Introduction to PostGIS

Version 1.0

Mark Leslie, Paul Ramsey

May 06, 2024

Table des matières

1 Bienvenue **1**
1.1 Conventions d'écriture 1

2 Partie 1 : Introduction **3**
2.1 Qu'est-ce qu'une base de données spatiales ? 3
2.2 Qu'est-ce que PostGIS ? 7

3 Partie 2 : Installation **11**

4 Partie 3 : Créer une base de données spatiales **17**
4.1 Le Dashboard et PgAdmin 17
4.2 Créer une base de données 19
4.3 Liste des fonctions 23

5 Partie 4 : Charger des données spatiales **25**
5.1 Shapefile ? Qu'est-ce que c'est ? 27
5.2 SRID 26918 ? Qu'est que c'est ? 28
5.3 Les choses à essayer : rendre spatiale une base de données existante 30
5.4 Les choses à essayer : visualiser des données avec uDig 30

6 Partie 5 : A propos de nos données **31**
6.1 nyc_census_blocks 31
6.2 nyc_neighborhoods 33
6.3 nyc_streets 34
6.4 nyc_subway_stations 34
6.5 nyc_census_sociodata 35

7 Partie 6 : Requêtes SQL simples **37**
7.1 Requête de type SELECT 38
7.2 Liste de fonctions 40

8 Partie 7 : Exercices simples de SQL **41**
8.1 Liste des fonctions 42

9 Partie 8 : Les géometries **43**
9.1 Introduction . . . 43
9.2 Les tables de métadonnées . . . 44
9.3 Réprésenter des objets du monde réel . . . 46
9.4 Entré / Sortie des géométries . . . 52
9.5 Liste des fonctions . . . 55

10 Partie 9 : Exercices sur les géométries **57**
10.1 Exercices . . . 58

11 Partie 10 : Les relations spatiales **61**
11.1 ST_Equals . . . 61
11.2 ST_Intersects, ST_Disjoint, ST_Crosses et ST_Overlaps . . . 63
11.3 ST_Touches . . . 65
11.4 ST_Within et ST_Contains . . . 65
11.5 ST_Distance et ST_DWithin . . . 69
11.6 Liste des fonctions . . . 70

12 Partie 11 : Exercices sur les relations spatiales **73**
12.1 Exercices . . . 73

13 Partie 12 : Les jointures spatiales **77**
13.1 Jointure et regroupement . . . 77
13.2 Jointures avancées . . . 80
13.3 Liste de fonctions . . . 81

14 Partie 13 : Exercices sur jointures spatiales **83**
14.1 Exercices . . . 83

15 Partie 14 : L'indexation spatiale **87**
15.1 Comment les index spatiaux fonctionnent . . . 88
15.2 Requête avec seulement des index . . . 89
15.3 Analyse . . . 90
15.4 Néttoyage . . . 91
15.5 Liste des fonctions . . . 91

16 Partie 15 : Projections des données **93**
16.1 Comparaison de données . . . 94
16.2 Transformer les données . . . 94
16.3 Liste des fonctions . . . 95

17 Partie 16 : Exercices sur les projections **97**
17.1 Exercices . . . 97

18 Partie 17 : Coordonnées géographiques **99**
18.1 Utiliser le type 'Geography' . . . 102
18.2 Création d'une table stockant des géographies . . . 103
18.3 Conversion de type . . . 103
18.4 Pourquoi (ne pas) utiliser les géographies . . . 104
18.5 Liste des fonctions . . . 104

19 Partie 18 : Fonctions de construction de géométries **107**
19.1 ST_Centroid / ST_PointOnSurface . . . 107

19.2 ST_Buffer . . . 108
19.3 ST_Intersection . . . 111
19.4 ST_Union . . . 111
19.5 Liste des fonctions . . . 114

20 Partie 19 : Plus de jointures spatiales **115**
20.1 Création de la table de traçage des recensements . . . 115
20.2 Polygones/Jointures de polygones . . . 117
20.3 Jointures utilisant un large rayon de distance . . . 118

21 Partie 20 : Validité **121**
21.1 Qu'est-ce que la validité ? . . . 121
21.2 Détecter la validité . . . 122
21.3 Réparer les invalides . . . 123

