ENSG Géomatique ÉCOLE NATIONALE DES SCIENCES GÉOGRAPHIQUES	support de cours	Département de Cartographie et d'Analyse de l'Information Géographique

PostGIS

Cours et Support de TP

Auteurs : Emmanuel Fritsch, Jacques Gautier
Version 2021.1

Sommaire

1 Présentation

1.1 Présentation du TP

PostGreSQL est un SGBD (Système de Gestion de Base de Données) libre de très bonne qualité. De nombreuses applications professionnelles sont construites sur PostGreSQL. PostGIS est une surcouche de PostGresSQL qui permet de gérer la géométrie.

L'objectif pédagogique de ce TP est :
- de réviser les fonctionnalités élémentaires des SGBD
- de vous faire découvrir PostGIS
- de vous amener à l'autonomie dans la découverte d'un SGBD.

Les notions et techniques étudiées sont :
- la connexion à PostgreSQL
- le format WKT
- l'intégration dans une table de fichiers shape
- les requêtes sémantiques et spatiales
- les jointures spatiales
- Le langage pl/pgSQL
- l'utilisation des index

Dans le cadre de ce TP, nous allons travailler avec des données sismologiques d'épicentres de tremblements de terre, représentés par des points. Nous croiserons ces données avec les données surfaciques des surfaces terrestres du globe.

1.2 Rendu attendu

Pour vous aider à maintenir votre attention tout au long du TP, et pour aider l'enseignant à vous évaluer, certaines questions demandent de votre part que vous écriviez la réponse dans le formulaire à la fin de l'énoncé. Les questions réclamant des réponses sont signalées ainsi :

Question 0 – Ceci est une question dont la réponse doit être notée dans le cadre n° 0. Ce cadre n° 0 est là pour vous permettre de noter vos nom et prénom.

Les autres questions et les manipulations du logiciel sont indiquées ainsi :

⇨ Ceci est une instruction qui ne demande pas de rendu. Mais si vous la négligez, vous n'arriverez peut-être pas à répondre aux questions suivantes.

Même lorsque l'enseignant ne demande pas que le travail lui soit rendu, il est néanmoins conseillé, pour une bonne mémorisation du cours, d'écrire les résultats obtenus, et en particulier de recopier les requêtes que vous codez.

1.3 Méthode

Ce TP alterne les phases où vous êtes guidés pas à pas dans la manipulation de l'interface avec des phases qui vous mènent à l'autonomie. Dans cet objectif, vous devez penser à aller chercher l'information dans les documentations techniques disponibles sur internet.

Le moteur de recherche Google convient très bien à cet objectif. Si vous cherchez par exemple de l'information sur l'import de fichiers Excel dans une base PostgreSQL, vous pouvez, au choix :

- vous rendre sur la documentation en ligne de PostgreSQL et rechercher dans la doc comment importer des données depuis un fichier Excel
- taper « import de fichiers Excel dans PostgreSQL » dans une barre de recherche Google, ce qui vous enverra vers la page correspondante dans la documentation en ligne (au mieux), ou une discussion dans un forum sur ce sujet.

1.4 Prérequis

Les prérequis de ce TP sont une connaissance préalable de PHP et d'au moins un SGBD. Il faut savoir utiliser un éditeur de texte.

La connaissance de QGIS est souhaitée. A défaut, celle d'autres SIG est nécessaire.

1.5 Données

Ce TP est accompagné des données qui permettent de le mettre en œuvre. Ces données sont dérivées de données disponibles sur internet :

- le fichier CMT des tremblements de terre (http://www.globalcmt.org/CMTfiles.html)
- le fichier des limites administratives sur l'étendue du globe : (http://www.gadm.org/world)

Ces données ont été retouchées pour être adaptées au TP.

Le TP ne fonctionnera pas correctement si vous utilisez les données référencées ci-dessus. Il faut utiliser les données dérivées, telles qu'elles sont livrées avec le TP.

1.6 Matériel et logiciels

Pour mettre en œuvre ce TP, il faut avoir accès à un poste informatique équipé :

- d'un serveur postGreSQL
- de pgAdmin 4, le GUI standard de postgreSQL. Pour les besoins du TP, l'utilisation de pgAdmin 3, GUI toujours fonctionnel, pourra être demandée.
- d'un navigateur, de préférence Firefox
- d'une connexion internet, pour avoir accès à la documentation en ligne
- de QGIS, pour visualiser les données
- pour la fin du TP, il faut disposer d'un serveur web local équipé de PHP, avec les extensions pgsql.

Un bon éditeur de texte, par exemple notepad++, est par ailleurs utile.

Pour vérifier la disponibilité de postGreSQL :

⇨ Ouvrez pgAdmin 4/III

Dans le panneau gauche de pgAdmin, vous trouvez un explorateur qui vous permet de plier et déplier les différentes composantes de votre BD. Un clic-droit sur un élément vous permet d'accéder aux opérations autorisées sur cet élément.

Une instance du serveur local de PostGreSQL doit apparaître sous la forme :

```
PostgreSQL 9.4 (localhost:5432)
```

Il est possible que la version de PostGreSQL installée sur votre machine soit postérieure à la version 9. Ex :

```
PostgreSQL 12 (localhost:5432)
```

Cela ne pose pas de problème pour ce TP, les notions étudiées ne changeant pas selon les différentes versions.

- ⇨ Double-cliquez sur cette instance du serveur local,
- ⇨ Entrez le mot de passe : *postgres*

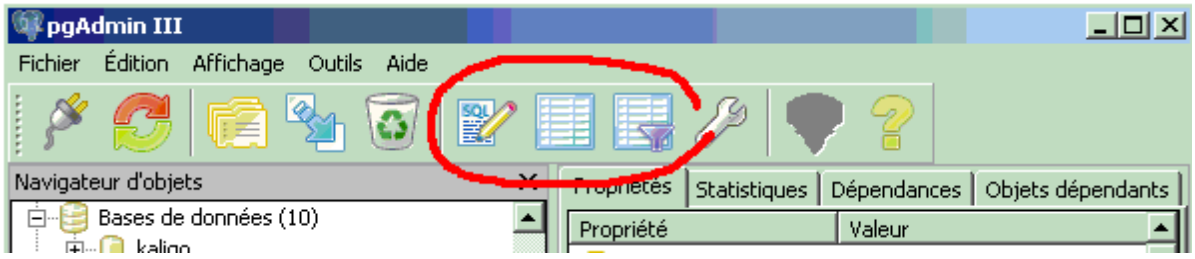
> Attention, le couple postgres/postgres comme login/motdepasse est une faille de sécurité majeure sur un serveur en ligne. Il n'est tolérable que parce que nous allons travailler sur un serveur local.
> A partir du moment où vous mettrez votre serveur sur un réseau, même pour un serveur de développement, même pour un réseau local, il faudra au minimum changer le mot de passe du rôle de connexion `postgres`,

- ⇨ Dépliez la racine « bases de données » en cliquant sur la petite croix. Vérifiez que la base **`postgres`** est présente. Dépliez cette base, dépliez « schémas », dépliez le schéma **`public`**.
- ⇨ pgAdmin vous montre alors qu'il y a une table **`spatial_ref_sys`**. Il peut y avoir aussi un table **`geometry_columns`** dans cette base (selon les versions **`geometry_columns`** est une **table** ou une **vue**)

1.7 PgAdmin

1.7.1 PgAdmin III

Dans l'interface de pgAdmin III, il y a trois boutons qu'on utilise très souvent, et que je vous présente maintenant :



Le premier de ces boutons est l'éditeur de requête SQL. Il permet d'écrire et de faire tourner du code SQL sur une base. Le bouton n'est actif que lorsqu'une **base** est sélectionnée.

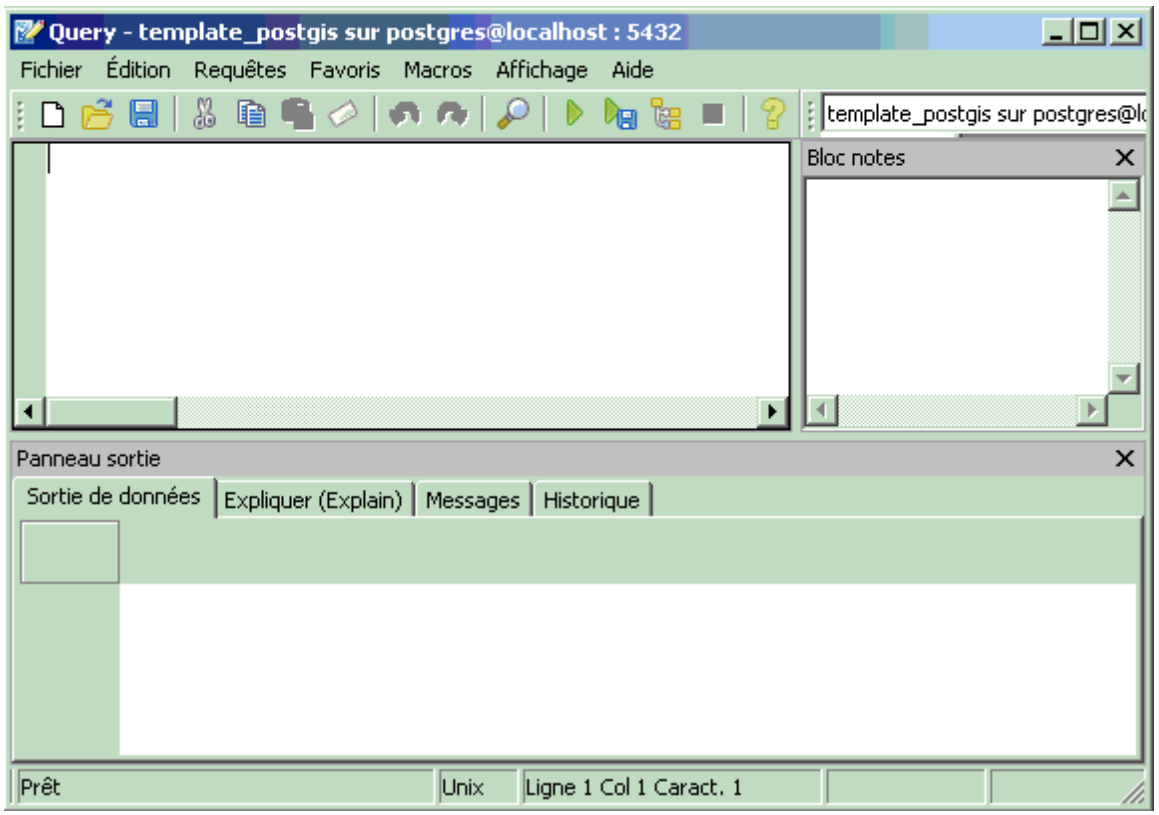
Les deuxième et troisième boutons ne sont actifs que si une **table** est sélectionnée. Le deuxième bouton permet d'afficher toutes les données de la table sous la forme d'un tableau. Le troisième bouton permet de saisir un filtre sur la table pour n'en afficher qu'un nombre réduit de lignes (une restriction).

L'éditeur de requête

Regardons plus attentivement l'éditeur de requête.

- ⇨ Cliquez sur le premier des trois boutons décrits ci-dessus (le bouton « SQL » de pgAdmin).

L'éditeur ressemble à ceci :



Le cadre en haut à gauche permet de taper la requête SQL. Le bouton constitué d'un triangle vert exécute la requête. Le cadre en bas permet de lire la réponse à la requête, et les erreurs éventuelles.

- ⇨ Tapez le code SQL suivant : `SELECT 2+2`, puis cliquez sur le triangle vert.
- ⇨ Tapez le code SQL suivant : `SELECT 2+deux`, puis cliquez sur le triangle vert.
- ⇨ Tapez le code SQL suivant : `SELECT 2+'deux'`, puis cliquez sur le triangle vert.

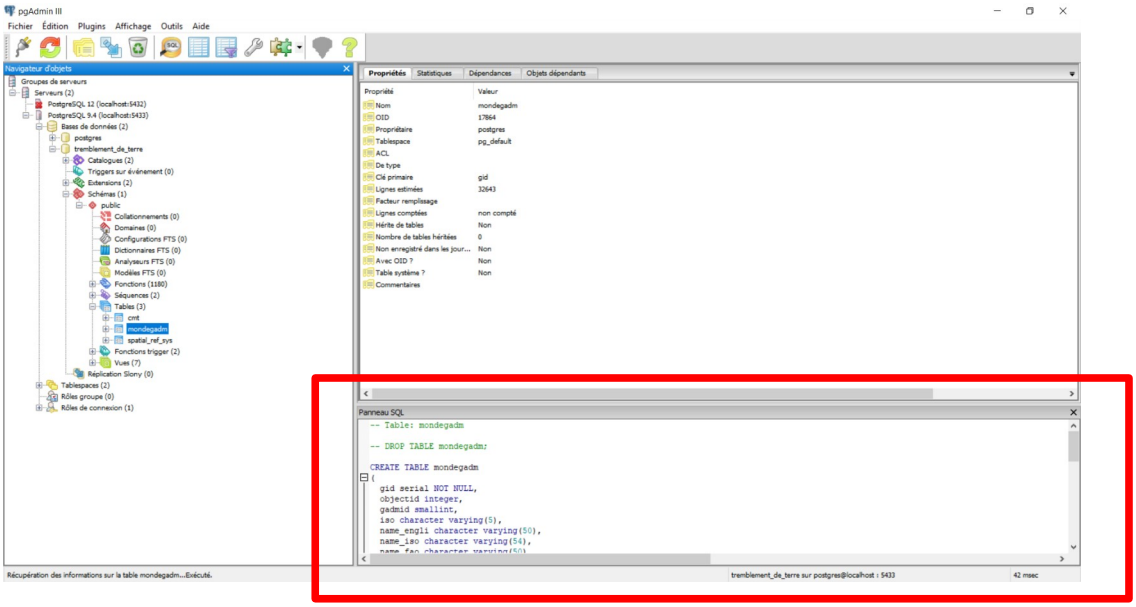
1.7.2 PgAdmin 4

L'interface est légèrement différente dans PgAdmin4.
Vous pouvez d'ores et déjà remarquer que l'application PgAdmin III ne se lance pas en tant qu'application séparée, mais au travers d'un navigateur Web.

