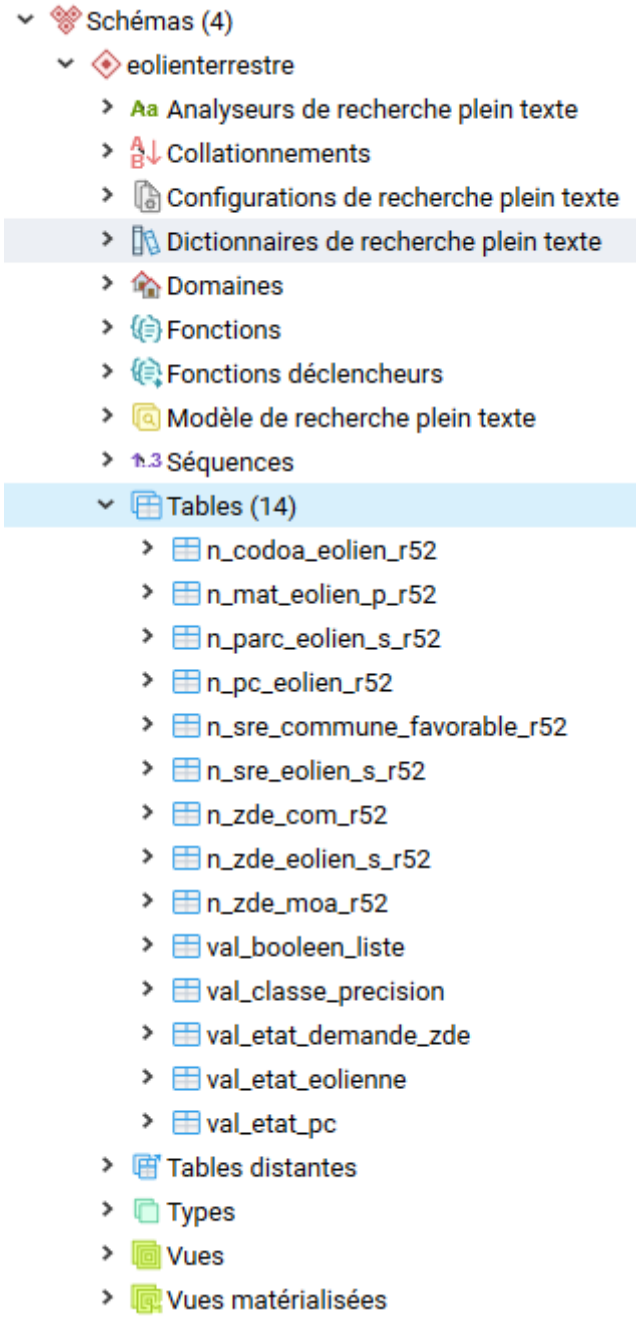
Données | EXPLAIN | Messages | Notifications

```
NOTICE:  la table « n_sre_eolien_s_r52 » n'existe pas, poursuite du traitement
CREATE INDEX

Requête exécutée avec succès en 183 msec.
```

Dans le schéma **eolienterrestre** de la base :

- rafraîchir (clic-droit) les tables pour afficher leur nombre exact : il y a normalement 9 tables n_ vides (dont 4 tables géométriques) et 5 tables val_ peuplées avec les valeurs codées...



- vérifier la table (val_etat_eolienne),
- afficher les données 

Données | EXPLAIN | Messages | Notifications

	code [PK] character (10)	libelle character (100)
1	CO	Construite
2	NCO	Non construite
3	DE	Démontée
4	AU	Autre

Rétablir le *search_path* :

```
SET Search_path TO "$user",public;
```

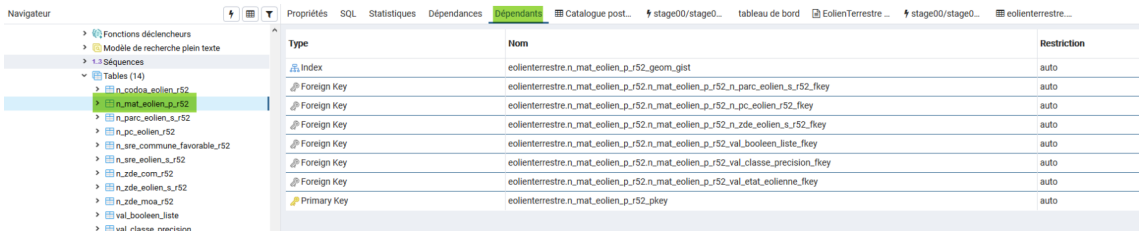
A noter qu'il suffit de se déconnecter et reconnecter pour rétablir le search_path (la modification par la commande SET n'étant valide que pour la session en cours).

Rappel : pour modifier de façon permanente un *search_path* pour un utilisateur il faut utiliser `ALTER USER`, exemple :

```
ALTER USER utilisateur_eolien SET search_path TO eolienterrestre, public ;
```

Complément : Visualisation des contraintes sur les tables

Dans pgAdmin, l'onglet **`dépendants`** de chaque table liste les contraintes associées à cette table, comme on le voit ci-dessous avec les contraintes de la table **`n_mat_eolien_p_r52`**.