22 Partie 21 : Paramétrer PostgreSQL pour le spatial **125**
22.1 shared_buffers . . . 128
22.2 work_mem . . . 129
22.3 maintenance_work_mem . . . 129
22.4 wal_buffers . . . 130
22.5 checkpoint_segments . . . 131
22.6 random_page_cost . . . 132
22.7 seq_page_cost . . . 133
22.8 Recharger la configuration . . . 134

23 Partie 22 : Égalité **135**
23.1 Égalité . . . 135

24 Annexes A : Fonctions PostGIS **141**
24.1 Constructeurs . . . 141
24.2 Sorties . . . 141
24.3 Mesures . . . 141
24.4 Relations . . . 142

25 Annexes B : Glossaire **143**

26 Annexes C : License **145**

Index **147**

CHAPITRE 1

Bienvenue

1.1 Conventions d'écriture

Cette section présente les différentes conventions d'écriture qui seront utilisées dans ce document afin d'en faciliter la lecture.

1.1.1 Indications

Les indications pour vous, lecteurs de ce document, seront notées en **gras**.

Par exemple :

> Cliquez sur **Suivant** pour continuer.

1.1.2 Code

Les exemples de requêtes SQL seront affichés de la manière suivante :

```
SELECT postgis_full_version();
```

Cet exemple peut être saisi dans la fenêtre de requêtage ou depuis l'interface en ligne de commande.

1.1.3 Notes

Les notes sont utilisées pour fournir une information utile mais non critique pour la compréhension globale du sujet traité.

Note : Si vous n'avez pas mangé une pomme aujourd'hui, le docteur devrait se mettre en route.

1.1.4 Fonctions

Lorsque les noms de fonctions sont contenus dans une phrase, ils sont affichés en **gras**.

Par exemple :

ST_Touches(geometry A, geometry B) retourne vrai si un des contours de géométrie touche l'autre contour de géométrie

1.1.5 Fichiers, Tables et nom de colonne

Les noms de fichiers, les chemins, le noms de tables et les noms de colonnes seront affichés comme suit

Select the `name` column in the `nyc_streets` table.

1.1.6 Menus et formulaires

Les menus et les éléments de formulaire comme les champs ou les boîtes à cocher ainsi que les autre objets sont affichés en *italique*.

Par exemple :

Cliquez sur *Fichier* > *Nouveau*. Cochez la case qui contient *Confirmer*.

1.1.7 Organisation

Les différentes sections de ce document permettent d'évoluer progressivement. Chaque section suppose que vous ayez terminé et compris les sections précédentes.

Certaines sections fournissent des exemples fonctionnels ainsi que des exercices. Dans certains cas, il y a aussi des sections "Les choses à essayer" pour les curieux. Ces tâches contiennent des problèmes plus complexes que dans les exercices.

CHAPITRE 2

Partie 1 : Introduction

2.1 Qu'est-ce qu'une base de données spatiales ?

PostGIS est une base de données spatiale. Oracle Spatial et SQL Server 2008 sont aussi des bases de données spatiales. Mais qu'est-ce que cela signifie ? Qu'est-ce qui différencie un serveur de base de données spatiales d'un serveur de base de données non spatiale ?

La réponse courte, est ...

Les bases de données spatiales permettent le stockage et la manipulation des objets spatiaux comme les autres objets de la base de données.

Ce qui suit présente brièvement l'évolution des bases de données spatiales, puis les liens entre les données spatiales et la base de données (types de données, index et fonctions).

1. **Types de données spatiales** fait référence aux géométries de type point, ligne et polygone ;
2. L'**indexation spatiale** est utilisée pour améliorer les performances d'exécution des opérations spatiales ;
3. Les **fonctions spatiales**, au sens *SQL*, sont utilisées pour accéder à des propriétés ou à des relations spatiales.

Utilisés de manière combinée, les types de données spatiales, les index et les fonctions fournissent une structure flexible pour optimiser les performances et les analyses.