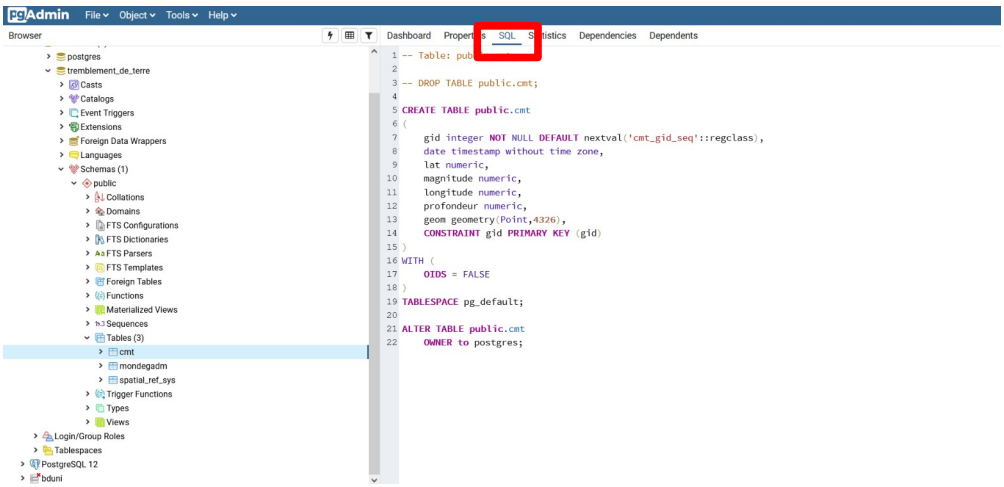
PgAdmin4 intègre différentes nouvelles fonctionnalités par rapport à PgAdmin III, qui peuvent être intéressante pour la gestion de base de données : par exemple, un onglet 'Dashboard' permet d'avoir un suivi en temps réel de la consultation d'une base de données hébergée sur votre serveur PGSQL.
Cependant, concernant les notions vues pendant ce cours, les deux application PgAdmin III et 4, sont équivalentes, et vous interagissez de la même manière avec votre base de données dans les 2 applications. Les seules différences sont au niveau de l'interface et sur l'accès à certains outils.

Panneau SQL

Contrairement à PgAdmin III, le panneau SQL indiquant la requête SQL permettant de créer un élément sur lequel vous avez cliqué dans l'interface n'est pas affiché par défaut dans PgAdmin 4.



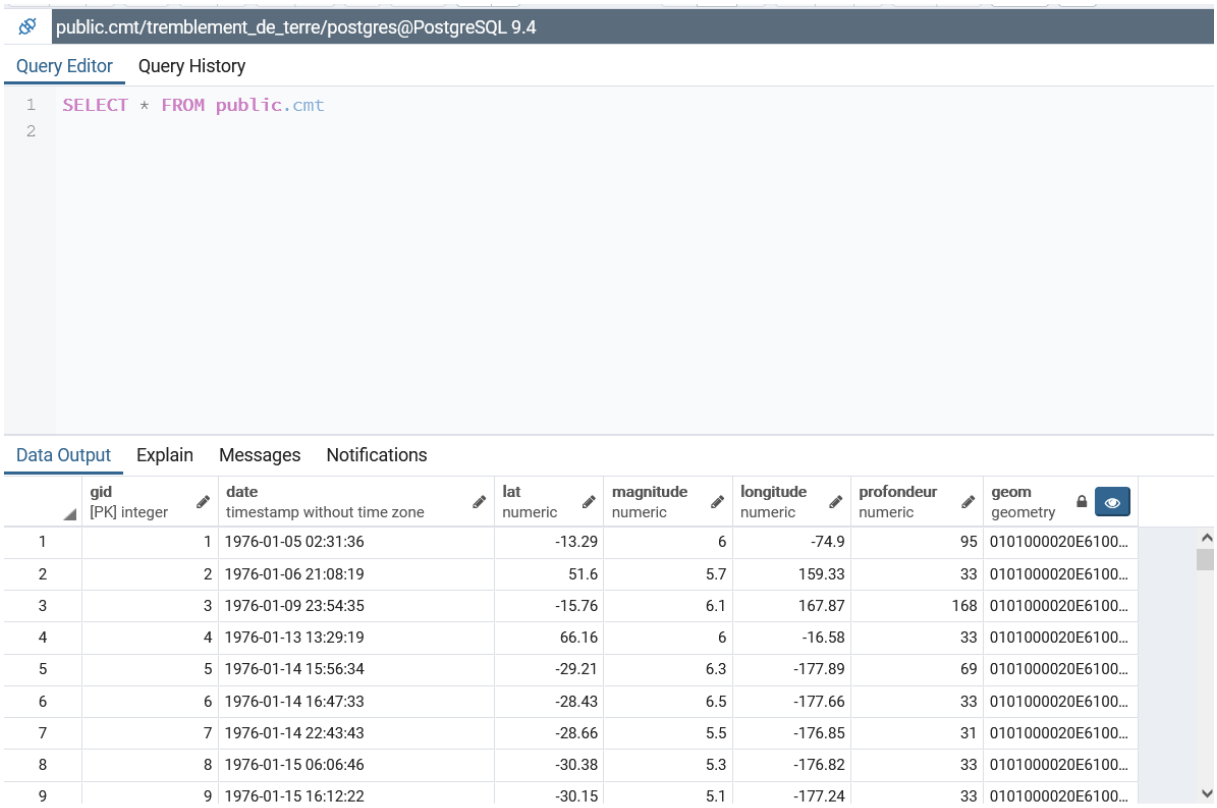
Pour afficher l'équivalent dans PgAdmin 4, vous devez sélectionner l'onglet 'SQL' dans la fenêtre de droite.



Outil de requête SQL et affichage du contenu d'une table

Les boutons permettant d'accéder à l'outil de requête SQL, et d'afficher le contenu de la table sont légèrement différents dans PgAdmin 4.

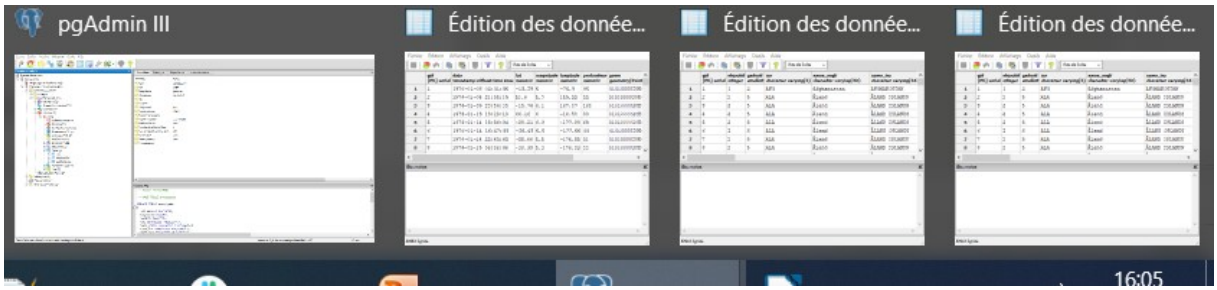
Dans PgAdmin III, l'outil de requête SQL et l'outil de visualisation du contenu de la table s'ouvrent dans deux types de fenêtres différents.
Dans PgAdmin 4, ces outils s'ouvrent dans une même interface, gardant la même structure dans les deux cas, avec en haut la requête SQL effectuée (ou son équivalent lorsque vous demandez à ouvrir une table), et en bas le résultat de la requête SQL effectuée (et donc le contenu de la table lorsque vous demandez à ouvrir une table)



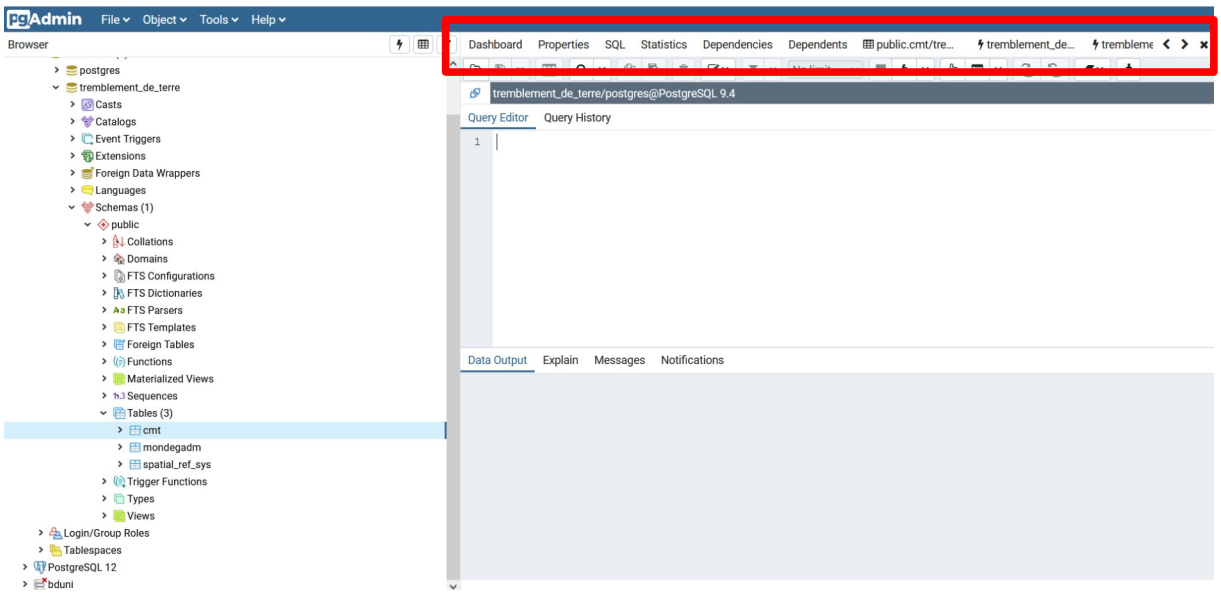
	gid [PK] integer	date timestamp without time zone	lat numeric	magnitude numeric	longitude numeric	profondeur numeric	geom geometry
1	1	1976-01-05 02:31:36	-13.29	6	-74.9	95	0101000020E6100...
2	2	1976-01-06 21:08:19	51.6	5.7	159.33	33	0101000020E6100...
3	3	1976-01-09 23:54:35	-15.76	6.1	167.87	168	0101000020E6100...
4	4	1976-01-13 13:29:19	66.16	6	-16.58	33	0101000020E6100...
5	5	1976-01-14 15:56:34	-29.21	6.3	-177.89	69	0101000020E6100...
6	6	1976-01-14 16:47:33	-28.43	6.5	-177.66	33	0101000020E6100...
7	7	1976-01-14 22:43:43	-28.66	5.5	-176.85	31	0101000020E6100...
8	8	1976-01-15 06:06:46	-30.38	5.3	-176.82	33	0101000020E6100...
9	9	1976-01-15 16:12:22	-30.15	5.1	-177.24	33	0101000020E6100...

Disposition en fenêtres/onglets

Dans PgAdmin III, chaque demande pour afficher le contenu d'une table, ou ouvrir un outil de requête créé une nouvelle fenêtre Windows.



Dans PgAdmin 4, chaque demande pour afficher le contenu d'une table, ou ouvrir un outil de requête créé un nouvel onglet dans la partie droite de la page PgAdmin dans votre navigateur.



1.8 Création d'une base PostGIS

Nous allons maintenant créer une base, et la doter des fonctionnalités de postGIS.

- ⇨ Dans la colonne de gauche de PgAdmin, clic-droit sur « base de données », puis clic sur « ajouter une base de données ».
- ⇨ Entrez **`tremblement_de_terre`** comme nom de la base.
- ⇨ Vérifiez que l'encodage est en UTF8.

- ⇨ Cliquez sur OK. Attendez un peu.

Votre base doit maintenant être créée : la base apparaît dans pgAdmin.

- ⇨ Dans l'éditeur de requête, entrez : `CREATE EXTENSION postgis`.
- ⇨ Lancez cette requête.

Si cette requête renvoie une erreur, c'est sans doute que postgis n'est pas installé sur votre serveur postGreSQL. Le plus simple est de désinstaller postGreSQL et de le réinstaller avec postGIS. Pour cela, utilisez l'installeur de EnterpriseDB (https://www.enterprisedb.com/downloads/postgres-postgresql-downloads). Dans le cadre de ce TP, nous recommandons une des versions 9, par exemple la version 9.4.

Effectuez cette manœuvre dans PgAdmin. Une fois terminée cette manœuvre, votre base a dû maintenant recevoir les fonctionnalités de postGIS, mais cela n'aparaît pas dans pgAdmin dans les extensions de votre base. Pourquoi ?
L'éditeur SQL de PgAdmin est un client. La fenêtre principale de PgAdmin est un autre client. Ils sont indépendants l'un de l'autre, et en dehors des requêtes qu'ils échangent avec le serveur, ils sont indépendants de celui-ci.
En ce moment, l'éditeur SQL a créé l'extension postGIS sur le serveur, mais l'interface principale de pgAdmin ne le sait pas. Ce que vous voyez dans votre interface n'est pas à jour. Il va falloir resynchroniser cette interface avec le serveur.

- ⇨ Dépliez la base **`tremblement_de_terre`** : vérifiez qu'aucune nouvelle table n'est apparue.
- ⇨ Clic-droit, refresh : une nouvelle table apparaît.

> Lorsque vous constatez une incohérence dans votre base de données, tentez le rafraîchissement de l'interface : cela règle votre problème dans la majorité des cas.

- ⇨ Vérifiez que la table **`spatial_ref_sys`** a bien été créée.
- ⇨ Dépliez cette table **`spatial_ref_sys`**, dépliez ses colonnes. Que devrait contenir cette table ?

- ⇨ Au passage, vérifiez qu'il y a aussi quelques centaines de fonctions. Le nombre exact varie selon la version de PostGIS.

Toutes les fonctions PostGIS sont préfixées par les trois caractères ST_.

- ⇨ Dépliez les fonctions, et vérifiez que les premières que vous voyez sont bien préfixées par ST_

- ⇨ Affichez le contenu de la table **`spatial_ref_sys`**, puis de la vue **`geometry_columns`**. Essayez d'ajouter manuellement un élément dans la table, puis dans la vue. Que constatez-vous ? Comment expliquez vous la différence de comportement ?