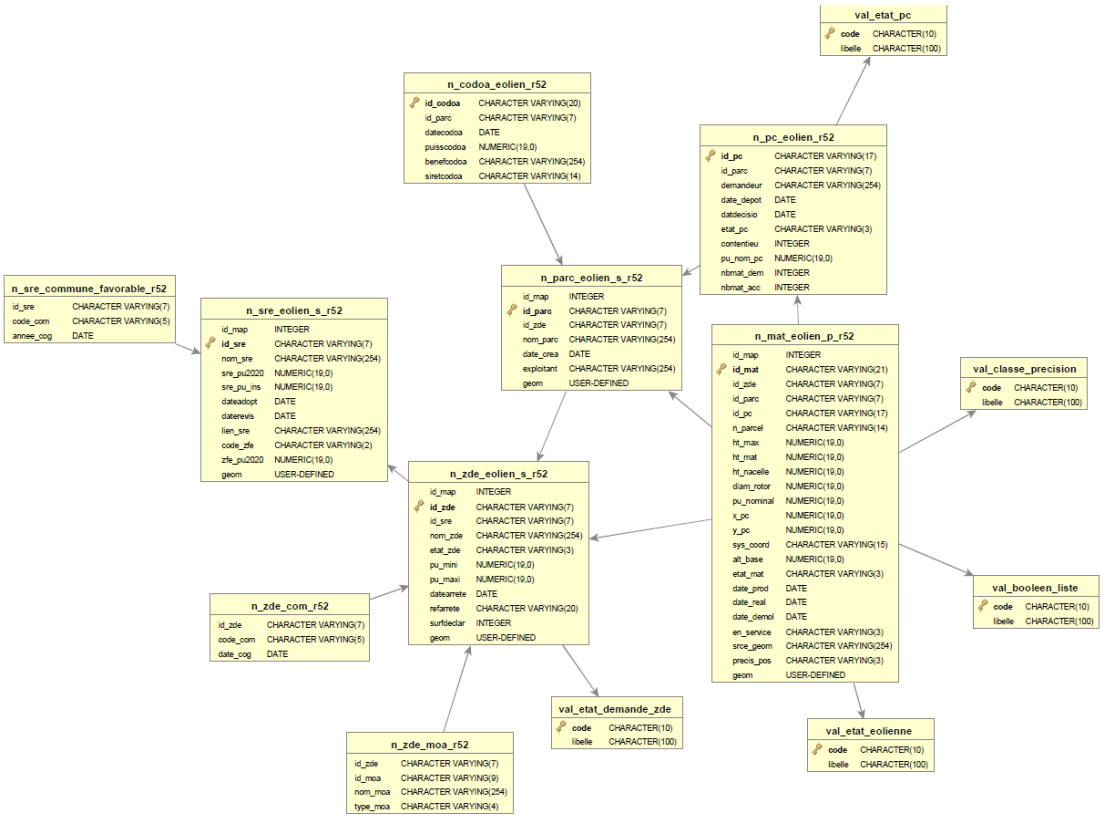


Des outils comme *DBVisualizer*[35] ou *pgModeler*[36] permettent de visualiser le schéma physique de la base en effectuant une rétro-ingénierie du modèle final obtenu après intégration du géostandard.

Cela permet de vérifier l'existence des liens entre les tables, créés lors de l'implémentation du géostandard via le script SQL.

Ci-dessous un exemple effectué sur le **géostandard de l'éolien terrestre.**

35 - https://www.dbvis.com
36 - http://pgmodeler.com.br



Cette représentation permet de vérifier le résultat de l'exécution du script SQL et l'existence des liens entre tables. Elle peut aussi être comparée au modèle contenu dans le géostandard.

C. Importer et visualiser des données

Pour les généralités sur l'importation de données, on se reportera au chapitre consacré à ce sujet dans le module 3 - gestion des bases.

Nous aborderons ici deux méthodes, la première avec ogr2ogr que nous ne mettrons pas en œuvre , et la seconde avec QGIS et une commande de géotraitement.

Les données à importer sont contenues dans les fichiers SHP du jeu de données de formation. Il s'agit de données réelles, produites par la DREAL des Pays-de-Loire, cataloguées sur le Géocatalogue. elles sont téléchargeables sur la plateforme SIGLOIRE : faire une recherche "éolien terrestre" sur *catalogue de SigLoire*[37]

Méthode : Import de fichiers SHP dans la base PostgreSQL avec ogr2ogr

Voir au préalable le chapitre sur l'utilisation d'ogr2ogr. - p.47

Les imports peuvent être réalisés par fichiers batchs, adaptables autant que de besoin (parfois l'encodage peut être en LATIN1 au lieu de l'UTF8), avec par exemple pour les parcs éoliens la syntaxe suivante :

```
echo off
```

37 - https://catalogue.sigloire.fr/geonetwork/srv/fre/catalog.search

```
set PGCLIENTENCODING=UTF8
ogr2ogr -append -update -f "PostgreSQL" -nlt PROMOTE_TO_MULTI PG:"dbname='EolienTerrestre' host='localhost' user='postgres' password='postgres'" n_parc_eolien_s_r52.shp
```

Ci-dessous le batch de création des permis de construire des éoliennes (la couche n_pc_eolien_r52) à partir de la base non géographique n_mat_pc_eolien_p_r52.dbf : cet exemple plus complexe tient à la particularité des données sources compatibles mais non conformes avec le standard, qui nécessitent dans la requête ogr2ogr une sélection SQL commençant par « SELECT UNIQUE ... » et le renommage de la couche de sortie avec -nln n_pc_eolien_r52 pour être conforme avec la structure créée dans la base de données

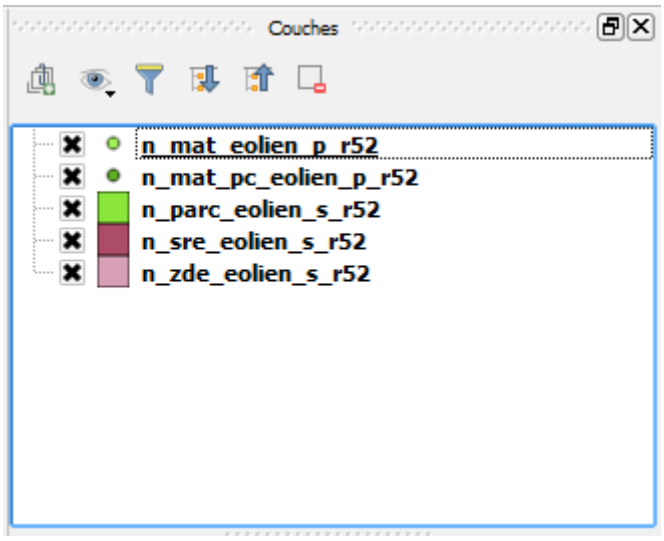
```
echo on
set PGCLIENTENCODING=UTF8
ogr2ogr -append -update -f "PostgreSQL" -select "SELECT UNIQUE id_pc, id_parc, demandeur, date_depot, datdecisio, etat_pc, contentieu, pu_nom_pc, nbmat_dem, nbmat_acc FROM n_mat_pc_eolien_p_r52" -nlt NONE -nln n_pc_eolien_r52 PG:"dbname='EolienTerrestre' host='localhost' user='postgres' password='postgres'" n_mat_pc_eolien_p_r52.dbf
```

Attention : les contraintes d'intégrité sur la base de données imposent également d'appeler les batchs dans cet ordre :

1. import des SRE
2. import des ZDE
3. import des parcs éoliens
4. import des mats et des permis de construire

Méthode : Import des données avec QGIS et la boîte à outils de géotraitement (Processing)

Dans QGIS, ouvrir les couches SHP du jeu de données formation.



