E. Exercice 01 - gestion des droits

1. 01 - gestion des droits

Après le cours et le pas à pas, voici un exercice à effectuer seul sur la gestion des droits : se positionner sous PgAdmin sur le schéma *production*.

Question

Mettre en pratique les acquis pour faire en sorte que *garyXX* :

- puisse créer de nouvelles tables.
- dispose des droits sur les tables existantes (ajouter, modifier et supprimer des enregistrements) du schéma *production*
- Dispose des droits d'écriture sur les objets créé par *stageXX* dans le schéma *production*

Pour vérifier cette dernière possibilité , charger dans QGIS la table *commune_densite.shp* (fournie dans les données du stage) avec le rôle *stageXX*

et utiliser le bouton 

de DBManager pour charger cette couche dans le schéma production (on appellera la table *commune_densite* et on n'utilisera pas d'option de création que nous verrons plus loin dans le cours).

Changer de connexion pour vous connectez avec le rôle *garyXX* et vérifier les privilèges.

Faire également en sorte que *michaelXX* puisse accéder en lecture au schéma *production*.

Consigne : Envoyer un message aux tuteurs pour indiquer que vous avez réalisé l'exercice en rappelant votre numéro de stagiaire.

Si vous avez mis des mots de passe autres que le nom de login pour *garyXX* et *michaelXX*, merci de les communiquer aux tuteurs.

2. Validation de l'exercice n°01 - gestion des droits

Comme c'est le premier exercice tutoré, certains utilisent la liste de discussion, plutôt que la BAL pour les tuteurs. Il faut leur faire la remarque. La liste de discussion est à destination de l'ensemble des apprenants. La BAL est le moyen d'envoyer les exercices tutorés et de dialoguer avec les tuteurs.

Pour la correction, il s'agit de vérifier que les stagiaires ont bien réalisé l'exercice mais également le 'pas à pas' qui précède.

Chaque stagiaire est identifié par un numéro symbolisé par *XX* dans la suite.

vérifier que :

- la base *droitXX* existe. Attention au respect de la casse, exemple Droit00 n'est pas correct.
- les rôles de groupe *lecteursXX* et *ecrivainsXX* existent.
- La base *droitXX* contient les schémas *production* et *consultation*. Le propriétaire des schémas (voir dans l'onglet propriété en se positionnant sur le schéma dans PgAdmin) doit être *stageXX*.

  Les ACL pour *production* doivent être {stageXX=UC/stageXX, ecrivainsXX=UC/stageXX} ce qui signifie que *stageXX* a accordé les droits Usage et Create à *stageXX* et au rôle de groupe *ecrivainsXX*. On peut faire remarquer qu'on pourrait également accorder le droit USAGE au rôle de

groupe *lecteurXX* sur le schéma *production*. Les droits par défaut sur les tables doivent être au minimum {ecrivainsXX=arwd/stageXX}, ce qui signifie que stageXX a accordé les droits pour ses futurs objets a=INSERT, r=SELECT, w=UPDATE d=DELETE (les autres possibilités sont D= TRUNCATE, suppression des lignes d'une table ou d'un ensemble de tables, x= REFERENCES pour la spécification de contraintes de clefs étrangères, t=TRIGGER création de trigger). Ces précisions peuvent, le cas échéant, être données dans le message accompagnant la correction de l'exercice.

Les ACL pour *consultation* doivent être {stageXX=UC/stageXX, lecteurXX=U/stageXX} et les droits par défaut {lecteurXX=r/stageXX}.

- Le schéma *production* contient les tables *route_xy* et *zonageppri_lafleche*, et *commune_densite*, le schéma *consultation* contient la table *commune*. les ACL sur *route_xy* et *zonageppri_lafleche* doivent contenir ecrivainsXX=arwd/stageXX.
- vérifier que la table *commune_densite* n'a pas été importé dans le schema *public*
- Les rôles de connexion *garyXX* (lecteur et écrivain) et *michaelXX* (lecteur) existent et appartiennent aux bons rôles de groupe. Dans les propriétés *garyXX* doit être membres de *ecrivainsXX*, *lecteurXX* et *michaelXX* doit être membre de *lecteurXX*.
- La vérification sous QGIS (se connecter avec le rôle *garyXX* et vérifier que les tables *route_xy* et *zonageppri_lafleche* ont les privilèges 'select, insert, update, delete'. Charger *route_xy* et vérifier que l'on peut passer en mode d'édition sous QGIS) n'est pas obligatoire si tout ce qui précède est respecté.