- ⇨ Tapez le code SQL suivant : **`SELECT * FROM geometry_columns`**, et vérifiez que cette vue est vide.
- ⇨ Tapez le code SQL suivant : **`SELECT * FROM spatial_ref_sys`**, vérifiez que cette table est peuplée.
- ⇨ Tapez le code SQL suivant : **`SELECT * FROM spatial_ref_sys WHERE srid=4326`**. On trouve la référence à la projection plate carrée que nous utiliserons dans le cours de ce TP.
- ⇨ Tapez le code SQL suivant :

  ```
  SELECT * FROM spatial_ref_sys WHERE srtext LIKE '%Douala%'.
  ```

  Cette requête permet de trouver toutes les projections dont la description contient le mot « Douala ». La projection actuellement utilisée sur le Cameroun y est-elle référencée ?

Attention, La table **spatial_ref_sys** est protégée par deux contraintes :

- La contrainte **spatial_ref_sys_pkey** empêche de mettre deux fois la même valeur dans la colonne **srid**.
- La contrainte **spatial_ref_sys_srid_check** oblige à ce que la valeur de **srid** soit comprise entre 1 et 222.

⇨ Non, pas entre 1 et 222. Dans pgAdmin, explorez les contraintes de cette table pour trouver la vraie plage de valeur autorisée pour **srid**.

Il existe une autre méthode pour activer postGIS sur une base, c'est de créer la base en copiant directement son contenu depuis une base qui possède déjà postGIS. Ainsi, si vous créez une base **tremblement_de_terre_2** en prenant **tremblement_de_terre** comme modèle (il suffit de sélectionner **tremblement_de_terre** dans le champ modèle de l'interface), la nouvelle base récupérera toutes les fonctionnalités de la première. C'est une méthode facile pour faire un clone d'une base.
Si vous faites le test, n'oubliez pas de détruire **tremblement_de_terre_2** ensuite.

Si la destruction de la base **tremblement_de_terre_2** échoue, cela peut être dû au fait que la base est restée ouverte. Repliez-la, puis avec un clic-droit sur « base de données », cliquez sur « rafraîchir ». Relancez alors la destruction de la base.

2 Structuration de la base et insertion des données

2.1 Découverte des données

Nous allons maintenant importer des données de tremblements de terre, géolocalisés par l'épicentre du séisme.

⇨ Avec un éditeur de texte (par exemple NotePad++), ouvrez le fichier CMT au format **.ndk** pour la période 1976-2005

⇨ Regardez rapidement les données (la première ligne), et la description du format **.ndk**.

https://www.ldeo.columbia.edu/~gcmt/projects/CMT/catalog/allorder.ndk_explained

2.2 Construction d'une table

L'intégration de ces données dans la base de données est une opération un peu compliquée, et que nous renvoyons à plus tard. Nous allons partir d'un fichier intermédiaire, où les données sont déjà préparées et mieux structurées : un format csv.

⇨ Avec un moteur de recherche, trouvez comment intégrer des données au format csv dans postgreSQL.

Avant de faire les insertions dans une table des tremblements de terre, nous devons créer cette table. Et avant de créer la table, nous devons construire le modèle physique des données.

Donc : dans un premier temps on analyse **tdt.csv** pour lister les colonnes que nous allons mettre dans notre table. Ensuite, nous allons utiliser pgAdmin pour construire la table des tremblements de terre. Pour finir, nous allons peupler cette table à partir des données qui se trouvent dans **tdt.csv** .

2.2.1 Modèle physique des données

⇨ Pour définir le format de la table dans laquelle seront rangés les tremblements de terre, analysez les différentes colonnes du fichier **`tdt.csv`** .

Truc et Astuce : parmi les petits détails que vous ne pouvez pas deviner tout seul, apprenez que pour la colonne **`dateheure`**, nous allons utiliser un type SQL qui s'appelle **`timestamp without time zone`**.

Notez au passage que PostgreSQL ne fait pas dans la concision. SQL en général préfère des noms à rallonge mais faciles à comprendre, plutôt que courts mais ésotériques.

Question 1- Ecrivez la liste des colonnes avec leur type.

Attention, n'utilisez jamais de majuscule dans postGreSQL. Les noms comportant des majuscules sont autorisés, mais doivent être échappés ensuite avec des "quote double". C'est une source d'erreurs de syntaxe très fastidieuses. Mieux vaut donc écrire tous les noms en minuscules. Là où en PHP, javascript ou C on utiliserait une variable **`dateHeure`**, en SQL on préférera donc **`dateheure`** ou **`date_heure`**.

2.2.2 Construction de la table

Pour construire rapidement la base en apprenant à vous servir de pgAdmin, je vous conseille de suivre pas à pas les instructions qui suivent :

⇨ clic-droit sur « public »
⇨ Ouvrez "ajouter un objet" / "ajouter une table".
⇨ Onglet « Propriété » : Saisissez le nom de la table : **`tdt`**
⇨ Onglet «Colonnes » : Saisissez les noms de colonnes (avec leur type) :
  - **`id`** (de type **`SERIAL`**),
  - **`dateheure`** (de type **`timestamp without time zone`**),
  - **`latitude`**, **`longitude`** et les autres (de type **`real`**).
⇨ Onglet « Contraintes » : ajoutez une clef primaire, portant sur la colonne **`id`**.
⇨ Cliquez sur OK : la table est créée.

Si vous ne saisissez pas de clef primaire, un message vous avertit que vous n'avez pas de clef primaire sur votre table, ce qui limitera les opérations que vous pouvez faire avec. En particulier avec pgAdmin, vous ne pourrez pas modifier les données en table. C'est pour cette raison que nous avons créé une colonne d'identifiant.

Dans une base avec plusieurs tables et des liens explicites entre tables, l'identifiant sert de support aux liens entre tables, et garantit l'intégrité de l'information portée par le lien.

Plus précisément, si la colonne **enseignant** de la table cours contient « Emmanuel Fritsch » et qu'il y a plusieurs Emmanuel Fritsch dans la table **enseignant**, on introduit une ambiguïté dans la base. En revanche, si la colonne **enseignant** de la table des cours contient l'identifiant de l'enseignant, cette colonne pointe vers l'enseignant de manière fiable et sans ambiguïté.
Attention, ce que nous tenons pour un identifiant n'en est pas toujours un, ou n'est un identifiant que sous certaines hypothèses qui connaissent des exceptions. Ainsi, un numéro de téléphone mobile peut être considéré comme un identifiant, mais il y a parfois des gens qui partagent un même terminal mobile. Pour cette raison, il est plus robuste de créer une colonne qui n'existe que pour servir de support à la clef primaire. Elle se nomme généralement `id` ou `gid`.

Syntaxe SQL pour la création de tables, de colonnes, de clefs, etc.

Tous les objets que nous créons avec l'interface de PgAdmin peuvent être créés directement avec le code SQL, à condition que l'on en connaisse la syntaxe.

Pour cela, on peut explorer la documentation, mais pour ces opérations élémentaires de création et de modification des structures de données, il est plus simple de consulter le code des objets existants directement dans pgAdmin.

- ⇨ Sélectionnez la table **`tdt`** en cliquant dessus, puis activez l'onglet SQL (en haut à gauche) : dans le panneau SQL (à droite), vous voyez apparaître le code SQL de création de cette table. Vous pouvez sélectionner ce code, et le recopier pour créer une table semblable.
- ⇨ Naviguez dans les colonnes de la table **`tdt`** : sélectionnez une colonne, et regardez le code SQL correspondant : il s'agit d'une requête de modification de la table, qui y ajoute une colonne.

Le type SERIAL

Quel est donc ce type **`SERIAL`** que nous avons affecté à la colonne **`id`** ?

Le type **`SERIAL`** est un raccourci qui permet en une seule déclaration :

- de créer une colonne de type **`integer`**
- de lui ajouter, en valeur par défaut, une valeur incrémentale
- de créer le compteur spécifique qui va gérer cet incrément. Ce compteur s'appelle une « séquence » (du latin *sequor*, suivre).

- ⇨ Vérifiez que la séquence a été créée.
- ⇨ Vérifiez que la valeur par défaut de la colonne **`id`** est la valeur du compteur incrémentée.

Question 2 – Trouvez dans pgAdmin l'instruction SQL qui dit que la valeur par défaut de la colonne `id` est la valeur du compteur incrémentée. Recopiez-la.

2.3 Insertion dans la table

Nous allons maintenant intégrer le fichier **`tdt.csv`** au moyen de la requête COPY.

- ⇨ A partir de la documentation en ligne, rédigez la requête qui copie un fichier csv dans une table SQL.
- ⇨ Lancez cette requête.

L'éditeur SQL doit vous signaler que 24.000 lignes ont été créées.

Si vous avez fait tourner deux fois la requête, vous avez intégré deux fois les données. Elles se trouvent donc dupliquées. Dans ce cas :

- ⇨ Videz la table avec la requête : `DELETE FROM tdt`
- ⇨ Relancez la requête de copie des données (une seule fois).

2.4 Contrôle des données

Vous venez d'importer des données. Vous devez les contrôler. Ce contrôle concerne aussi bien des erreurs de bas niveau (les données sont-elles bien arrivées ? Y a-t-il des valeurs NULL ?) que des erreurs plus complexes (mes données reflètent-elles bien la réalité?). Ce contrôle doit traiter aussi bien des données sémantiques que de leurs géométries.

Ce contrôle peut se faire d'abord de manière visuelle, en ouvrant la table, lorsqu'elle n'est pas trop volumineuse, ou en consultant les *n* premiers éléments. Ainsi, par un clic-droit sur la table, puis « Afficher les données » / « visualiser le 100 premières lignes », vous avez un accès rapide à un échantillon de vos données.

De manière plus fouillée, on utilisera quelques commandes SQL simples pour vérifier la cohérence des attributs. Pour cela, nous allons devoir voir ou revoir la syntaxe de quelques commandes et clauses SQL simples.

3 Requêtes SELECT simples

3.1 La clause WHERE

La requête SELECT permet d'extraire des informations d'une table.

1– On n'a pas besoin d'extraire toutes les colonnes. On indique les noms des colonnes à extraire :

```
SELECT id, magnitude FROM table
```

2– Et surtout, on n'a pas besoin d'extraire toutes les lignes (tous les objets stockés dans la table). On donne les conditions que doivent remplir les objets à extraire :

```
SELECT id, magnitude FROM table WHERE magnitude>7.5
```

Ici, seuls les objets de magnitude supérieure à 7.5 sont retenus.

Les conditions que doivent respecter les objets à extraire sont spécifiées au moyen de la clause WHERE. Il est possible de combiner plusieurs conditions avec les opérateurs booléens, le plus souvent AND et OR. Par exemple :

```
SELECT id, magnitude, geom
FROM table
WHERE magnitude>5.5 AND profondur>300
```

⇨ Adaptez la requête précédente et faites-la tourner sur la base.

3.2 La fonction count()

⇨ Dans la documentation de PostGreSQL, trouvez la signification de la fonction count() ; utilisez-la pour déterminer le nombre d'objets dans la table `tdt`. Vérifiez que le nombre trouvé correspond à celui donné par pgAdmin.

⇨ Ecrivez la requête qui détermine le nombre de tremblements de terre qui répondent à la condition `magnitude>6.5 AND profondeur>500`.

Question 3 - Recopiez cette requête et son résultat.

3.3 Les Clauses ORDER BY et LIMIT

⇨ Dans la documentation de PostGreSQL, trouvez la signification de la clause `ORDER BY` et de la clause `LIMIT`.
⇨ Écrivez la requête qui trouve la magnitude du tremblement de terre de plus forte magnitude.
⇨ Écrivez la requête qui trouve la magnitude du tremblement de terre le plus profond.

Question 4 – Recopiez les deux requêtes, et leur résultat.

3.4 Contrôle des données - suite

Pour vérifier l'existence des données, on utilisera la fonction `count()`, éventuellement la condition `IS NULL`. Pour vérifier la distribution des valeurs d'un attribut, on utilisera les fonctions `min()`, `max()`, `avg()`, etc. Lorsque les valeurs sont discrètes, on pourra afficher la distribution complète, avec les effectifs pour chaque valeur de l'attribut.

⇨ Vérifiez que toutes les latitudes et longitudes sont renseignées (pas de valeur NULL)
⇨ Vérifiez le minimum et le maximum de la profondeur et de la magnitude.

L'attribut `magnitude` semble n'avoir qu'un seul chiffre après la virgule. Il doit présenter une distribution discrète.

⇨ Écrivez la requête qui donne, pour chaque valeur de magnitude, le nombre de tremblements de terre affichant cette magnitude.

Indice : on va utiliser une fonction `count()`, un `GROUP BY`, et, pour rendre le résultat plus lisible, un `ORDER BY`.

Question 5 – Recopiez :

- la requête qui construit cette distribution

- les trois premières lignes de cette distribution

⇨ Comment interprétez-vous cette distribution en termes de contrôle des données ? Plus précisément : quelles sont les hypothèses qu'on peut faire pour expliquer l'aspect de cette distribution ?
Ou plus précisémnt encore : quels phénomènes, de nature physique ou humaine, ont modelé la forme de cette distribution, en particulier en chacune de ses extrémités ?

Question 6 – Ecrivez votre réponse à cette question.

Si vous êtes très motivé, vous pouvez chercher sur internet comment transformer cette requête pour qu'elle produise un histogramme en ASCII.

3.5 Fonctions d'agrégation

Les fonctions `count()`, `avg()`, `max()`, `min()` (et quelques autres) sont des fonctions d'agrégation : lorsqu'il y a une fonction d'agrégation dans la clause SELECT, les lignes en sortie de la requête ne sont pas des lignes de la table, mais des regroupements de lignes. Tout se passe comme si plusieurs lignes étaient traitées en une seule.

La clause GROUP BY indique comment se fait le regroupement de lignes. Sans la clause GROUP BY, toutes les lignes sont regroupées en une seule ligne – c'est ce que nous avons fait avec la fonction count().

> Si je veux afficher la valeur de `magnitude` de mes regroupements de ligne, il faut que cette valeur soit commune à tous les objets de chaque regroupement. Il font donc que j'utilise un `GROUP BY` sur `magnitude`.