Comme indiqué dans le *chapitre consacré à ce sujet dans le module 3* - p.48, dans QGIS :

- lancer **Traitement - Boîte à outils**
- cliquer, dans la liste des traitements proposés, sur **GDAL** puis **Divers vecteur**
- choisir le traitement **Exporter vers PostgreSQL(connections**

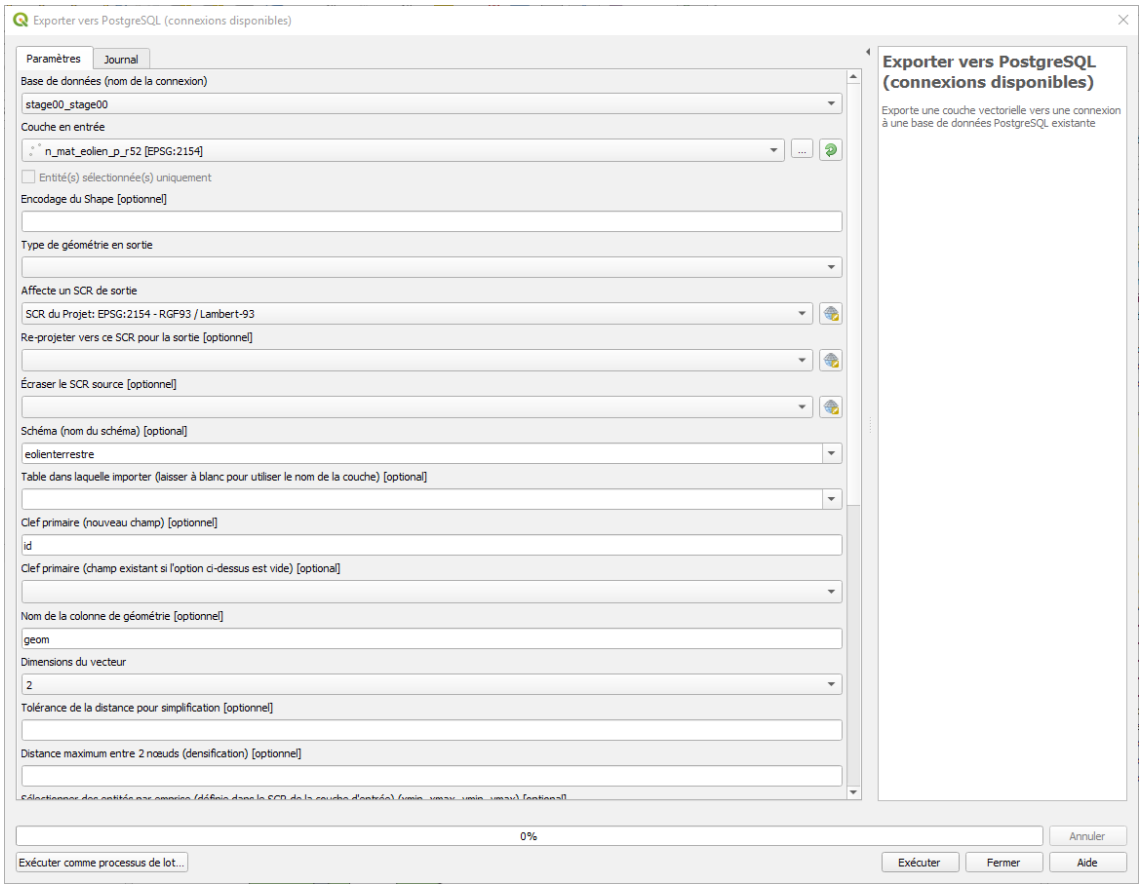
`disponibles)`

Renseigner les différents champs de la fenêtre de dialogue d'importation :

- **Base de données (nom de la connexion)** : choisir la connexion qui permet d'accéder à **`stagexx`**
- **Couche en entrée** : choisir, dans la liste déroulante, la couche ouverte dans QGIS que l'on souhaite envoyer vers la base de données. On choisit en premier n_sre_eolien_s_r52

  Type de géométrie : MULTIPOLYGON

  SCR de Sortie : SCR du Projet (EPSG : 2154)
- **Nom du schéma** : eolienterrestre (attention à l'orthographe)
- **Nom de la table** : la table de la base qui correspond à la couche ouverte dans QGIS (on peut laisser vide si les deux tables ont le même nom)
- **Clef primaire** : choisir id_sre
- **Plus bas** :
  - décocher `Écraser la table existante`
  - cocher `Ajouter à la table existante`
  - Cocher : `Convertir en morceaux multiples`



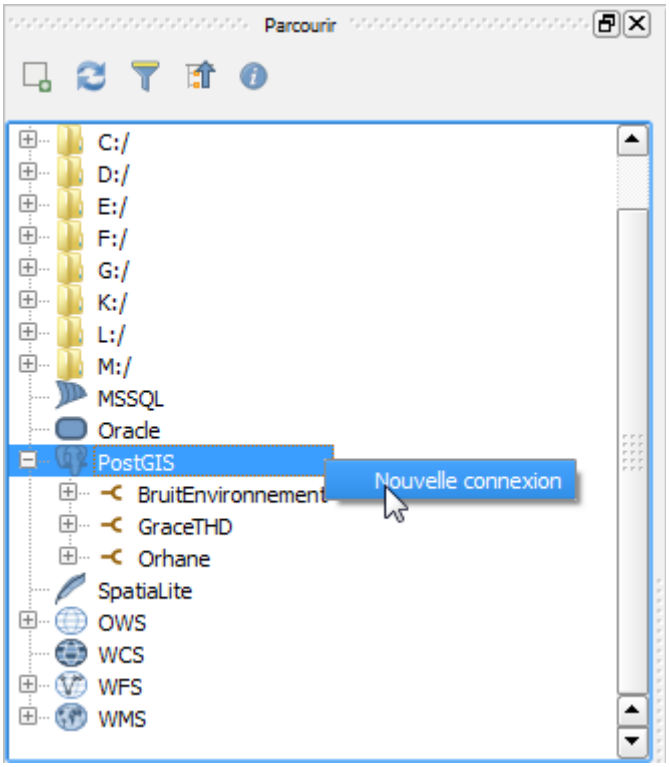
Valider en cliquant sur *Exécuter*

Vérifier l'importation des données, dans pgAdmin, en rafraîchissant la liste des tables : constater que la table contient bien une ligne.