Vérifier que le stagiaire n'a pas réalisé plusieurs fois de suite une restauration. La table *route_xy* doit contenir 3523 enregistrements. Si ce n'est pas le cas il faut supprimer en cascade les tables dans les schémas *production* et *consultation* et recommencer la restauration.

On peut interroger les tables du catalogue système pour avoir le tableau général des relatons d'appartenance entre rôles de connexion et rôles de groupe.

```
select gr_id.rolname as groupe_name, mb_id.rolname as membre,
grt_id.rolname as grantor
from pg_auth_members as auth , pg_authid as gr_id ,pg_authid as
mb_id,pg_authid as grt_id
where auth.roleid=gr_id.oid
and auth.member=mb_id.oid
and auth.grantor=grt_id.oid
```

Complément : suppression de données

Pour supprimer des données dans une table, même si on dispose du droit delete, il faut que la table dispose d'une clef primaire. Il est possible d'ajouter une clef primaire (dans l'exemple pour la table *commune*) sous pgAdmin avec

```
ALTER TABLE consultation.commune
ADD CONSTRAINT pk_id PRIMARY KEY (id);
```

Ce que l'on peut également faire par clic droit -> ajouter un objet -> Clef primaire.

A noter qu'il ne faut pas qu'il y ait des enregistrements dupliqués. Pour la suppresion des doublons dans une table on peut renvoyer vers *ce site*[10].

10 - http://sqlpro.developpez.com/cours/doublons/

Complément : erreur : la base template_SIG est accédé par d'autres utilisateurs.

C'est une erreur que peuvent rencontrer les stagiaires. Il faut leur demander de se déconnecter le la base *template_sig* (clic droit -> se déconnecter de la base de données).

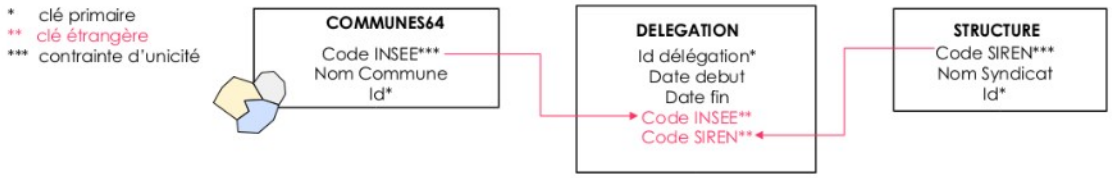
F. Exercice 03 - intégrité réfrentielle

1. Exercice

Cet exercice va vous permettre de vérifier l'intérêt de la gestion de l'intégrité référentielle dans un SGBD

Question

Toujours dans le schéma *travail (base stage XX)*, l'objectif est de créer les tables *Delegation* et *Structure* à partir du modèle de données ci-dessous :



Les fichiers *liste structure 64.dbf* et *liste delegation 64.dbf* sont fournis.

- Les importer sous PostgreSQL en passant par QGIS et DBbManager, puis compléter avec les contraintes du modèle de données (clef primaire et contraintes d'unicité).
- Pour les clefs étrangères, on utilisera l'action CASCADE en mise à jour et suppression.
- Afficher sous QGIS les tables *communes64* (géométrique) et les tables attributaires *structure* et *délégation*.
- Vérifier dans la table *délégation* l'existence de délégations avec le code siren 200026409 en utilisant un filtre d'expression.
- Dans la table *structure* supprimer le SIVOM de LAGOR (code siren = 200026409) et enregistrer.
- Vérifier à nouveau la table délégation.
- Conclure sur l'intérêt des clefs étrangères et du modèle relationnel et indiquer des exemples dans votre patrimoine de données qui pourraient bénéficier du modèle relationnel et la raison.