Si j'oublie le `GROUP BY`, SQL renvoie une erreur, et c'est normal : si je fais un `GROUP BY magnitude` et que je demande à afficher la latitude du résultat, cette demande n'a pas de sens, puisque chaque ligne du résultat mélange des tremblements de terre ayant des latitudes différentes. En revanche, la clause `GROUP BY magnitude` garantit que chaque ligne regroupe des tremblements de terre ayant tous la même magnitude.

Si je ne mets pas la clause `GROUP BY`, toutes les lignes sont agrégées en une seule, et aucun attribut n'est directement accessible (il ne peut figurer que comme paramètre d'une fonction d'agrégation).

De fait, le terme « group by » est mal choisi : il est plus intuitif de retenir que lorsqu'on met une fonction d'agrégation, on agrège toutes les lignes, en protégeant les colonnes qui figurent dans la clause `GROUP BY`.

Au risque de lasser, je vous demande de bien retenir ce fonctionnement de l'agrégation :

1. si la requête appelle une fonction d'agrégation, cette requête tend à regrouper toutes les lignes ;
2. dans ce cas, `GROUP BY magnitude` va **empêcher de regrouper** les objets dont les valeurs de magnitude sont différentes.

3.6 Le sélecteur universel

Dans les documentations et cours en ligne, on trouve souvent la syntaxe suivante :

```
SELECT * FROM table [WHERE condition]
```

Le caractère '*' (étoile, ou star, ou joker) est un sélecteur universel : il permet de sélectionner la totalité des colonnes de la table considérée, en évitant ainsi d'avoir à les nommer une à une. Dans les requêtes SQL ordinaires, vous devez NE PAS l'utiliser.

> Le sélecteur universel '*' ne doit pas être utilisée. C'est un conseil pour votre pratique professionnel, et une consigne dans le cadre de ce cours et de son évaluation. La seule exception est la définition des vues.

Pour les explications sur cette règle, voir : https://sql.sh/31-caractere-etoile (lien consulté le 3 novembre 2021). Pour l'exception, voir immédiatement ci-dessous.

4 Les vues

Attention, attention : une vue SQL est un concept purement SGBD, qui n'a rien à voir avec la visualisation des données.

4.1 A quoi servent les vues ?

Dans PostGreSQL, il est possible d'écrire des requêtes qui sélectionnent les 300 tremblements de terre de plus forte magnitude. Pour cela, on utilise les clauses « `ORDER BY` … `LIMIT` ».

Dans QGIS, il n'est pas possible de limiter l'affichage des données aux 300 tremblements de terre de plus forte magnitude, en utilisant ces clauses « `ORDER BY` … `LIMIT` ». Pour faire cette opération, il faut définir une vue dans PostGIS, et appeler cette vue dans QGIS.

> Une vue, en SQL, est une table virtuelle – on peut faire sur cette table des requêtes `SELECT`, mais pas d'insertion ni de mise à jour (requêtes `INSERT` et `UPDATE`).
> Derrière cette table virtuelle se cache une requête qui permet d'accéder aux données qui se trouvent dans les vraies tables de la base.

Pour le serveur SQL, une vue est considérée exactement comme une sous-requête. Lors de l'exécution, le nom de la vue sera remplacé par la sous-requête à laquelle elle correspond.

4.2 Comment créer une vue ?

Pour créer une vue :

- dépliez le schéma « public » de votre BD,
- clic-droit sur « vue », puis « ajouter une vue »,
- dans l'onglet « propriété », donnez-lui un nom (comme un nom de table),
- dans l'onglet « définition », entrez la requête SQL qui correspond à cette vue (requête que vous aurez testée au préalable),
- cliquez sur OK.

La vue est créée. Vous pouvez la tester, au moyen d'une requête SELECT, comme si c'était une table.

Surtout, une fois que vous avez créé une vue, vous pouvez recopier son code SQL et le modifier pour en créer de nouvelles.

⇨ Créez la vue qui correspond aux seuls tremblements de terre de magnitude supérieure à 5.5. Testez-la. Copiez son code de création. Détruisez-la.

- ⇨ Avec le code SQL de la cue précédente, créez la vue **`tdt_magni_sup`** qui correspond aux 300 tremblements de terre de plus forte magnitude.
- ⇨ Avec la fonction **`count()`**, vérifiez qu'il y a bien 300 tremblements de terre dans cette vue.
- ⇨ Ecrivez la requête SELECT qui détermine la magnitude du plus faible tremblement de terre de la vue.

Question 7 – Recopiez la requête qui crée la vue correspondant aux 300 tremblements de terre de plus forte magnitude.

4.3 Définition d'une vue et sélecteur universel

Nous avons vu plus haut que dans une requête SQL standard, l'utilisation du sélecteur universel est source de lourdeurs potentielles dans l'exécution : la recopie de toutes les colonne d'une table charge consomme inutilement la ressource. Cet argument est particulièrement fort si on tient compte de l'évolutivité des modèles de données : « Certes, aujourd'hui, ma table n'a pas beaucoup de colonnes, et donc le sélecteur universel ne coûte pas cher, mais qui sait si demain, dans une extension du modèle, on ne triplera, ou ne décuplera pas le nombre de colonnes ? »

Dans la définition d'une vue, cet argument d'évolutivité du modèle vient jouer dans l'autre sens : si on fait évoluer le modèle de la table **`tdt`**, il sera très utile d'avoir défini la vue **`tdt_magni_sup`** au moyen du sélecteur universel. En effet, si on définit spécifie les colonnes de **`tdt`** à conserver **`tdt_magni_sup`** alors :
- si on ajoute des colonnes à **`tdt`**, elles n'apparaîtront pas dans **`tdt_magni_sup`**.
- si on enlève des colonnes à **`tdt`**, leur appel dans **`tdt_magni_sup`** provoquera une erreur.

Voilà pourquoi, dans la définition d'une vue, l'utilisation du sélecteur universel est une option envisageable. En utilisant le sélecteur universel, on sera sûr qu'au fil des évolutions de schéma de la base, la vue **`tdt_magni_sup`** restera cohérente avec la table **`tdt`** dont elle dérive.

Mais l'utilisation des vues va au-delà de ce cas d'utilisation. Les vues sont utiles par exemple pour masquer la structure de la base aux utilisateurs du front office, dans une optique de sécurité, et dans ce cas, le sélecteur universel reste proscrit.

5 Les sous-requêtes

Les magnitudes et les profondeurs semblent cohérentes. Sont-elles conformes à la réalité ? Les tremblements de terre les plus violents laissent des traces ailleurs que dans les bases de données. Nous allons vérifier que certains des tremblements de terre de la base sont historiquement connus.

Pour cela, nous utiliserons les sous-requêtes.

> Une sous-requête est une requête dont le résultat est utilisé dans une autre requête (la requête principale).

Les sous-requêtes sont des extraits de table, qui sont utilisées dans une autre requête à la place d'une vraie table. C'est ce que nous allons voir d'abord. Une sous-requête peut aussi renvoyer une valeur, valeur qui est utilisée dans une des clauses de la requête principale. C'est ce que nous regarderons dans un second temps.

5.1 Sous-requête pour une table

Une sous-requête est d'abord conçue pour renvoyer un extrait de table, et être utilisée dans la requête principale comme si c'était une table. Dans ce cas, elle doit obligatoirement recevoir un alias :

```
SELECT date_heure, magnitude
FROM (SELECT * FROM tdt ORDER BY magnitude LIMIT 10) AS tmp
```

- ⇨ Faites tourner cette requête,
- ⇨ Modifiez-la pour obtenir le même résultat avec une requête plus simple.

Cette sous-requête n'apporte pas grand-chose, puisqu'on arrive à produire la même chose sans sous-requête. Nous verrons plus loin l'utilité de ce type de sous-requête, comme alternative aux jointures.

5.2 Sous-requête pour une valeur

Si la valeur renvoyée par la sous-requête est unique (une seule ligne et une seule colonne), alors elle peut être utilisée comme une valeur.

Nous allons commencer avec un exemple artificiel : extraire identifiants, coordonnées et date du tremblement de terre de plus forte magnitude.

La requête et la sous-requête auront la forme suivante :

```
SELECT attributs
FROM tdt
WHERE magnitude = (SELECT la plus forte magnitude)
```

- ⇨ Modifiez cette requête pour qu'elle fonctionne. Faites-la tourner.

Pour trouver le tremblement de terre de plus forte magnitude, la vraie solution est celle que nous avons vue plus haut : `SELECT attributs FROM tdt ORDER BY magnitude DESC LIMIT 1`. Les deux requêtes donneront le même résultat, sauf s'il y avait des ex-aequo pour la place de premier.

En revanche, la solution "`ORDER BY colonne LIMIT n`" ne fonctionne pas si on cherche tous les tremblements de terre dont la magnitude est inférieur de 0.3 points au plus à la magnitude la plus forte.
La requête et la sous-requête deviennent alors :

```
SELECT attributs
FROM tdt
WHERE magnitude > (SELECT la plus forte magnitude – 0.3)
```

- ⇨ Modifiez cette requête pour qu'elle fonctionne. Faites-la tourner.

⇨ Vérifiez sur internet quelques tremblements de terre réels correspondant à ceux renvoyés par la requête.

Naturellement, on aurait pu écrire directement : **`magnitude=8.9-0.8`**, ou : **`magnitude=8.1`** mais la requête est alors moins générique : elle suppose qu'on connaît préalablement la magnitude maximale ; par ailleurs, on ne peut pas l'adapter facilement aux autres attributs, comme la profondeur (et puis j'avais besoin de cet exemple pour illustrer les sous-requêtes). On verra d'autres exemples de sous-requêtes plus loin.

Attention, si vous attendez une seule valeur, et que la sous-requête en envoie plusieurs, la valeur ne sera pas valide.
Comme ce n'est pas une erreur de syntaxe, l'erreur sera déclenchée au cours de l'exécution de la requête, et non pas au début, lors du contrôle de la syntaxe (cf. plus bas la partie sur le plan de requête et la clause **`EXPLAIN`**).

5.3 Exercice

Pour aller plus loin, nous allons utiliser les vues. Notre objectif est de reprendre la définition de la vue **`tdt_magni_sup`** afin de lui donner une forme déterministe.

5.3.1 Position du problème

Cette requête, telle que nous l'avons définie, présente une grosse faille : lorsque je prends les 300 tremblements les plus violents, le 300^{e} a une magnitude de 6.4, mais le 304^{e} aussi. Rien ne permet de garantir que si l'on refait la même requête sur un autre ordinateur, ou un autre jour, le 304^{e} ou le 306^{e} d'aujourd'hui ne sera pas choisi à la place du 300^{e}. Cela peut modifier en aval les calculs qu'on fait sur la vue, par exemple la profondeur moyenne. Autrement dit, notre requête n'est pas déterministe, et son résultat dépend de la façon dont les objets sont rangés dans la base.

Il y a au moins deux façons de changer la requête pour la rendre déterministe :

- on peut ajouter un critère dans le tri, par exemple le critère de profondeur. Ainsi, à magnitude égale, on considérera comme plus dangereux un tremblement de terre moins profond ;
- ou bien on considère que le nombre de tremblements de terre ne doit pas être contraint à l'unité près : notre requête devient « prenons les "*environ 300"* tremblements de terre de plus forte magnitude », ou dit plus précisément : « prenons les tremblements de terre dont la magnitude est supérieure ou égale à celle du 300^{e} tremblement de terre de plus forte magnitude ».

Attention, **ces deux méthodes changent ce que nous demandons au serveur**, et c'est normal : si la requête que nous avons définie précédemment n'est pas déterministe, c'est parce que l'objet de la requête lui-même, « les trois cents tremblements de terre de plus forte magnitude », n'est pas déterminé. C'est pour fixer cette indétermination que sont conçues les deux méthodes ci-dessus.
Nous allons choisir la deuxième méthode, celle qui nous conduit à accepter de prendre un peu plus de 300 tremblements de terre.

5.3.2 Mise en oeuvre

- ⇨ Recopier la requête SQL créant la vue **`tdt_magni_sup`**. Renommer la vue **`tdt_magni_sup en tdt_magni_sup_v0`**.

- ⇨ En vous aidant de la documentation, utilisez la clause OFFSET pour trouver la magnitude du 300e tremblement de terre le plus violent.

- ⇨ Ecrivez la vue **`tdt_magni_sup`**, en mettant en sous-requête le code qui produit la magnitude seuil. On écrira donc : **`SELECT xxxx WHERE magnitude>(SELECT xxxx)`**

Question 8 – Recopiez la requête qui crée la vue modifiée.

- ⇨ En vous inspirant du code de la vue **`tdt_magni_sup_v0`**, écrivez la requête qui calcule le nombre, la moyenne, le minimum et le maximum des profondeurs des tremblements de terre de plus forte magnitude.

- ⇨ Dans la requête précédente, remplacez la vue par la sous-requête correspondante, et vérifiez que le résultat ne change pas.

Pour vérifier plus facilement que les deux requêtes donnent le même résultat, vous pouvez les lancer en même temps en les reliant avec la clause UNION.

Attention, UNION supprime les doublons : l'équivalence des deux requêtes est prouvée par le retour d'une seule ligne. Si vous voulez faire apparaître les deux résultats, il faut leur ajouter une colonne :
SELECT 1, attributs FROM etc. UNION SELECT 2, attributs FROM etc.

6 Les géométries

Nous avons des données, mais ce sont des données sémantiques : si vous tentez de les afficher dans QGIS, elles apparaîtront comme des feuilles excel. Pour gérer la géométrie de nos objets dans PostGreSQL, voici donc notre premier contact avec PostGIS.

Nous allons d'abord voir comment créer des géométries à partir des informations sémantiques de latitude et de longitude.
Dans la plupart des cas, nous importons des données qui sont déjà géographiques, par exemple des fichiers shape, et c'est cela que nous explorerons dans un second temps.

6.1 Le type point de PostGreSQL : un piège

La connaissance de la latitude et la longitude permet de positionner un tremblement de terre, mais d'un point de vue conceptuel, ces deux valeurs sont en fait les deux composantes d'une réalité géométrique : le point. Dans PostGreSQL, il existe un type point qui permet de stocker sur une seule colonne les deux valeurs de latitude et de longitude.

- ⇨ Explorer la documentation de PostGreSQL à la recherche des fonctions qui utilisent ce type point.

Comme vous pouvez le constater, le type point de PostGreSQL est dépourvu des fonctions de manipulation qu'on attend généralement d'un SIG. Par ailleurs, PostGreSQL n'a pas de type pour manipuler des géométries plus complexes (lignes et surfaces).