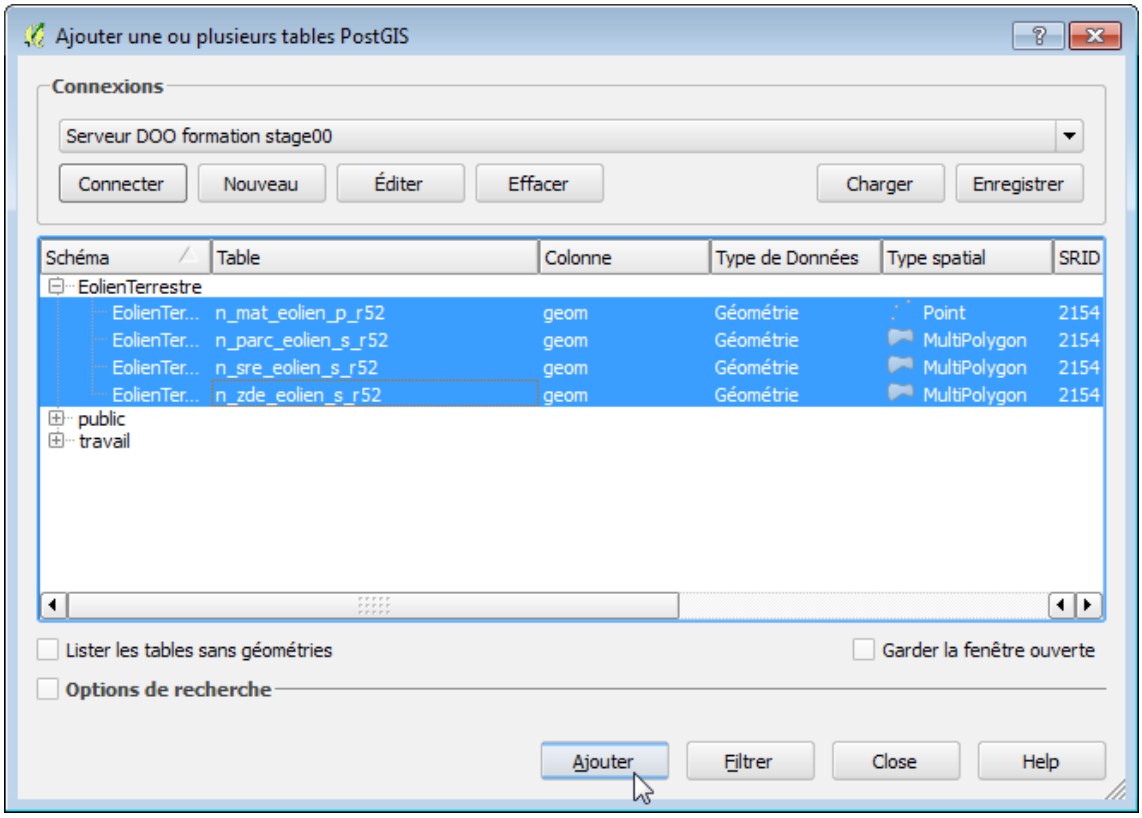
Méthode : Consultation des données de la base PostgreSQL avec Qgis

Dans QGIS, il y a 3 méthodes pour ajouter une couche vectorielle depuis une base PostgreSQL/PostGIS :

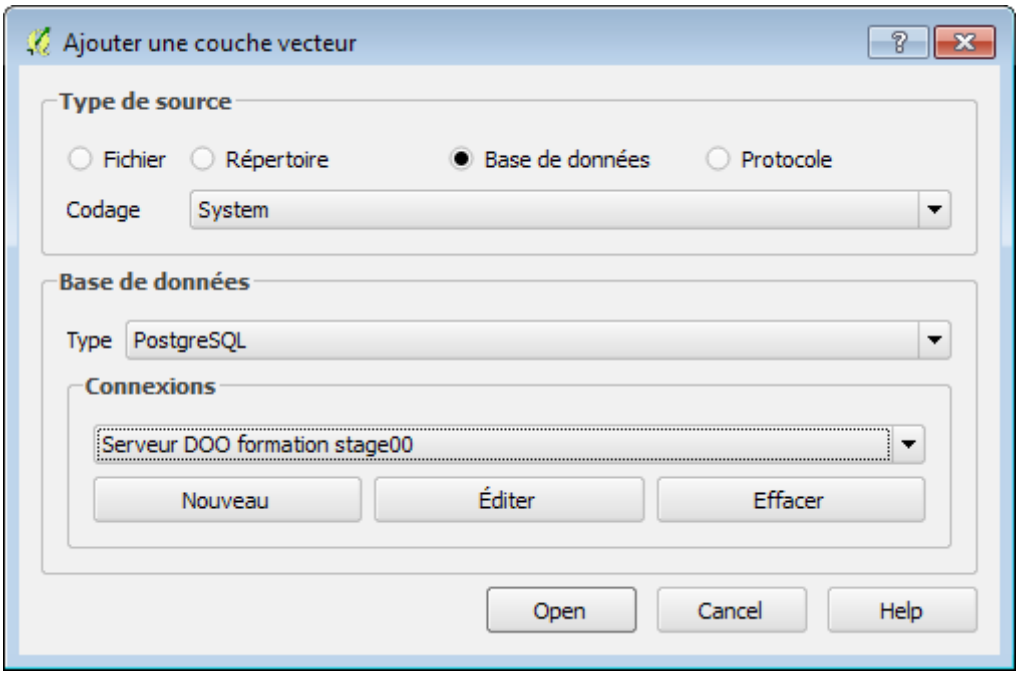
- soit dans le panneau navigateur ou parcourir : choisir la connexion dans la liste PostGIS. Puis le schéma, et les tables à ouvrir (double cliquer)