2. Validation de l'exercice n°03 - intégrité référentielle

Il s'agit de vérifier que les stagiaires ont bien réalisé l'exercice.

Bien vérifier que les stagiaires ont mis les contraintes d'unicité comme décrit dans le diagramme.

En réponse aux remarques des stagiaires sur l'utilité de l'intégrité référentielle, on

peut parfois faire référence à l'utilité des triggers que l'on voit plus loin dans la formation.

Pour les contraintes d'unicité on peut proposer d'adopter une règle de nommage : ex : uni_code_insee (éventuellement ajouter le nom de la table uni_code_insee_communes64).

vérifier que la table *commune64* compte 547 enregistrements.

Pour vérifier rapidement le reste de l'exercice :

commune64, la définition de la table doit comporter deux contraintes :

```
CONSTRAINT communes_64_pkey PRIMARY KEY (id),
CONSTRAINT unicite_insee_communes64 UNIQUE ("INSEE_Commune")
```

structure, la définition de la table doit comporter deux contraintes :

```
CONSTRAINT structure_pkey PRIMARY KEY (id),
CONSTRAINT uni_siren UNIQUE (n_siren)
```

delegation, la définition de la table doit comporter trois contraintes :

```
CONSTRAINT delegation_pkey PRIMARY KEY (id),
CONSTRAINT fkinsee FOREIGN KEY (code_insee)REFERENCES travail.communes_64 ("INSEE_Commune") MATCH SIMPLE ON UPDATE CASCADE ON DELETE CASCADE,
CONSTRAINT fksiren FOREIGN KEY (code_siren)REFERENCES travail.structure (n_siren) MATCH SIMPLE ON UPDATE CASCADE ON DELETE CASCADE
```

G. Exercice 07 - SQL UPDATE

1. 07 - 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 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....**

2. Validation de l'exercice 07 : CREATE TABLE, UPDATE, ALTER TABLE

Suivi des modifications

Octobre 2017 : On demande désormais de typer les géométries ce qui est une bonne pratique.

du coup le UPDATE est désormais utilisé pour changer les noms de communes

et on utilise ALTER pour modifier le type du champ de géométrie de POINT à POLYGON et USING pour préciser en même temps la conversion avec un buffer.

Cet exercice pose en principe peu de problèmes. Il faut penser à mettre les champs entre "", puisqu'on utilise des majuscules.

H. Exercice 09 - CASE ... WHEN

1. 09 - 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

2. Validation de l'exercice 09 : CASE ... WHEN

Dans la solution on peut utiliser sum() à la place de count() puisque l'on renvoi du numérique.

L'expérience montre que certains stagiaires ont du mal à démarrer l'exercice. Dans ce cas on peut leur transmettre un squelette de la solution en leur demandant de compléter.

Par exemple :

```
SELECT "Statut" AS "Statut commune",
count(CASE WHEN "Code_Arrondissement" = '1' THEN 1 ELSE NULL END) AS "Bayonne",
count (... un peu la même chose...,
count(... idem)
FROM travail.commune64
GROUP BY "Statut" ORDER BY "Statut"
```

On peut en mettre un peu moins pour une première aide comme :

```
SELECT "Statut" AS "Statut commune",
count (CASE WHEN "Code_Arrondissement" ='xxx' THEN xx ELSE xx END) AS
"Bayonne",
......
```

I. Exercice 10 - sous-requêtes

1. 10 - 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	URI
3	MOUGUERRE
4	LAHONCE
5	SAINT-PIERRE-D'IRUBE
6	VILLEFRANQUE

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

2. Validation de l'exercice 10 : sous-requêtes

Cet exercice peut poser des difficultés aux stagiaires qui ne maîtrisent pas totalement le SQL. Il convient alors de les accompagner et leur proposant des squelettes de solutions.