> Le type **`point`** n'est pas utilisable pour une application manipulant massivement des géométries d'objets.

Nous avons typiquement ici un exemple de cul-de-sac logiciel : le type point n'a jamais été vraiment développé par le projet PostGreSQL. L'apparition de vraies fonctions géométriques dans PostGreSQL s'est faite dans un projet différent, PostGIS, la cartouche spatiale de PostGreSQL. Le type **`point`** de PostGreSQL est tombé en désuétude, et n'est conservé que pour des raisons de compatibilité ascendante.

Question 9– Cherchez ce que signifie « compatibilité ascendante » et résumez-le en 30 mots.

> Le type **`point`** de PostGreSQL ne doit JAMAIS être utilisé.

Dans la suite du TP, nous travaillerons principalement avec le type **`geometry`** de PostGIS. Il comporte un sous-type **`POINT`** qui ne doit pas être confondu avec le **`type point`** de PostGreSQL.

6.2 Le type geometry

6.2.1 Créer une colonne de type geometry

Le type **`point`** n'est pas un type de PostGIS : il appartient à PostGreSQL. Comme PostGIS est la vraie composante spatiale de PostGreSQL, le type **`point`** n'a pas subi les développements et les optimisations que nous allons trouver dans PostGIS.

Pour profiter de la puissance de PostGIS, nous allons maintenant rajouter une colonne de type **`geometry`** dans notre table **`tdt`**.

Attention : la structuration de notre BD n'est pas optimale. Dans une BD optimale, nous éviterons de dupliquer l'information géométrie sous la forme 1- latitude/longitude, 2- type **`geometry`** de PostGIS.
La redondance de nos géométries est ici purement pédagogique : elle va nous permettre de faire des comparaisons entre les différents formats de nos colonnes.

Pour créer une colonne de type geometry, on peut utiliser la commande SQL suivante :

```
SELECT AddGeometryColumn( 'nom_de_la_table',
                          'nom_de_la_colonne',
                           srid,
                          'type_de_géométrie',
                           dimension );
```

avec :

- Le SRID est un entier, identifiant de la projection utilisée. Sauf précision contraire, nous n'utiliserons dans ce TP que la projection plate carrée (coordonnées latitude et longitude de WGS84), dont l'identifiant est 4326.

- La dimension est 2 ou 3, selon que les données sont en 2D ou en 3D. Ici, la dimension sera toujours 2.
- le type de géométrie (ou plutôt le sous-type puisque le type dans postgreSQL est **geometry**) sera ici le sous-type '**POINT**'.

Il est possible de créer une colonne de type **geometry** directement, mais la fonction **AddGeometryColumn** présente deux avantages :

- elle est compatible avec les anciennes versions de PostGIS, dans laquelle elle était obligatoire.
- elle comporte dans ses paramètres toutes les informations nécessaires à la création d'une géométrie optimale, compatible avec toutes les applications qui se brancheront ensuite sur PostGIS.

Un peu d'histoire : à l'origine, la fonction **AddGeometryColumn** permettait de référencer la colonne géométrique dans la table **geometry_columns**. Cette dernière est utilisée par certaines applications qui suivent les recommandations de l'Open Geospatial Consortium (OGC). A partir de la version 2 de PostGIS, cette table **geometry_columns** a été transformée en une vue, qui collecte automatiquement dans la base toutes les informations sur les colonnes géométriques.

A quoi sert la vue **geometry_columns** ? Elle met en forme les méta-données sur les colonnes de type **geometry**. Ces méta-données sont requises par certaines applications qui suivent les recommandations de l'OGC.

- ⇨ Dans l'éditeur SQL de PgAdmin, écrivez la commande SQL en utilisant la fonction **AddGeometryColumn** pour créer **geom**, une colonne de type **geometry** sur la table **tdt**.
- ⇨ Dans pgAdmin (fenêtre principale) vérifiez que cette colonne a été créée (Refresh) et regardez dans le panneau SQL (en bas à droite de l'interface) la syntaxe de cette création de colonne.
- ⇨ Reprenez cette syntaxe pour créer créez une colonne **geom2** de type **geometry** , reprenant la définition de la colonne **geom** dans pgAdmin.
- ⇨ Lisez le contenu de la vue **geometry_columns**. Vérifiez que **geom** et **geom2** sont référencées.
- ⇨ Supprimez la colonne **geom2**.

> Pour créer une colonne de type **geometry**, il est utile d'utiliser la fonction **AddGeometryColumn**, car sa syntaxe est facilement accessible en ligne. L'autre possibilité est d'imiter la syntaxe d'une création de colonne géométrique déjà existante, en la recopiant dans le panneau SQL.

Question 10 – Recopiez :

- l'instruction utilisant la fonction AddGeometryClumn qui vous a permis de créer la colonne geom.

- l'instruction ALTER TABLE en laquelle PostGIS a traduit l'instruction que vous avez utilisée

(on trouvera cette traduction dans PgAdmin).

6.2.2 Remplir une colonne de type geometry

Nous avons créé la colonne géométrique, mais cette colonne est vide (ses valeurs sont toutes `NULL`). Il faut maintenant la remplir, avec des objets géométriques construits à partir des informations portées par les colonnes `latitude` et `longitude`.

- ⇨ Cherchez une voie directe : elle est facile à trouver sur internet
- ⇨ Cherchez dans la documentation de PostGIS le format d'entrée d'un point dans une colonne de type `geometry` (indice : WKT, `geomFromText`)
- ⇨ pour construire un WKT, paramètre de la fonction `geomFromText`, vous devez concaténer des chaînes de caractères. Cherchez l'opérateur de concaténation de PostGreSQL.

Attention, vous ne pouvez pas tester ce format WKT dans une case du tableau de données, car le type `geometry` n'est pas directement lisible dans pgAdmin : il est codé en binaire.

- ⇨ Ecrivez la requête `UPDATE`, et faites-la tourner pour remplir la colonne `geom`.

Question 11 – Recopiez la requête UPDATE de la colonne `geom`, utilisant la fonction GeomFromText.

Pour vérifier que l'opération s'est bien passée, nous ferons un affichage graphique plus tard. Pour le moment, vérifions simplement dans pgAdmin que la colonne `geom` est désormais remplie (avec du WKB, la transcription en binaire du WKT).

6.3 Type Geometry et OGC

Vous devez tout savoir sur l'OGC, sur le type geometry, et sur ses sous-types.

Sur l'OGC, le minimum à lire :
* https://fr.wikipedia.org/wiki/Open_Geospatial_Consortium
la page en anglais, moins bonne, est néanmoins complémentaire :
* https://fr.wikipedia.org/wiki/Open_Geospatial_Consortium
(liens contrôlés le 3 nov. 2021)

Sur les différentes normes produites par l'OGC :
http://www.opengeospatial.org/docs/is
(lien contrôlé le 3 nov. 2021)
Sur les géométries, cherchez « SFS » dans la page précédente : la norme SFS est en deux parties. Parcourez le document pour voir à quoi ressemble une norme, et regardez les figures pour prendre connaissance des modèles de données qui définissent les géométries.

Pour un résumé et le détail de l'implémentation dans PostGIS :
https://postgis.net/docs/using_postgis_dbmanagement.html#RefObject
(lien contrôlé le 3 nov. 2021)

Question 12 – Le modèle comporte les sous-types point, line, et quelques autres. En explorant la documentation de PostGIS, donnez tous ceux qui sont disponibles dans PostGIS. Si certains de ces types sont absents de PostGIS, donnez-en au moins un exemple.

6.4 Intégration d'un fichier shape

Nous allons maintenant ajouter à notre base un fond de carte qui permettra de replacer les tremblements de terre par rapport aux continents.

Si nous affichions nos données maintenant, elles seraient très peu lisibles : une collection de points dont la signification et la position terrestre seraient très difficiles à analyser. Pour donner du sens à nos données, nous allons les placer sur un fond de carte. Ce fond de carte est un produit dérivé par simplification du fichier **`gadm.shape`** de l'université de Berkeley.

Dans l'invite de commande (accessible, entre autre, dans le menu *Système Windows*), nous allons lancer le programme shp2pgsql. La syntaxe de l'instruction shp2pgsql est la suivante :

```
shp2pgsql -s 4326 -c -D -i -I D:\partage\gadm\MondeGadm.shp monde > monde.sql
```

Cette instruction (à taper sur une seule ligne) prend en entrée le fichier **`MondeGadm.shp`** et écrit les instructions qui vont permettre de construire la table **`monde`**. Ces instructions sont enregistrées dans le fichier **`monde.sql`**.

Il faut ensuite intégrer ce fichier **`monde.sql`** dans la base **`tremblement_de_terre`** :

```
psql -d tremblement_de_terre -h localhost -U postgres -f monde.sql
```

- ⇨ Dans l'invite de commande de PostGresSQL, faites tourner les deux instructions précédentes.
- ⇨ En vous aidant de sa commande d'aide (**`shp2pgsql ?`**), déterminez la signification de l'option **`-I`** de **`shp2pgsql`**.

Question 13 – Cherchez dans `pgAdmin` le nom de l'objet créé par l'option `-I`.

Notez qu'on peut condenser les deux commandes en une seule, en utilisant | (opérateur de redirection, à prononcer « païpe »), ce qui fait l'économie du stockage du fichier **`monde.sql`**. En revanche, durant toute la phase de débugage, il est utile de maintenir les deux commandes séparées, pour mieux détecter les sources d'erreurs, et pour disposer du fichier **`monde.sql`** dans lequel on peut aussi sourcer certaines erreurs.

Autre solution : PostGIS ShapeFile and DBF Loader/Exporter

Recherchez et ouvrez l'application PostGIS ShapeFile and DBF Loader/Exporter.

Cette application vous permet de vous connecter à une base de données PostGreSQL, intégrant l'extension PostGIS, et d'y importer le contenu de fichier shapefile ou dbf en tant que nouvelle table.
Re-importez le contenu du shapefile gadm au moyen de cet outil. Que constatez-vous au niveau des tables créées au moyen de ces deux méthodes.

7 Visualisation

7.1 Visualisation sous QGIS

- ⇨ Ouvrez QGIS.
- ⇨ Dans QGIS, ouvrez une connexion avec la base de données (ajouter une couche / couche PostGIS).

Note : vous voyez apparaître la liste des couches géométriques disponible dans votre base de données. QGIS a constitué cette liste en interrogeant la vue **`geometry_columns`** décrite plus haut.

- ⇨ Choisissez les tables et les colonnes que vous voulez afficher.
- ⇨ Contrôlez que les données sont cohérentes (accumulation des tremblements de terre sur les limites de faille, par exemple sur l'Indonésie et la Cordillère des Andes).

- ⇨ Si les données ne sont pas cohérentes, cherchez la source du problème.

- ⇨ Limitez l'affichage des données aux seuls tremblements de terre de magnitude supérieure à 5.5.

L'affichage permet de découvrir un piège grossier dans lequel tombe 70% des élèves (et de enseignants). Si vous n'êtes pas tombé dans le piège, allez voir vos camarades pour comprendre de quoi il s'agit. Si vous êtes tombés dans le piège, et que vous avez compris le bug, vous pourriez corriger le code en amont, et reconstruire vos données.

Comme il arrive que ce bug apparaisse pour des géométries plus complexes, il existe une fonction postGIS qui corrige les coordonnées. Trouvez cette fonction.

Question 14 – documentez le bug, et donnez le nom de la fonction PostGIS qui permet de le corriger.

7.2 Visualisation en KML

KML est le format standard de Google Earth.

- ⇨ Trouvez sur internet la méthode pour faire un export de PostGIS au format KML.
- ⇨ Affichez dans Google Earth les 100 tremblements de terre de plus grande magnitude

8 Jointures spatiales

Nous allons parler dans cette partie des jointures spatiales, qui constituent une classe particulière de jointures.

> Une jointure est un mécanisme permettant d'extraire de l'information qui se trouve répartie dans deux tables différentes.

> Plus exactement, la jointure est le mécanisme qui consiste à apparier les objets d'une table avec les objets d'une autre table.

> Le concept de jointure, très important en SQL et plus largement en base de données relationnelle, n'est pas abordé ici. Ce concept constitue un prérequis de ce cours, et je vous encourage vivement à l'acquérir si vous ne le maîtrisez pas encore.

Si vous connaissez le concept de jointure, mais que vous hésitez sur la syntaxe SQL, la suite de ce TP devrait vous suffire pour réactiver cette pratique. Il n'y a pas de grande différence de suntaxe entre une jointure sémantique et une jointure spatiale.

Les données que nous manipulons dans ce TP ne nous permettent pas d'utiliser des jointures sémantiques : nous n'avons que deux tables, sans relation entre elles sinon par leur appartenance à l'espace terrestre.

Ce sont donc des jointures spatiales que nous allons maintenant mettre en œuvre.

8.1 Jointures dans une requête SELECT

Nous allons appliquer le concept sur un exemple simple : je cherche à faire la différence entre les tremblements de terre terrestres et les tremblements de terre sous-marins. L'information sur les tremblements de terre se trouve dans la table **`tdt`** (ou dans la vue **`tdt_magni_sup`**, car nous allons nous restreindre aux tremblements de terre les plus puissants).

Mais pour connaître la nature terrestre ou sous-marine du tremblement de terre, nous devons aller chercher l'information dans la table **`monde`**.

Le mécanisme qui permet d'interroger les deux tables en même temps s'appelle la jointure. Pour extraire la liste des tremblements de terre terrestre, je vais sélectionner ceux qui sont à l'intérieur du contour d'un pays.

Ma requête de jointure va alors s'écrire :

```
SELECT t.id, t.geom, t.magnitude
FROM tdt_magni_sup AS t, monde AS m
WHERE ST_Within( t.geom, m.geom )
```

Analysons les différentes clauses de cette requête, en commençant par le **`FROM`** :

- **`FROM tdt_magni_sup AS t, monde AS m`** : la clause **`FROM`** comporte ici deux tables, **`tdt_magni_sup`** et **`monde`**. La clause **`AS`** sert à définir, pour chaque table, un alias. Pour le SGBD, l'utilisation des alias est transparente. C'est en revanche un confort de lecture pour l'utilisateur, le rédacteur, le relecteur : une ou deux lettres seulement pour les table, et les noms complet pour les colonnes, c'est une convention qui simplifie fortement la compréhension des requête complexes.
- **`SELECT t.id, t.geom, t.magnitude`** : la liste des colonnes que nous allons garder. Pour indiquer au SGBD dans quelle table se trouve chacune des colonnes recherchées, nous préfixons le nom de la colonne avec l'alias de la table.
- **`WHERE ST_Within( t.geom, m.geom )`** : ceci est la condition de jointure. La fonction **`ST_Within()`**, offerte par PostGIS, permet de déterminer si une géométrie (le premier

paramètre de la fonction) est incluse dans une seconde géométrie (le second paramètre). Notez qu'ici aussi les noms de colonne sont préfixés par les noms de table.