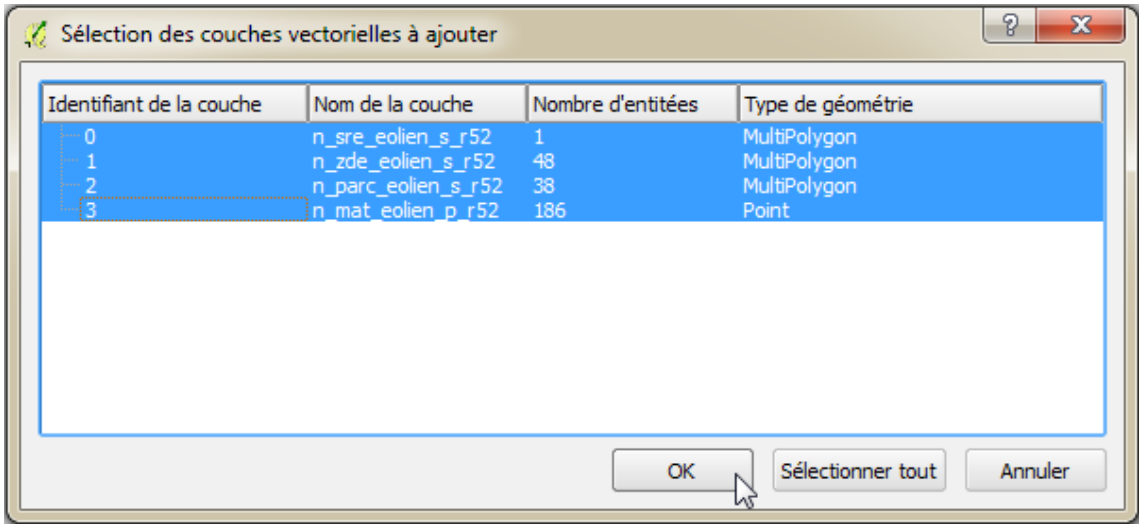


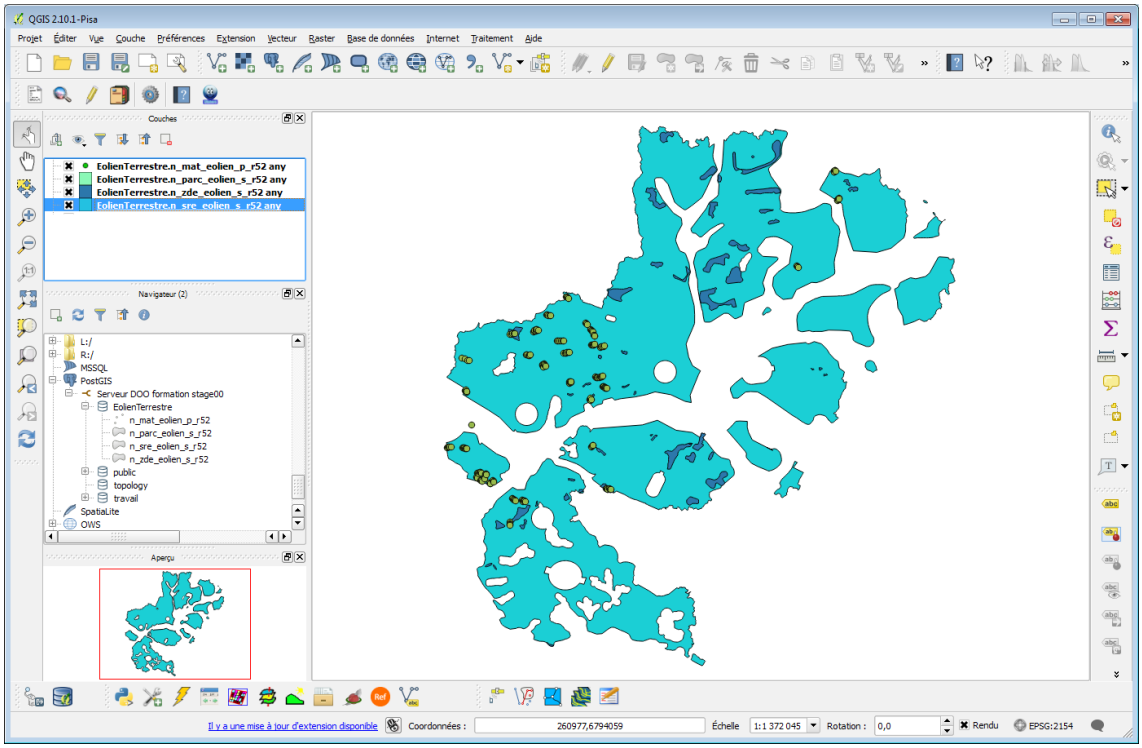
- soit avec le bouton **Ajouter une couche PostGIS**  : de la même manière, choisir la connexion, le schéma et la ou les tables, puis **Ajouter**



ou utiliser la commande **Ajouter une couche vecteur**  : cocher le type de source **Base de données**, puis choisir le type **PostgreSQL**. Sélectionner la connexion, puis les tables à ouvrir.




Sélection des couches vectorielles à ajouter
Identifiant de la couche
Nom de la couche
Nombre d'entitées
Type de géométrie
0
n_sre_eolien_s_r52
1
MultiPolygon
1
n_zde_eolien_s_r52
48
MultiPolygon
2
n_parc_eolien_s_r52
38
MultiPolygon
3
n_mat_eolien_p_r52
186
Point
OK
Sélectionner tout
Annuler


QGIS 2.10.1-Pisa
Projet
Éditer
Vue
Couche
Préférences
Extension
Vecteur
Raster
Base de données
Internet
Traitement
Aide
Couches
EolienTerrestre.n_mat_eolien_p_r52 any
EolienTerrestre.n_parc_eolien_s_r52 any
EolienTerrestre.n_zde_eolien_s_r52 any
EolienTerrestre.n_sre_eolien_s_r52 any
Navigateur (2)
L:/
R:/
MSSQL
PostGIS
Serveur DOO formation stage00
EolienTerrestre
n_mat_eolien_p_r52
n_parc_eolien_s_r52
n_sre_eolien_s_r52
n_zde_eolien_s_r52
public
topology
travail
SpatiaLite
OWS
Aperçu
Il y a une mise à jour d'extension disponible
Coordonnées :
260977,6794059
Échelle
1:1 372 045
Rotation :
0,0
Rendu
EPSG:2154

Solution des exercices

> Solution n°1 *(exercice p. 14)*

On peut utiliser une requête utilisant directement les 3 tables :

```
SELECT
  structure.nom_struct,
  structure.n_siren,
  structure.id,
  commune64.geom
FROM
  travail.commune64,
  travail.delegation,
  travail.structure
WHERE
  commune64."INSEE_Commune" = delegation.code_insee AND
  delegation.code_siren = structure.n_siren AND
  structure.nature_jur = 'CC';
```

Rajoutons un ORDER BY nom_struct :
cela donne un tableau (extrait) :

	nom_struct character varying(150)	n_siren character varying(25)	id integer	geom geometry(MultiPolygon,2154)
1	COMMUNAUTE DE COMMUNES D'AMIKUZE	246401616	52	01060000206A0800000100000001(
2	COMMUNAUTE DE COMMUNES D'AMIKUZE	246401616	52	01060000206A0800000100000001(
3	COMMUNAUTE DE COMMUNES D'AMIKUZE	246401616	52	01060000206A0800000100000001(
4	COMMUNAUTE DE COMMUNES D'AMIKUZE	246401616	52	01060000206A0800000100000001(
5	COMMUNAUTE DE COMMUNES D'AMIKUZE	246401616	52	01060000206A0800000100000001(
6	COMMUNAUTE DE COMMUNES D'AMIKUZE	246401616	52	01060000206A0800000100000001(
7	COMMUNAUTE DE COMMUNES D'AMIKUZE	246401616	52	01060000206A0800000100000001(

Il faut modifier ce code, car nous souhaitons le contour des CC... il faut donc faire un st_union() (voir le module SQL de QGIS perfectionnement) des géométries des communes appartement à une même CC et group by pour faire l'agrégation.