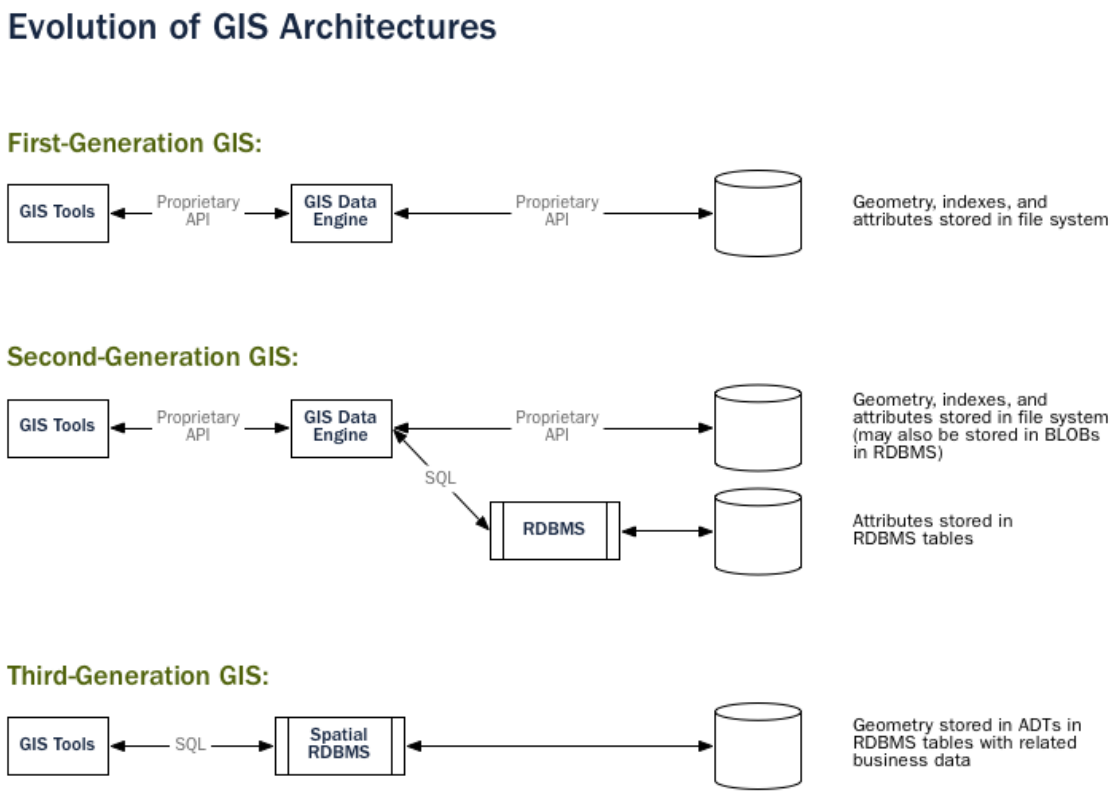
2.1.1 Au commencement

Dans les premières implémentations *SIG*, toutes les données spatiales étaient stockées sous la forme de fichiers plats et certaines applications *SIG* spécifiques étaient nécessaires pour les interpréter et les manipuler. Ces outils de gestion de première génération avaient été conçus pour répondre aux besoins des utilisateurs pour lesquels toutes les données étaient localisées au sein de leur agence. Ces outils propriétaires étaient des systèmes specifiquement créés pour gérer les données spatiales.

La seconde génération des systèmes de gestion de données spatiales stockait certaines données dans une base de données relationelle (habituellement les "attributs" ou autres parties non spatiales) mais ne fournissaient pas encore la flexibilité offerte par une intégration complète des données spatiales.

Effectivement, les bases de données spatiales sont nées lorsque les gens ont commencé à considérer les objet spatiaux comme les autres objets d'une base de données .

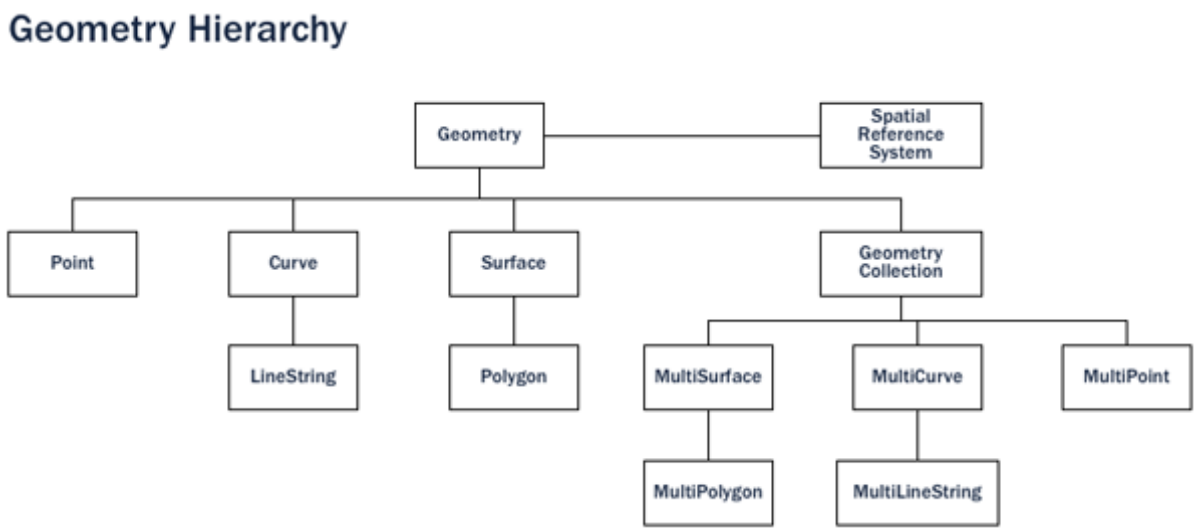
Les bases de données spatiales intègrent les données spatiales sous forme d'objets de la base de données relationnelle. Le changement opéré passe d'une vision centrée sur le SIG à une vision centrée sur les bases de données.