Attention :

- **tdt_magni_sup** n'est pas une table, c'est une vue, mais tant qu'on reste dans une requête **SELECT**, les vues sont utilisées comme des tables.
- Lorsqu'on met un alias sur une table, on n'a plus le droit d'utiliser le nom de la table : dans toute la requête, l'alias doit remplacer le nom de la table.
- En général, on n'est pas obligé non plus de préfixer les noms de colonnes avec les noms de leur table, mais cela facilite la lisibilité du code. Il est donc toujours recommandé de le faire (et dans certains cas, c'est obligatoire).
- et voici justement l'un des cas où l'on est obligé de préfixer les noms de colonnes, c'est lorsque le nom de colonne apparaît dans les deux tables. Si on ne préfixe pas la colonne, le serveur ne peut pas décider laquelle des deux colonnes il doit choisir (on dit qu'il ne peut pas *résoudre l'ambiguïté*). C'est courant avec les colonnes **nom**, **geom**, **id**, et **gid**, mais aussi **proprietaire**, **date**, etc.
- Un autre cas qui oblige à mettre un alias, c'est le cas de l'auto-jointure, c'est-à-dire lorsque la table figure deux fois dans la jointure. Nous verrons ce cas un peu plus loin.

Revenons à notre requête :

```
SELECT t.id, t.geom, t.magnitude
FROM tdt AS t, monde AS m
WHERE ST_Within( t.geom, m.geom )
```

⇨ Testez cette jointure dans pgAdmin.
⇨ Transformez cette jointure pour qu'elle renvoie, outre les colonnes déjà spécifiées, le nom du pays dans lequel a eu lieu le tremblement de terre.

Question 15 – Recopiez la requête modifiée

8.2 Jointures en JOIN … ON

L'inconvénient de la syntaxe de jointure étudiée ci-dessus, c'est qu'elle mélange toutes les tables dans la clause **FROM**, et qu'elle mélange, dans la clause **WHERE**, toutes les conditions de jointures avec les conditions de filtre.

⇨ Etudiez la documentation de la syntaxe **JOIN … ON**.
⇨ Traduisez la sous-requête précédente dans la syntaxe **JOIN … ON**.

Quelle est la différence entre les deux jointures ?

1. En dehors des requêtes non-déterministes, le résultat des deux requêtes est le même.
2. La syntaxe **JOIN … ON** est réputée légèrement plus facile à optimiser pour le serveur.
3. Surtout, avec la séparation des conditions de jointure et de filtre, cette syntaxe offre à l'utilisateur une bien meilleure lisibilité du code.

8.3 Jointure négative

Revenons à notre précédente requête :

```
SELECT t.id, t.geom, t.magnitude
FROM tdt AS t, monde AS m
```

```
WHERE ST_Within( t.geom, m.geom )
```

⇨ Transformez cette jointure pour qu'elle renvoie le **nombre** de tremblements de terre, parmi les 300 plus forts, qui sont terrestres.

Nous voudrions maintenant nous intéresser à ceux des tremblements de terre, parmi les 300 plus forts, qui sont sous-marins. En SQL, l'opérateur de négation est `NOT`. Pensez-vous que l'on puisse simplement transformer la condition de jointure `ST_Within()` en `NOT ST_Within()` ?

⇨ Testez la condition de jointure de `ST_Within()` en `NOT ST_Within()`. Combien de tremblements de terre sous-marins compterions-nous avec cette méthode ?
⇨ Vérifiez le résultat : la somme des deux fait-elle environ 300 ?

Question 16 – Ecrivez le nombre de tremblements de terre, parmi les 300 plus forts, qui sont terrestres. Ecrivez le nombre de tremblements de terre que l'on trouve si l'on transforme `ST_Within()` en `NOT ST_Within()`.

La plupart des étudiants qui ne savent pas ce qu'est une jointure trouvent ici un résultat aberrant, et s'en étonnent. Comment est calculée la jointure ?

- Pour la première requête, le SGBD considère tous les couples (tremblement de terre ; contour de pays) et il ne garde que ceux pour lequel le tremblement de terre est à l'intérieur du pays. Le SGBD ne retient que les tremblements de terre associés à un contour, et comme les contours de pays sont disjoints, chaque tremblement de terre n'est retenu qu'une seule fois.
- Pour la deuxième requête, si vous remplacez la condition `ST_Within()` par une condition `NOT ST_Within()`, le SGBD va considérer tous les couples. Or, pour chaque tremblement de terre, on va trouver un grand nombre de pays dans lequel ce tremblement de terre **ne se trouve pas** : à chaque fois, le couple (tremblement de terre ; contour de pays) qui correspond à ce critère sera retenu.

Vous devez réfléchir à votre condition de jointure en gardant en mémoire le mécanisme de jointure décrit ici : le SGBD prend en compte **toutes les paires** que l'on peut construire avec les éléments des deux tables, et **élimine** les paires qui **ne respectent pas** la condition de jointure.

Pour vous le représenter :

⇨ Prenez un crayon.
⇨ Dessiner un tableau avec quinze lignes et dix colonnes, comme s'il y avait quinze contours de pays, et dix tremblements de terre. Nous allons travailler sur cette simulation réduite de nos données.

Attention, ce tableau est purement illustratif, pédagogique. Il n'a pas de réalité comme objet. En particulier, ce *tableau* n'est pas une *table* SQL. Il représente de manière concrète toutes les combinaisons abstraites qu'on pourrait construire si on combinait un objet d'une table avec un objet de l'autre table. L'ensemble de toutes ces combinaisons s'appellent le « produit cartésien » des deux tables.

Parmi les dix tremblements de terre, imaginons qu'il y en a quatre qui sont terrestres. Ils appartiennent donc chacun à un contour.

⇨ Dans votre tableau, mettez une croix dans quatre cases, chaque case prise dans une colonne différente, pour indiquer que ces cases vérifient la condition de jointure **`ST_Within(t.geom,m.geom)`**.

Toutes les autres cases, celles qui n'ont pas de croix, correspondent à la condition de jointure négative : **`NOT ST_Within(t.geom,m.geom)`**.

Elles sont au nombre de **`10*15-4=146`**. Si on procède ainsi pour avoir les tremblements de terre sous-marins, on tombe évidemment sur une aberration. Au lieu de compter tous les tremblements de terre sous-marins, on a compté des pairs d'objet (**`tdt`**,**`monde`**) y compris des objets sans signification, comme un tremblement de terre au Japon apparié avec le contour de l'Irlande (et chaque tdt est compté plusieurs fois, une fois pour chacun des contours de la base qui ne satisfait pas la condition de jointure – par exemple le même tremblement de terre japonais sera compté successivement avec tous les contours auxquels il n'appartient pas).

Retenons cette règle, qui vaut aussi bien pour les jointures géométriques que sémantiques :

> Une condition de jointure négative est presque toujours une erreur.

⇨ Alors, comment allons-nous trouver les tremblements de terre maritimes ? Cherchez et trouvez.

Indice : avec une sous-requête, on va chercher.... (compléter la suite).

⇨ Ecrivez le système requête-sous-requête sous les deux formes de jointure (« **`FROM table1, table2 WHERE`** », et « **`FROM table 1 JOIN table2 ON`** »).

8.4 Jointure à gauche

Cherchez dans la documentation ce qu'est une jointure à gauche.

⇨ Modifiez la vue **`tdt_magni_sup`** pour que cette vue offre les 300 tremblements de terre les plus forts, avec, pour ceux qui sont terrestres, le nom du pays où ils ont eu lieu. Pour les tremblements de terre maritime, le nom de pays doit être nul.

⇨ Modifiez la requête précédente pour que la vue **`tdt_magni_sup`** renvoie, pour les tremblements de terre maritimes, à la place du nom de pays, la chaîne de caractère 'maritime'.

Indice : vous utiliserez la fonction `COALESCE()`.

8.5 Jointures dans une requête UPDATE

C'est dans les requêtes **`SELECT`** que les jointures sont les p lus utilisées, mais il est parfois nécessaire de les utiliser aussi dans les requêtes **`UPDATE`** et **`DELETE`**.

Il est possible de faire des jointures dans une requête UPDATE, avec les limitations suivantes :

- La mise à jour ne pourra concerner qu'une seule table.
- Cette table doit être une vraie table. Pas de mise à jour sur une vue !
- La syntaxe **JOIN** … **ON** n'est pas utilisable dans une requête **UPDATE**.

Les deux premières limitations sont évidentes, conséquences directes de la nature même de la mise à jour. La limitation sur la syntaxe est une spécificité de PostGreSQL. En mySQL, la syntaxe **JOIN** … **ON** est valide.
Par ailleurs, tout ce qu'on dit ici sur les mise-à-jour s'applique aussi sur les délétions.

Nous allons voir cela tout de suite sur un exemple : pour pouvoir sélectionner les tremblements de terre sous-marins, nous allons ajouter dans la table **tdt** une colonne qui portera l'information terrestre/sous-marin. Cette colonne sera nommée **terrestre**, et sera de type booléen. Puis nous remplirons cette colonne au moyen d'une requête de mise à jour, qui utilisera une jointure sur la table **monde** :

```
UPDATE tdt
SET terrestre=true
FROM monde
WHERE ST_Within(tdt.geom, monde.geom)
```

- ⇨ Dans la table **tdt**, ajoutez une colonne **terrestre**, de type booléen (boolean).
- ⇨ Faites tourner la requête **UPDATE** ci-dessus.
- ⇨ Ecrivez et faites tourner la requête **UPDATE** qui va mettre à **false** les valeurs qui ne sont pas déjà à **true**.

- ⇨ Vérifiez vos résultats par un affichage sous QGIS.

Les tremblements indiqués comme maritimes et terrestres se superposent-ils bien avec les mers et les terres de la table monde ?

9 Scripts PL/pgSQL

Si l'on voulait effectuer les tâches effectuer par SQL au moyen de langages de programmation, il faudrait parcourir les tables avec des boucles. Le principe du langage SQL est de donner au serveur des instruction portant sur les données, et c'est le serveur qui traduit ces requêtes sous la forme de boucles. L'utilisateur en est déchargé. Le langage SQL se substitue ainsi largement aux langages de programmation.

Il y a en revanche des cas où l'on peut souhaiter répéter plusieurs fois une opération que le langage SQL ne sait pas répéter. Il serait possible d'utiliser un script externe, qui répètera plusieurs fois l'activation d'une commande SQL. Cette méthode, parce qu'elle implique un échange systématique des données entre le script externe et le SGBD, est coûteuse en temps. Il est plus intéressant de faire tourner un script directement dans le SGBD.

> PostGreSQL possède un langage de script : PL/pgSQl.

Ce langage permet de définir des fonctions, que l'on peut appeler ensuite dans n'importe quelle expression, par exemple avec une requête **SELECT**.

Pour faire des tests de charge de PostGIS, nous allons peupler la base avec des enregistrements fictifs, pour lui faire contenir environ un million d'objets.

- ⇨ Adaptez le code suivant à la table des tremblements de terre :

```
CREATE OR REPLACE FUNCTION ajoute_valeurs_aleatoires(nbre integer)
RETURNS void AS $$
DECLARE
    -- declarations
BEGIN
  FOR k IN 1 .. nbre LOOP
        FOR i IN -179 .. 179 LOOP
              FOR j IN -89 .. 89 LOOP
INSERT INTO table(x,y) VALUES ( i+random() , j+random() ) ;
              END LOOP ;
        END LOOP ;
  END LOOP ;
END;
$$ LANGUAGE plpgsql;
```

⇨ Ajoutez à la fin du code les instructions qui permettent de remplir la colonne **`geom`** avec les valeurs qui se trouvent dans les colonnes **`latitude`** et **`longitude`**.

Pour appeler cette fonction qui ne renvoie pas de résultat, on utilise la syntaxe suivante :

```
SELECT ajoute_valeurs_aleatoires(3) ;
```

⇨ Déterminez la valeur du paramètre à passer à cette fonction pour obtenir entre deux et dix millions d'enregistrements. Lancez la fonction avec cette valeur.

Aïe ! Une erreur.

⇨ Analysez le message d'erreur, et réglez le problème. Relancez la fonction pour obtenir entre dix et trente millions d'enregistrements.

⇨ Vérifiez le nombre d'objets dans la table des tremblements de terre.

Question 17 - Ecrivez le nombre d'objets qui sont maintenant dans la table des tremblements de terre.

10 Les index

Les index sont un point technique vital et pourtant souvent négligés dans l'apprentissage des bases de données.

Un index est un mécanisme qui accélère le filtrage des objets à partir d'un attribut. Une bonne gestion des index est nécessaire pour donner à votre base de données une efficacité optimale. A contrario, si vous ne définissez pas d'index, ou bien si vous les définissez mal, certaines requêtes sur votre base de données tourneront de manière très ralentie, jusqu'à bloquer votre architecture logicielle.

10.1 Présentation

Pour mettre un index sur la colonne **`magnitude`** :

- on déplie la table tdt, on déplie « index ».
- clic-droit sur « index », « ajouter un index »

- onglet « propriété » : on donne un nom à cet index (par exemple **`k_magnitude`**) et on choisit la méthode d'accès « btree »
- onglet « colonnes » : on sélectionne magnitude, et on clique sur « ajoute ».
- on valide avec OK

⇨ Relancez la requête qui calcule le maximum et le minimum de la profondeur.
⇨ Notez le temps d'exécution (en bas à droite de la fenêtre). En relançant la requête, vérifiez que ce temps d'exécution est stable.
⇨ En suivant les instructions ci-dessus, mettez un index sur les colonnes **`magnitude`** et **`profondeur`**.
⇨ Relancez la requête pour vérifier si ce temps a diminué.