Cela donne :

```
SELECT
  structure.nom_struct,
  structure.n_siren,
  structure.id,
  st_union(commune64.geom)
FROM
  travail.commune64,
  travail.delegation,
  travail.structure
WHERE
  commune64."INSEE_Commune" = delegation.code_insee AND
  delegation.code_siren = structure.n_siren AND
  structure.nature_jur = 'CC'
  GROUP BY n_siren,structure.id,nom_struct
  ORDER BY nom_struct;
```

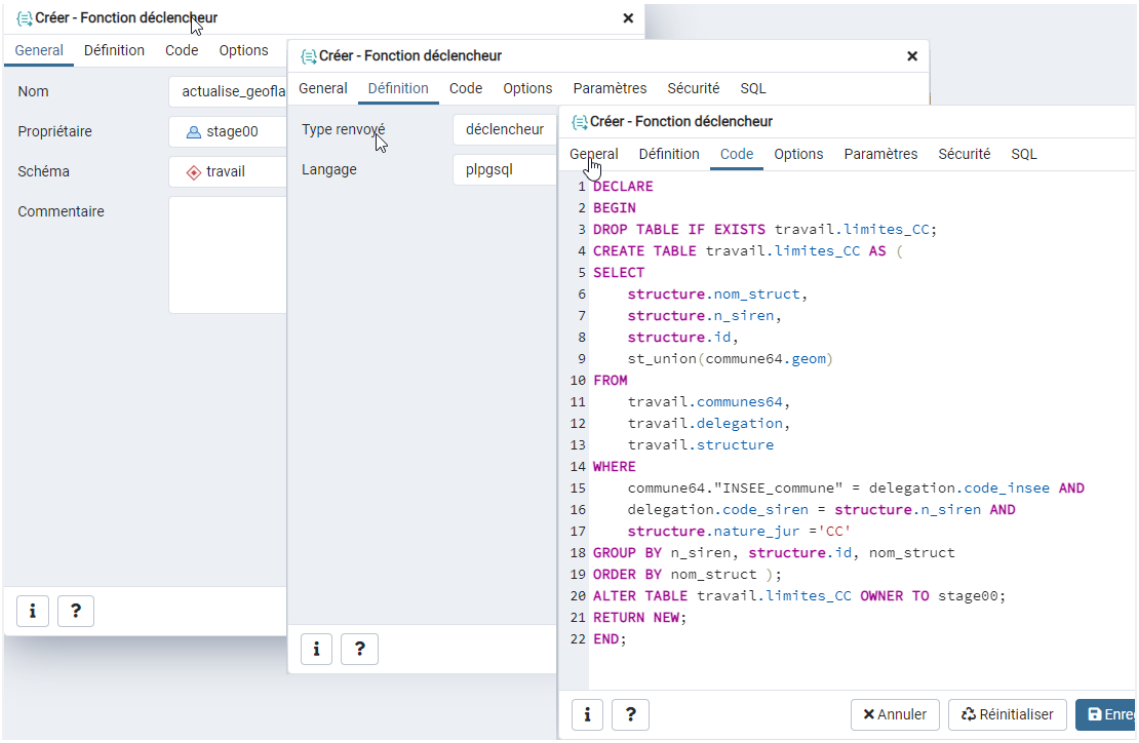
Si on veut créer la table on utilisera donc :

```
CREATE TABLE travail.limites_CC AS (
SELECT
row_number() over() as id_qgis,
scom.nom_struct,
scom.n_siren,
scom.id,
st_union(com.geom)::geometry(multipolygon,2154) as geom
FROM
travail.commune64 AS com,
travail.delegation AS dcom,
travail.structure AS scom
WHERE
com."INSEE_Commune" = dcom.code_insee AND
dcom.code_siren = scom.n_siren AND
scom.nature_jur = 'CC'
GROUP BY scom.n_siren,scom.id,scom.nom_struct
ORDER BY nom_struct);
```

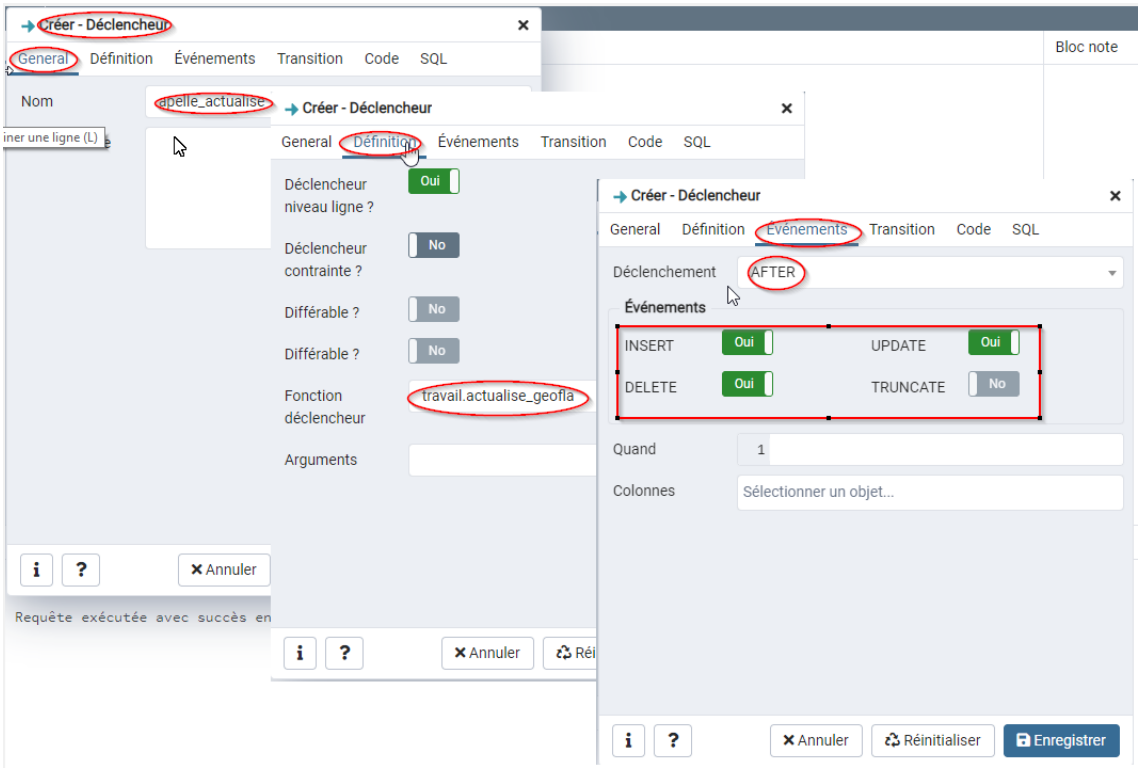
Noter le typage explicite de la géométrie fusionnée en MULTIPOLYGON et l'ajout d'une numérotation automatique des lignes avec le row_number() over().