Note : Un système de gestion de base de données peut être utilisé dans d'autres cadres que celui des SIG. Les bases de données spatiales sont utilisées dans divers domaines : l'anatomie humaine, les circuits intégrés de grandes envergures, les structures moléculaires, les champs electro-magnétiques et bien d'autres encore.

2.1.2 Les types de données spatiales

Une base de données classique propose par exemple les types chaînes de caractères et date. Une base de données spatiales ajoute les types de données (spatiales) pour représenter les **entités géographiques**. Ces types de données spatiales permettent d'accéder à des propriétés de l'entité géographique comme ses contours ou sa dimension. Pour bien des aspects, les types de données spatiales peuvent être vus simplement comme des formes.



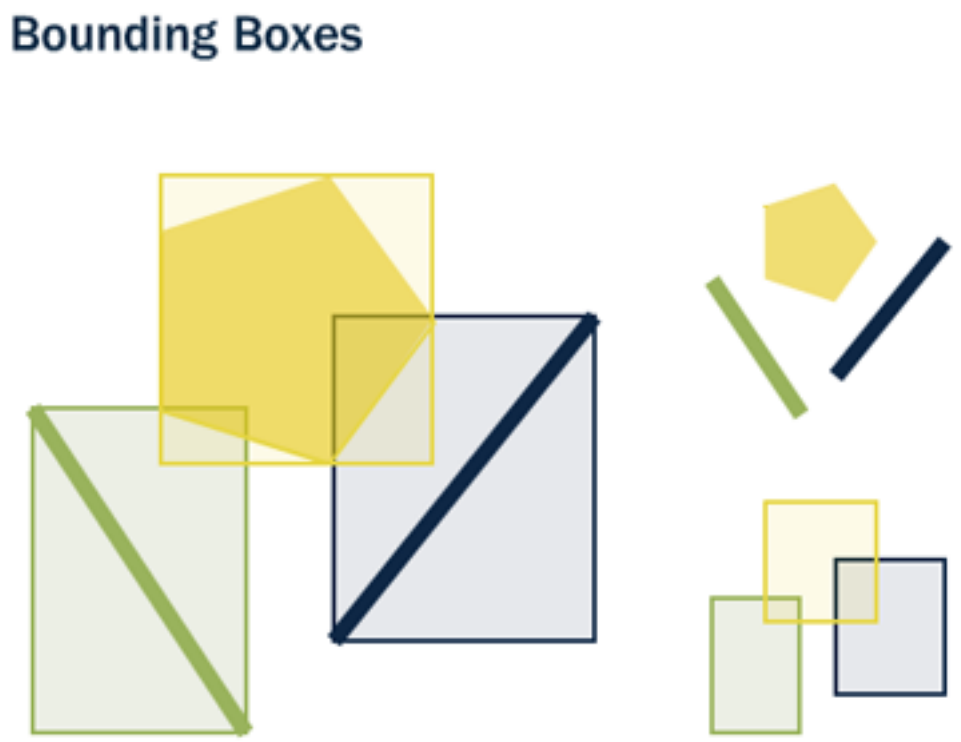
Les types de données spatiales sont organisés par une hiérarchie de type. Chaque sous-type hérite de la structure (les attributs) et du comportement (les méthodes et fonctions) de son type supérieur dans la hierarchie.