Normalement, le gain de temps est faible ou imperceptible. Dès qu'on procède à des requêtes plus complexes, le gain apporté par les index augmente.

> Le professeur n'est pas là simplement pour surveiller le travail. Il a parfois des choses importantes à vous raconter. C'est le cas sur les index. Veillez à ce qu'il n'oublie pas de vous en parler.

⇨ Pour la table **`tdt`**, vérifiez que l'index sur la colonne **`magnitude`** existe, et qu'il n'y en a pas sur la colonne **`geom`**.
⇨ Pour la table **`monde`**, vérifiez l'existence d'un index sur la colonne **`geom`**. Est-ce un index basé sur un arbre binaire ?

10.2 La clause EXPLAIN

⇨ Ecrivez « **`EXPLAIN`** » au début d'une des deux requêtes. Relancez. Regardez le résultat.
⇨ Enlevez « **`EXPLAIN`** ». Vérifiez le résultat.

Question 18 - En vous aidant de la documentation, expliquez précisément à quoi sert l'instruction `EXPLAIN`.

10.3 Comparaison expérimentale

Nous commençons par deux mises en garde :

> Attention : les manipulations et les mesures demandées dans cette partie sont un peu répétitives, et demandent de votre part de travailler avec soin.

et :

> Lorsque le TP vous demande de noter un temps d'exécution, vous ne devez avoir aucun autre programme qui tourne sur votre machine. En particulier les programmes de messageries, et tout autre programme faisant des accès réseaux inopinés doivent être éteints. De même, efforcez-vous de maintenir l'état de votre machine constant durant toute la durée de

cette partie, pour que les temps d'exécution mesurés aient la plus grande signification possible.

10.3.1 Les requêtes à comparer

Dans cette partie, nous allons utiliser le gestionnaire de requête SQL, que l'on ouvre avec le bouton « SQL » de pgAdmin :

- ⇨ écrivez une requête qui permette de sélectionner les tremblements de terre qui ont eu lieu entre 10 et 11 degrés de **`latitude`** et entre 20 et 21 degrés de **`longitude`**.
- ⇨ Ouvrez une seconde fenêtre de requête SQL. A l'aide de la documentation de postGIS, écrivez la même requête, mais en faisant porter la sélection sur la colonne **`geom`** de type **`geometry`**. Vous DEVEZ utiliser une fonction de test géométrique de PostGIS. Parmi ses paramètres je vous impose d'utiliser la fonction **`geomFromText`**, pour vous apprendre à rédiger une géométrie en WKT.

Note pour les enseignants : la syntaxe archaïque **`WHERE geom && 'WKT'`** semble valide, mais plante lorsqu'on modifie les index. A éclaircir.

Question 19 - Lorsque les deux requêtes fonctionnent correctement, notez les – conservez les aussi dans pgAdmin (elles vont servir tout de suite).

10.3.2 Les index à comparer

- ⇨ Essayez de créer un index **`btree`** sur le couple **`(longitude,latitude)`**,
- ⇨ essayez de créer un index **`gist`** sur le couple **`(longitude,latitude)`**,
- ⇨ essayez de créer un index **`btree`** sur la colonne **`geom`**,
- ⇨ essayez de créer un index **`gist`** sur la colonne **`geom`**.

Que constatez vous ?
Sur le couple **`(longitude,latitude)`**, on utilisera donc le **`btree`**, et sur la colonne **`geom`**, on utilisera un index **`gist`**.

Pour accélérer les manipulations, il sera utile de créer et supprimer ces index non pas au moyen de l'interface, mais directement par leur code SQL.

- ⇨ En recopiant le code produit par pgAdmin, préparer les requêtes de création et de destruction de l'index **`btree`** sur le couple **`(longitude,latitude)`**,
- ⇨ et celles de l'index **`gist`** sur la colonne **`geom`**.

10.3.3 Mesure des temps d'exécution

Pour rappel, lorsque j'ai plusieurs requêtes dans un éditeur de requêtes, en sélectionnant une requête, l'exécution ne portera que sur cette requête.

Question 20 – Exécutez les instructions suivantes et notez soigneusement les résultats qui apparaissent à l'écran.

- ⇨ Reprenez la requête sur les latitudes et les longitudes.
- ⇨ Lancez cinq fois cette requête, en notant les temps réalisés à chaque fois.
- ⇨ Ecrivez « **EXPLAIN** » en début de requête. Lancez la requête. Notez le résultat.

- ⇨ Sur les colonnes **latitude** et **longitude**, mettez un index **btree**. Relancez cinq fois la requête (sans **EXPLAIN**) puis une fois avec **EXPLAIN**. Notez les résultats.

Question 21 – Essayez le même enchaînement d'opérations avec la requête sur la colonne geom. Que se passe-t-il ?

Question 22– Quelles sont vos conclusions ?

- ⇨ Lorsque vous avez fini vos tests, supprimez tous les faux objets de la table **tdt**.
- ⇨ Supprimez les index sur les colonnes **latitude**, **longitude** et **geom**.

Attention de ne pas détruire les vrais tremblements de terre (si vous les détruisez, vous pouvez les reconstruire en faisant tourner les codes SQL que vous avez naturellement conservez depuis le début du TP).

10.4 Une erreur fréquente

Pour que le moteur PostGresSQL utilise l'index spatial, il faut que la requête s'appuie sur une fonction spatiale qui utilise les index.

La requête :

```
SELECT id FROM tdt
WHERE ST_X(geom)<21 AND ST_X(geom)>20
     AND ST_Y(geom)<11 AND ST_Y(geom)>10
```

n'utilise pas les index.

- ⇨ Cherchez à comprendre pourquoi. Demandez à l'enseignant.

Indice : la vraie question n'est pas de comprendre pourquoi cet exemple ne fonctionne pas, mais comment fonctionnent les requêtes pour lesquelles un index est activé.

11 Fonctions spatiales

11.1 Géométries valides et invalides

Un certain nombre de fonction spatiale de postGIS ne peuvent fonctionner que sur des géométries **valides**.

- ⇨ Qu'est-ce qu'une géométrie valide ? Comment teste-t-on qu'une géométrie est valide ?

⇨ Ecrivez et testez la requête qui compte le nombre de géométries invalides dans les tables **tdt** et **monde**.

```
SELECT count(*)
FROM cmt
WHERE NOT ST_IsValid(geom);
```

⇨ Quelle est l'aire maximale parmi les géométries invalides dans la table **monde** ?

⇨ Affichez dans QGIS toutes les surfaces non valides.

⇨ Est-il raisonnable de supprimer de la table **monde** les objets dont les surfaces ne sont pas valides ?

Et si on utilisait la fonction **ST_buffer()** pour rendre les géométries valides ?

⇨ lire la définition de cette fonction.

⇨ On y va prudemment !
- Créez la colonne **geom2** dans **monde**.
- Faites une requêtes de débugage : **UPDATE … SET geom2= … WHERE gid<10**.
- Lorsque la requête fonctionne, appliquez-là à toutes les géométries non-valides.
- Comptez le nombre de géométries invalides dans **geom2**.
- Affichez **geom2** dans QGIS. Contrôlez visuellement le résultat.
- Cherchez le maximum de la différence des aires entre **geom2** et **geom**.
- En quelle unité est exprimée cette différence de surface ? Donnez un majorant de cette différence, dans une unité pertinente du système métrique.
- Injectez **geom2** dans **geom** et supprimez **geom2**.

La fonction **ST_MakeValid()** est une bonne alternative. Le problème de cette fonction, c'est que vous ne maîtrisez pas les opérations qu'elle mettra en œuvre. Vous lisez dans sa documentation qu'elle peut produire un « dimensional collapse ». Savez-vous ce que c'est ? Passer par une fonction au comportement plus prévisible permet de garder une meilleure maîtrise de votre processus.
Si vous préférez passer par la fonction **ST_MakeValid()**, vous procéderez néanmoins selon le pas-à-pas ci-dessus, avec contrôle du résultat, pour garder le contrôle de vos données.

11.2 Les fonctions géométriques

Nous venons de voir la fonction **ST_Within()**, et nous avions vu plus haut la fonction **ST_GeomFromText()**. Arrêtons-nous un instant pour faire le point sur les fonctions spatiales de PostGIS.
Du grand nombre de fonctions qu'offre PostGIS, nous n'allons présenter que les principales. Pour les explorer plus avant, vous pouvez lire https://postgis.net/docs/reference.html qui en présente la liste documentée.
Dans la documentation, vous pouvez aussi naviguer grâce aux suggestions en bas de page. Elles offrent, pour chaque fonction, trois ou quatre autres fonctions dont les thèmes sont proches.

11.2.1 Les relations spatiales

Revenons sur la fonction `ST_Within()`, qui prend deux géométries en paramètre, et qui renvoie un booléen. PostGIS comporte plusieurs fonctions sur ce modèle ou sur un modèle semblable, et ce sont ces fonctions qui nous servirons pour nos conditions de jointure.

Les plus utilisées, outre `ST_Within()`, sont `ST_Intersects()`, `st_DWithin()`.

Signalons par ailleurs l'existence de ce que la documentation de PostGIS appelle curieusement des opérateurs géométriques (comme si les autres fonctions géométriques n'étaient pas des opérateurs). Ces "opérateurs géométrique" sont des fonctions très élémentaires, du type boîte englobante. Ils étaient conçus à l'origine pour être les seuls à pouvoir utiliser les index sur les géométries, mais l'utilisation des index a été étendue à bien d'autres fonctions, dont `ST_Within()`, `ST_Intersects()` et `st_DWithin()`.

11.2.2 Constructeur de géométries

Certaines fonctions renvoient des géométries. Parmi elles, signalons les constructeurs de géométries (*Geometry Constructors* dans la documentation), les éditeurs de géométries (*geometry editors*) et toutes les opérations qui modifient des géométries (*Geometry Processing*).

Au groupe des constructeurs appartiennent en particulier `ST_GeomFromText()`, et d'autres fonctions qui transcrivent une géométrie à partir d'un format externe, mais aussi `ST_MakePoint()` et tous les constructeurs de géométries à partir de géométries plus simples : par exemple la fonction `ST_LineFromMultiPoint()` construit une ligne (objet `line`) à partir d'un `MultiPoint`.

Dispersés dans les *Geometry Accesors*, *Geometry Processing* et Geometry Editors, on trouve des fonctions qui "démontent" une géométrie complexe en une ou des géométries plus simples, ou qui en extraient des morceaux. Signalons `ST_PointN()` qui renvoie le Nème point d'une ligne, `ST_Dump()` et `ST_Multi()`.

Signalons enfin les fonctions `ST_SetSRID()` et `ST_Transform()` pour la gestion du système de coordonnées.

11.2.3 Analyse spatiale

Dans les *Geometry Processing* on trouve entre autre :

- les opérateurs d'intersection, de réunion, de différence (`ST_Union`, `ST_Intersection`, `ST_SymDifference`, `ST_Difference`)
- des opérateurs construisant des objets issus de l'analyse spatiale, tel que tampons, enveloppe convexe, tesselation de Voronoï, etc. Nous avons vu plus haut, pour les tampons, la fonction `ST_Buffer()`.

Il y en a bien d'autres encore, et avoir lu la liste des fonctions, tâche fastidieuse, n'en est pas moins un bon moyen de se figurer la richesse de PostGIS.

Attention, une faute courante chez les élèves est la confusion entre les fonctions `ST_Intersection()` et `ST_Intersects()`. Ces deux fonctions ne renvoient pas du tout le même type d'objet.

⇨ Pour chacune des fonctions géométriques listées ici, vous avez naturellement consulté la documentation, pour en savoir plus. Si vous ne l'avez pas fait, faites le maintenant.

11.3 jointures sur un tampon

Nous allons utiliser maintenant quelques unes de ces fonctions pour réaliser des opérations d'analyse de nos données.
Pour commencer, nous intéressons à la détection des tremblements de terre côtiers, ceux qui se trouvent en mer, mais à une distance réduite du rivage, par exemple à moins de 30 km.

Voici ce qu'il ne faut jamais faire :

```
SELECT count(t.id)
FROM tdt AS t JOIN monde AS m
ON NOT t.terrestre AND ST_Distance( t.geom, m.geom ) < XXX
```

⇨ Quelle valeur devons-nous donner au paramètre XXX pour que la détection se fasse environ sur 30 km ?

Ne testez pas cette requête : elle est vraiment très coûteuse en temps. Si vous voulez tester le temps d'exécution de cette requête, prévoyez une pause déjeuner, voire une petite sieste.

⇨ Pourquoi est-elle coûteuse en temps ?

Pour éviter de trop dormir pendant que tourne le serveur, nous devons utiliser une autre méthode. Deux possibilités s'offrent à nous :

- soit nous appliquons une zone tampon (un buffer) de xxx° (environ 30 kilomètres) sur les terres émergées, puis nous sélectionnons les tremblements de terre qui appartiennent à ce buffer. Pour appliquer le buffer, il faut au préalable avoir regroupé toutes les terres émergées en une seule géométrie.
- soit nous utilisons la fonction `ST_DWithin()`, qui correspond exactement à notre besoin.

⇨ Cherchez la documentation de `ST_DWithin()`.

⇨ La documentation est ambiguë quant à l'unité utilisée dans le calcul. Inventez une manipulation qui fixe cette ambiguïté.

⇨ Écrivez la requête SELECT, puis testez-la.

⇨ Ajouter une colonne booléenne `cotier` dans la table `tdt` pour indiquer si ce tremblement de terre est côtier.

Nous savons maintenant d'un tremblement de terre s'il est terrestre, côtier ou de haute mer.

⇨ Affichez ces informations dans QGIS pour vérifier la validité de vos calculs.

Et on retient la leçon :

> Pour faire une jointure géométrique sur un tampon, il faut utiliser la fonction ST_DWithin, spécialement conçue pour cette tâche.

11.4 Autojointure

Notre objectif dans cette partie est de construire le contour des zones de forte sismicité. Une zone sismique est une zone où il y a beaucoup de tremblements de terre. Nous allons la calculer de la manière suivante :

- Nous allons isoler les tremblements de terre qui ont plus de N tremblements de terre autour d'eux, dans un rayons de R kilomètres.
- Autour de ces tremblements de terre, nous allons tracer le tampon sur un rayon de R2 kilomètres.