Visualiser la table dans QGIS à partir de DBManager

Créer la *fonction déclencheur* actualise_geofla :



Créons maintenant deux triggers identiques sur les tables *structures* et *delegation* :



Sous QGIS, supprimer par exemple la communauté de communes de la vallée d'Ossau dans la table des structures.

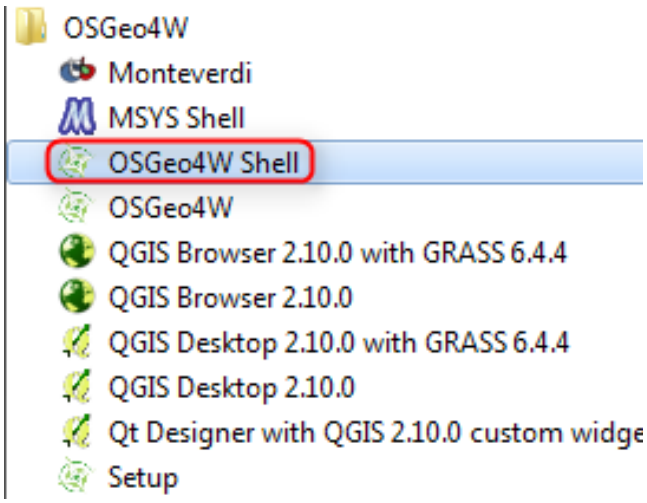
Pour rafraîchir la table *limites_cc* dans le canvas, faire par exemple un petit zoom panoramique. Vous devez voir disparaître la CC de la vallée d'Ossau.

Contenus annexes

- Import via ogr2ogr

GDAL (Geospatial Data Abstraction Library) est une bibliothèque libre permettant de lire et de traiter un très grand nombre de format d'images géographiques. Un sous-ensemble de cette bibliothèque est la bibliothèque OGR permettant d'accéder à la plupart des formats courants de données vectorielles. On trouvera une documentation en français sur *ce site*[38]. La documentation de référence sur ogr2ogr est *disponible ici.*[39]

Une installation de QGIS par OSGeo4W installe sur le poste local une console Shell permettant de lancer ogr2ogr (c'est également le cas pour les packages Ministère).



Si vous ne disposez pas du raccourci de lancement dans le menu vous pouvez lancer :

`C:\Program Files\QGIS 3.16\OSGeo4W.bat` (au besoin créer un raccourci sur le bureau).

On utilisera `ogr2ogr --version` pour récupérer le numéro de version installé :

```
C:\>ogr2ogr --version
GDAL 3.1.4, released 2020/10/20
```

Fermer la fenêtre shell en tapant `exit`.

Dans la pratique on utilisera surtout ogr2ogr dans des scripts sur le serveur (en poste local autant passer par les algorithmes de processing -menu traitement- qui offrent

38 - http://gdal.gloobe.org/
39 - http://www.gdal.org/ogr2ogr.html

une interface sous forme de boîte de dialogue).

Dans ce cas on utilisera le ogr2ogr installé avec les packages du serveur (*exemple pour Debian*[40]).

Il est possible de taper `ogr2ogr - - help` pour avoir une aide sur les paramètres.

Dans les options de ogr2ogr qui sont détaillées ici on notera en particulier :

- **`-f`** : format de sortie
- **`-overwrite`** : écrase les anciennes valeurs si existantes
- **`-append -update`** : ajoute des données sans écraser d'autres
- **`-nln`** : affecte nouveau nom à la table (si rien alors nom du fichier pour nom table)
- **`-s_srs`** : projection de la source
- **`-a_srs`** : assigne valeur de la projection en sortie
- **`-t_srs`** : transforme la projection (reprojection)
- **`-skipfailures`** : continue après un échec, ignorant l'objet en échec (en particulier si géométrie invalide).
- **`-nlt`** : exemple `-nlt MULTIPOLYGONE` pour imposer le type de géométrie.

Pour PostgreSQL on trouvera *ici la description*[41] des options spécifiques en particulier les Layer Creation Option (`-lco`).

Noter également l'importance de l'option `-config PG_USER_COPY YES` qui impose d'utiliser l'ordre `COPY` au lieu de `INSERT` et accélère beaucoup les traitements.

Exemple :

```
ogr2ogr -f "PostgreSQL" -append -nln "MaTable" -nlt MULTIPOLYGON -lco
GEOMETRY_NAME=the_geom -lco FID=gid -lco OVERWRITE=no -lco
SCHEMA=MonSchema -a_srs EPSG:2154 PG:"dbname='MaBase' host='MonHote'
port='PortDeHote' user='MonIdentifiant' password='MonMotDePasse'"
MonFichierSource
```