La solution avec WITH nécessite d'utiliser une jointure ou une sous-requête. La solution proposée utilise une jointure, mais une sous-requête est acceptable.

La solution propose = pour la requête 1, car la requête `SELECT n_siren FROM travail.structure WHERE structure.nom_struct = 'COMMUNAUTE DE COMMUNES NIVE-ADOUR'` renvoi un seul enregistrement, mais si on utilise IN on s'aperçoit avec EXPLAIN ANALYZE que la requête est beaucoup plus performante (Information à conserver si un apprenant pose la question). Voici la requête complète :

```
SELECT
"Nom_Commune"
FROM
travail.communes64
WHERE "INSEE_Commune" IN
(SELECT code_insee FROM travail.delegation WHERE code_siren IN
(SELECT n_siren FROM travail.structure WHERE structure.nom_struct =
'COMMUNAUTE DE COMMUNES NIVE-ADOUR'))
```

J. Exercice 12 non tutoré - vue modifiable

1. 12 - 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).

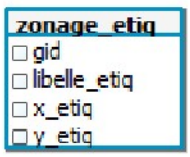
Cette nouvelle table aura pour attributs :

- 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)
```

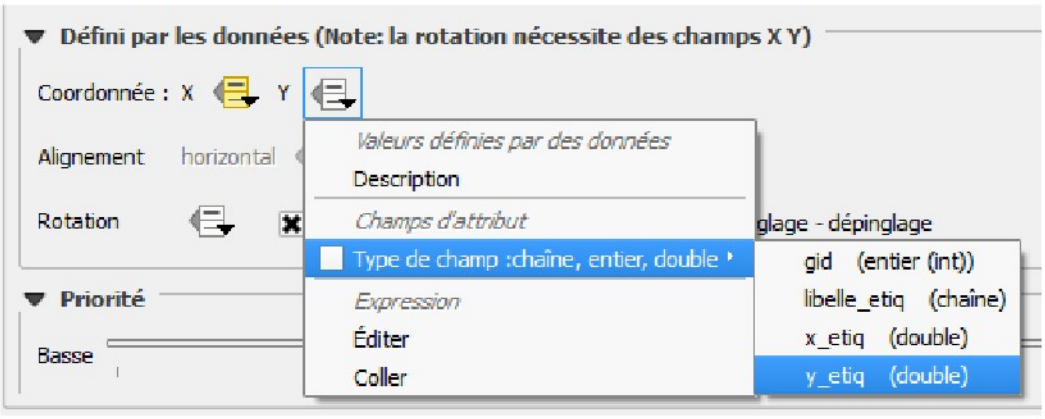
4) créer le trigger sur la vue vue_zonage.

5) faire un essai en modification d'étiquette en ajoutant la vue vue_zonage au canvas 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***

2. Validation de l'exercice 12 - vue modifiable

Cet exercice est difficile. Il faut aider les stagiaires en apportant des éléments de réponse en fonction de leurs difficultés.

exemple de fonction à compléter (on peut en donner un peu moins au début) :

```
CREATE FUNCTION travail.maj_zonage_etiq()
RETURNS trigger AS
$BODY$
BEGIN
IF OLD.gid IN (SELECT gid FROM travail.zonage_etiq) THEN
UPDATE travail.zonage_etiq
SET (libelle_etiq, x_etiq,NEW.x_etiq, NEW.y_etiq) = ???
WHERE gid=OLD.gid;
ELSE
INSERT INTO travail.zonage_etiq
VALUES(???);
END IF;
RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql;
```

Un des pièges principaux dans la création de la fonction est de comprendre que l'on doit modifier directement la table *travail.zonage_etiq* et non pas la vue.

En effet, un trigger sur une vue doit modifier les tables sources.

Il faut donc préciser aux stagiaires qu'il fait modifier la bonne table par un `UPDATE travail.zonage` (modification) et un `INSERT INTO travail.zonage_etiq VALUES` (création).

Autre piège utiliser `= NULL`, il faut utiliser `IS NULL`.