L'une des difficultés, que nous n'allons pas traiter ici, tient à la projection que nous utilisons : le calcul de la distance sur la sphère ne passe pas par la simple somme des carrés des coordonnées angulaires. Les distances que nous allons utiliser sont donc fausses, et ce d'autant plus qu'on s'élève en latitude. Nous essayerons de régler cela plus tard.

Comme nous cherchons à comparer des tremblements de terre avec eux-mêmes, la table **`tdt`** va figurer deux fois dans nos requêtes.

Détection d'un voisinage – le cas N=1

Pour détecter les tremblements de terre qui sont proches d'autres tremblements de terre, nous devons croiser la table tdt avec elle-même. La syntaxe ressemble à ceci :

```
SELECT t1.id, t1.geom, t1.magnitude
FROM tdt AS t1,
JOIN tdt AS t2 ON condition de jointure
                AND son complément indispensable
```

Cette requête est une auto-jointure. Ici, les alias de table sont obligatoires.

La condition de jointure est assez simple : elle ressemble à ce que nous avons vu pour la détection des tremblements de terre côtier, en **`ST_DWithin()`**. Mais quel est le complément indispensable ?

- ⇨ Imaginez trois tremblements de terre A, B et C. La distance entre A et B est faible. C est plus loin. Construisez le tableau cartésien de l'auto-jointure de ces trois tremblements de terre, et vérifiez comment se comporte votre condition de jointure sans le complément indispensable.
- ⇨ A quelle règle précédemment énoncée venons-nous de découvrir une exception ?

- ⇨ Terminez la requête, et testez-la

Il reste peut-être un problème ?

- ⇨ Réglez le problème.
- ⇨ Si vous ne voyez pas le problème, ou si le problème vous résiste, appelez l'enseignant.

Sur le modèle déjà exploré plus haut, nous allons ajouter une colonne **possede_voisin**, qui indiquera si le tremblement de terre est isolé ou pas.

- ⇨ Créez la colonne **possede_voisin**, de type booléen.
- ⇨ Réécrivez la requête ci-dessus pour en faire une requête de mise à jour de la colonne **possede_voisin**.

Question 23 – Recopiez cette requête de mise à jour.

Comptage des voisins – le cas N>1

Pour savoir si un tremblement de terre est vraiment isolé, ce n'est pas la présence d'un voisin qui compte, mais de plusieurs voisins : nous allons caractériser une zone sismique par la présence d'au moins 10 voisins dans un rayon de 0,4 degré.

Pour compter le nombre de voisins, nous allons maintenant utiliser la technique des sous-requêtes :

```
UPDATE tremblementdeterre AS t4
SET nbre_voisins=nbre
FROM (
  SELECT t1.id, count(*) AS nb
  FROM tremblementdeterre AS t1, tremblementdeterre AS t2
  WHERE ST_DWithin(t1.geom, t2.geom, 0,4) AND complément
  GROUP BY t1.id
) AS t3
WHERE t3.id=t4.id
```

- ⇨ Créez dans la table **tdt** la colonne qui permet de stocker le nombre de voisins de chaque tremblement de terre
- ⇨ La requête UPDATE est inachevée. Modifiez-la pour la rendre exécutable.

Question 24 – En vous aidant de la documentation, analysez les différents termes de la requête précédente pour en donner la signification. Que représente t4 ?

- ⇨ Faites tourner la requête. Visualisez son résultat sous QGIS.

- ⇨ En vous aidant de la documentation, écrivez la requête qui construit la zone sismique sous la forme d'une zone tampon de 0,8 degré autour des tremblements de terre possédant au moins 3 voisins.

- ⇨ Visualisez la zone sismique sous QGIS.

- ⇨ construisez autour des terres émergées un tampon d'un demi degré.

On note que la condition **ST_DWithin(t1.geom, t2.geom, x)** est symétrique. Elle est donc exécutée deux fois sur des valeurs symétriques, pour un résultat semblable. On aimerait bien diviser par deux la charge de calcul, mais je ne vois pas de méthodes simples pour obtenir cette optimisation sans complexifier énormément le code SQL.

11.5 Plus proche voisin

La question du plus proche voisin est une question assez différente de la recherche de la présence d'un objet dans un voisinage de rayon donné. C'est une fonctionnalité à laquelle je ne vois aucune utilité dans le cas des tremblements de terre, mais cela doit-il nous arrêter ? La recherche du plus proche voisin porte des enjeux forts dans le cas d'une recherche d'itinéraire, pour assurer le raccordement des points d'intérêts (points de départ et points d'arrivée) aux nœuds du graphe.

- ⇨ Écrivez une requête simple qui pour chaque tremblement de terre détecte son plus proche voisin.
- ⇨ Pourquoi cette requête n'est-elle pas opérationnelle ?
- ⇨ Inventez un ensemble de requêtes qui font le travail. Soyez pragmatique.

11.6 Type Geography

Nous avons fait tous nos calculs dans la projection plate carrée, en traitant les distances angulaires comme des vraies distances. Nous savons qu'une telle approximation n'est pas acceptable sur nos traitements, puisque nos données montent jusqu'à de hautes latitudes, et que nous avons fait des zones tampons et des calculs de distance.

Par ailleurs, nos traitements ont totalement négligé un autre aspect de la rotondité de la terre : la projection plate carré provoque une déchirure de l'espace géographique aux pôles et le long du méridien lng=180°.

Pour ce type d'usage, PostGIS a introduit un type **`geography`**. Ce type permet de prendre en compte les deux effets décrits ci-dessus.

- ⇨ A partir des colonnes **`geom`**, construisez des colonnes **`geog`**, de type **`geography`**.
- ⇨ Refaire les traitements précédents avec les colonnes **`geog`**. Vérifiez dans la documentation que les fonctions sont disponibles pour le type **`geography`**.

- ⇨ Pour investiguer l'utilité du type **`geography`**, comptez le nombre de tremblements de terre qui sont à moins de 30 kilomètres mais à plus de 0.3 degrés de la cote.

Pour ce travail sur le type **`geography`**, vous pourrez vous appuyer sur cette référence :
https://postgis.net/workshops/postgis-intro/geography.html

12 Intégration de données avec PHP

Les deux jeux de données que nous avons intégrés dans notre base étaient structurées chacune sous un format assez classique dont l'intégration s'est faite au moyen d'outils standards. Nous pouvons rencontrer des jeux de données plus exotiques, dont l'intégration dans SQL pourrait poser plus de difficulté.

C'est le cas, en réalité, du jeu de données sur les tremblements de terre, dont la version CSV a été préparée pour ce TP. Dans le monde réel, les données se trouvaient présentées sous un format à longueur fixe.

C'est ce format que je vous propose maintenant de traduire vers SQL. Pour cela, vous programmerez en PHP un lecteur qui « épluchera » le fichier ligne à ligne, reconnaîtra les différents champs et les intégrera dans la base.

Ce travail a déjà été fait pour une base mySQL.

Les fonctions PHP pour manipuler PostGreSQL et mySQL sont très proches. Généralement, une fonction pour mySQL, reconnaissable à son préfixe **`mysql_`**, est l'exacte parallèle d'une fonction **`pg_`**, mais de petites différences viennent parfois perturber la traduction. Ainsi, pour se connecter à une base, on utilise **`pg_connect`** pour PostGreSQL et **`mysql_connect`** pour mySQL, mais les paramètres de ces deux fonctions ne sont pas les mêmes.

Dans tous les cas, il vaut mieux se référer à la documentation en ligne (par exemple : http://fr.php.net/pgsql)

⇨ Modifiez et complétez le fichier **`lectureFichierCMT.php`** pour qu'il fasse l'insertion dans la table **`tdt`**.
Il faut penser à remplacer tous les préfixes **`mysql_`** par les préfixes **`pg_`**. Dans le code, vous devez remplacer les « XXX » par les instructions correctes.

⇨ créez le fichier **`connexion.inc.php`**
Vous irez chercher dans la documentation de postGreSQL la syntaxe de la fonction **`pg_connect()`**.

⇨ Débuguez le code jusqu'à ce qu'il fonctionne.
On vérifie que le code fonctionne en affichant le contenu de la table dans pgAdmin.

⇨ Lorsque le code fonctionne :
- supprimez la dernière ligne (**`$clef>100`**),
- avec pgAdmin, videz la base des données que vous avez déjà inscrites (cherchez la syntaxe de la fonction **`DELETE`** dans la documentation PostGreSQL)
- relancez l'intégration des données dans la base

Elapsed Time

Attention, il arrive que le programme s'arrête avant d'avoir lu toutes les données. En effet, pour des raisons de sécurité, le serveur arrête les scripts qui durent trop longtemps. Par défaut, la durée maximale d'un script est de 30 secondes.

Le serveur affiche alors une erreur « elapsed time », ou un message équivalent. Pour résoudre ce problème éventuel, vous devez allonger la durée maximale accordée à chaque script.

Même si votre script a tourné en moins de 30 secondes, je vous demande de faire l'exercice qui consiste à trouver comment régler ce problème.

⇨ trouvez comment allonger la durée maximale du script PHP sur votre serveur local (la solution est sûrement sur internet).

Question 25– Décrivez comment vous allongez la durée maximale du script PHP sur votre serveur local. Donnez la requête google (ou autre) qui vous a conduit au résultat.

13 Webographie

Une partie de cette webographie est maintenant obsolète. A la date du 31 octobre 2021, la documentation francophone de Postgis est parcellaire ou perdue (voir par exemple les erreurs de serveur de http://www.postgis.fr/wiki/Documentations).
On se reportera massivement sur la documentation anglophone, par exemple sur le site de l'OSGEO (ou https://postgis.net/documentation/) ou mieux encore, on utilisera un moteur de recherche générique.

13.1 PostGreSQL

http://docs.postgresql.fr/8.4/
(le point d'entrée)
http://docs.postgresql.fr/8.4/sql.html
(les chapitres sur le langage SQL)
http://docs.postgresql.fr/8.4/server-programming.html
(les chapitres sur la programmation serveur : PL/pgSQL, triggers, etc.)

13.2 PostGIS

La documentation officielle, en anglais :
* https://postgis.net/documentation/
(le point d'entrée)
* https://postgis.net/docs/manual-3.1/postgis_usage.html
(le guide utilisateur de PostGIS)

La traduction de la documentation en français :
* https://postgis.net/docs/manual-dev/postgis-fr.html
(le point d'entrée)
Une traduction est toujours un peu en retard. Celle de PostGIS n'est vraiment pas à jour. Au 31 octobre 2021, la partie « 4.3.5. Ensuring OpenGIS compliancy of geometries » n'est pas traduite. On la trouvera en anglais ici :
* http://postgis.refractions.net/docs/using_postgis_dbmanagement.html#OGC_Validity

Quelques bonnes pratiques :
http://www.geoinformations.developpement-durable.gouv.fr/fichier/pdf/m06_bonnes_pratiques_papier_cle2a34b3.pdf?arg=177833935&cle=3ab9857d615c7fd541765de760b2b6708f7cc945&file=pdf%2Fm06_bonnes_pratiques_papier_cle2a34b3.pdf

Le cours de David Techer :

* http://www.davidgis.fr/documentation/win32/html/index.html
  (le point d'entrée)
* http://www.davidgis.fr/documentation/win32/html/ch05.html
  (le tutorial)
* http://www.davidgis.fr/documentation/win32/html/ch06.html
  (des exemples)

14 A rendre

Question 0 – Identification :

Nom : Prénom :

Question 1– Liste des colonnes avec leur type :

	timestamp without time zone
latitude	Real
longitude	

Question 2 – Instruction SQL fixant la valeur par défaut de la colonne id :

Question 3 – Une requête utilisant la fonction count() :

Requête : Résultat :

Question 4 – Deux requêtes utilisant les clauses ORDER BY et LIMIT :

Écrivez la requête qui trouve la magnitude du tremblement de terre de plus forte magnitude. Résultat :

Écrivez la requête qui trouve la magnitude du tremblement de terre le plus profond.

Résultat :

Question 5 - Requête pour afficher la distribution des valeurs d'un attribut :

La requête :

Les premières ligne du résultat :

Question 6 - Comment interprétez-vous la répartition des valeurs de magnitude ?

Question 7 – La vue qui donne les 300 tremblements de terre de plus forte magnitude :

Question 8 - la requête qui détermine la magnitude du plus faible tremblement de terre de cette vue :

Question 9 – compatibilité ascendante :

Question 10 – Instruction créant la colonne geom :

Question 11 – UPDATE sur la colonne geom :

Question 12 - donnez tous les types OGC disponibles dans PostGIS, et, s'il en existe, un exemple de type OGC absent de PostGIS.

Les géométries OGC disponibles dans PostGIS :

un exemple de géométrie OGC absente de PostGIS :

Question 13 – Quel est le nom de l'objet créé par l'option -I ?

Question 14 - documentez le bug, et donnez le nom de la fonction PostGIS qui permet de le corriger

Question 15 – Jointure dans un SELECT – la requête modifiée :

Question 16 – nombre de tremblements de terre trouvés

Pour la requête avec WITHIN :

Pour la requête avec NOT WITHIN :

Question 17 – Combien d'objets maintenant dans la table tdt ?

Question 18 – A quoi sert l'instruction EXPLAIN ?

Question 19 – Deux requêtes SELECT … WHERE :

- sur `latitude` et `longitude` : - sur `geom` :

Question 20 – Comparaison des index pour une requête sur les colonnes `latitude` et `longitude` :

	Requête sur les colonnes `latitude` et `longitude`					
	EXPLAIN	Temps d'exécution (en ms)				
sans index						
		moyenne :				
avec **btree**						
		moyenne :				
Avec **btree** et **gist**						
		moyenne :				
avec gist (sans **btree**)						
		moyenne :				

Question 21 - Même chose pour la colonne `geom` :

	Requête sur les colonnes `latitude` et `longitude`					
	EXPLAIN	Temps d'exécution (en ms)				
sans index						
		moyenne :				
avec **btree**						
		moyenne :				
Avec **btree** et **gist**						
		moyenne :				
avec gist (sans **btree**)						
		moyenne :				

Question 22 – Quelles sont vos conclusions ?

Question 23 – transformation en requête de mise à jour :

Question 24 – analyse détaillée de la requête

Question 25 - Durée maximale du script PHP :

Comment allonger la durée maximale ?

Requête Google pour arriver à cette réponse :