Pour un exemple plus concret, mettons que l'on veuille importer la table "COMMUNE_DENSITE.shp (fournie dans le jeu de données) sous le nom commune_densite

et que cette table soit disponible sous i:\

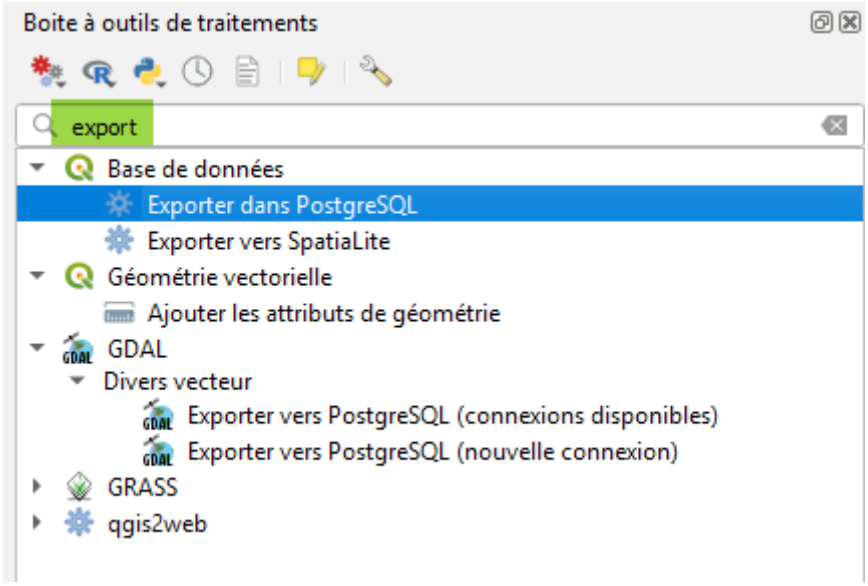
La commande sera :

```
ogr2ogr -f "PostgreSQL" -nln "commune_densite" -nlt MULTIPOLYGON -lco
GEOMETRY_NAME=geom -lco SCHEMA=travail -a_srs EPSG:2154
PG:"dbname='stage00' host='10.167.71.3' port='5432' user='stage00'
password='stage00'" i:/COMMUNE_DENSITE.SHP
```

- Import via Processing (menu traitement)

Un autre moyen d'importer une couche dans PostGIS est d'utiliser les algorithmes d'import vers PostGIS disponibles dans *Processing* (Boîte à outils de traitements)

40 - https://packages.debian.org/stable/gdal-bin
41 - http://www.gdal.org/drv_pg.html

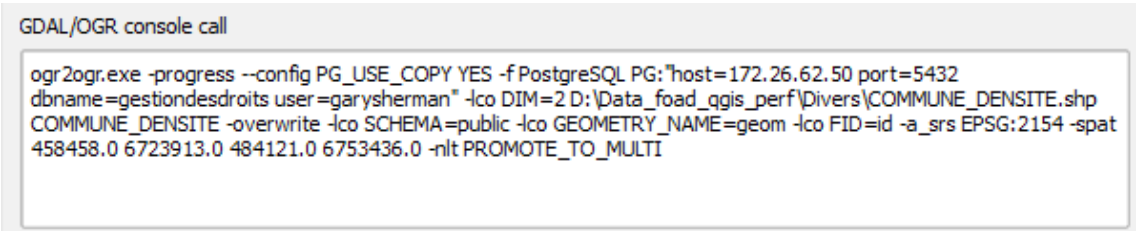


Par exemple, **`exporter vers PostgreSQL(connections disponibles)`** permet d'exporter dans PostGIS avec une connexion ouverte.

(nb : dans certains versions de QGIS le terme utilisé est *Importer* au lieu de *Exporter*)

Il s'agit en fait d'une aide à la rédaction d'une commande **ogr2ogr** qui apparaît en bas de la boîte de dialogue.

Exemple :

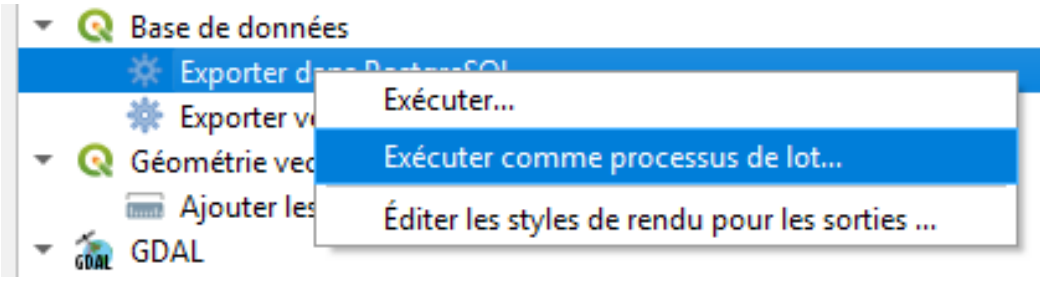


De nombreux paramètres sont réglables.

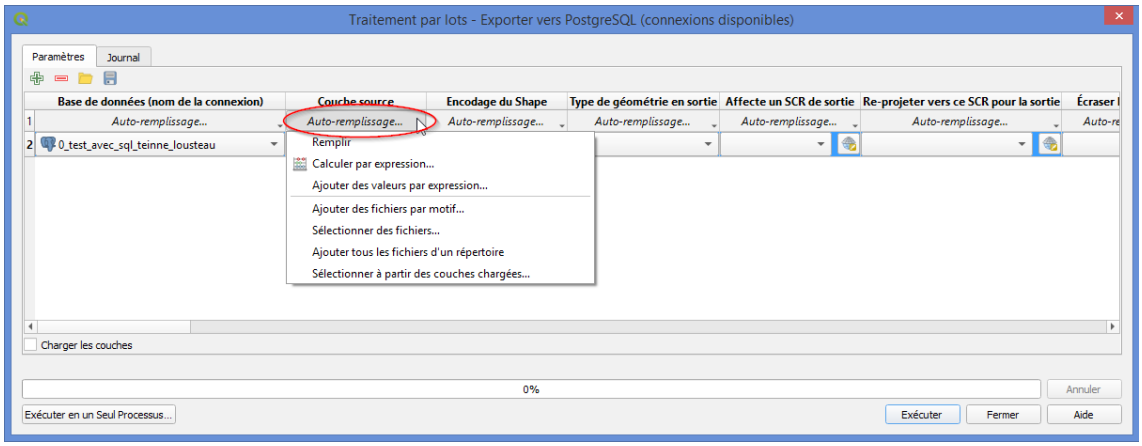
Ogr2ogr a été choisi par Faunalia pour réaliser cet algorithme. On trouvera *ici*[42] une justification en termes de performances par rapport à *shp2pgsql*.

À noter que cette méthode permet d'importer en lot. En effet un algorithme peut être lancé par **`clic droit → exécuter comme processus de lot`**.

42 - https://faunaliagis.wordpress.com/2014/11/24/a-new-qgis-tool-based-on-ogr2ogr-to-import-vectors-in-postgis-the-fast-way/



Le choix de plusieurs couches avec le bouton ... permet d'alimenter automatiquement autant de lignes que de couches à importer et de préciser ensuite les paramètres pour chaque couche :



L'import via ogr2ogr par cette méthode est considéré comme beaucoup plus performant en termes de rapidité que l'import par DBManager.