2.1.3 Index spatiaux et étendue

Une base de données ordinaire fournit des "méthodes d'accès" – connues sous le nom d'**index** – pour permettre un accès efficace et non séquentiel à un sous ensemble de données. L'indexation des types non géographiques (nombre, chaînes de caractères, dates) est habituellement faite à l'aide des index de type arbres binaires. Un arbre binaire est un partitionnement des données utilisant l'ordre naturel pour stocker les données hiérarchiquement.

L'ordre naturel des nombres, des chaînes de caractères et des dates est assez simple à déterminer – chaque valeur est inférieure, plus grande ou égale à toutes les autres valeurs. Mais, étant donné que les polygones peuvent se chevaucher, peuvent être contenus dans un autre et sont représentés par un tableau en deux dimensions (ou plus), un arbre binaire ne convient pas pour indexer les valeurs. Les vraies bases de données spatiales fournissent un "index spatial" qui répond plutôt à la question : "quel objet se trouve dans une étendue spécifique ?"

Une **étendue** correspond au rectangle de plus petite taille capable de contenir un objet géographique.



Les étendues sont utilisées car répondre à la question : "est-ce que A se trouve à l'intérieur de B ? " est une opération coûteuse pour les polygones mais rapide dans le cas ou ce sont des rectangles. Même des polygones et des lignes complexes peuvent être représentés par une simple étendue.

Les index spatiaux doivent réaliser leur ordonnancement rapidement afin d'être utiles. Donc au lieu de fournir des résultats exacts, comme le font les arbres binaires, les index spatiaux fournissent des résultats approximatifs. La question "quelles lignes sont à l'intérieur de ce polygone" sera interprétée par un index spatial comme : "quelles lignes ont une étendue qui est contenue dans l'étendue de ce polygone ?"

Les incréments spatiaux réels mis en application par diverses bases de données varient considérablement. Les index spatiaux actuellement utilisés par les différents systèmes de gestion de bases de données varient aussi considérablement. L'implémentation la plus commune est l'arbre R (utilisé dans PostGIS), mais il existe aussi des implémentations de type Quadtrees, et des index basés sur une grille.

2.1.4 Les fonctions spatiales

Pour manipuler les données lors d'une requête, une base de données classique fournit des **fonctions** comme la concaténation de chaînes de caractères, le calcul de la clef md5 d'une chaîne, la réalisation d'opérations mathématiques sur les nombres ou l'extraction d'informations spécifiques sur une date. Une base de données spatiales fournit un ensemble complet de fonctions pour analyser les composants géographiques, déterminer les relations spatiales et manipuler les objets géographiques. Ces fonctions spatiales sont utilisées comme des pièces de Lego pour de nombreux projets SIG.

La majorité des fonctions spatiales peuvent être regroupées dans l'une des cinq catégories suivantes :

1. **Conversion** : fonctions qui *convertissent* les données géographiques dans un format externe.
2. **Gestion** : fonctions qui permettent de *gérer* les informations relatives aux tables spatiales et l'administration de PostGIS.
3. **Récupération** : fonctions qui permettent de *récupérer* les propriétés et les mesures d'une géométrie.

4. **Comparaison** : fonctions qui permettent de *comparer* deux géométries en respectant leurs relations spatiales.
5. **Contruction** : fonctions qui permettent de *construire* de nouvelles géométries à partir d'autres.

La liste des fonctions possibles est très vaste, mais un ensemble commun à l'ensemble des implémentations est défini par la spécification term :*OGC SFSQL*. Cet ensemble commun (avec d'autres fonctions supplémentaires) est implémenté dans PostGIS.

2.2 Qu'est-ce que PostGIS ?

PostGIS confère au système de gestion de base de données PostgreSQL le statut de base de données spatiales en ajoutant les trois supports suivants : les types de données spatiales, les index et les fonctions. Étant donné qu'il est basé sur PostgreSQL, PostGIS bénéficie automatiquement des capacités orientées "entreprise" ainsi que le respect des standards de cette implémentation.

2.2.1 Mais qu'est-ce que PostgreSQL ?

PostgreSQL est un puissant système de gestion de données relationnel à objets (SGBDRO). Il a été publié sous la licence de style BSD et est donc un logiciel libre. Comme avec beaucoup de logiciels libres, PostgreSQL n'est pas contrôlé par une société unique mais par une communauté de développeurs et de sociétés qui le développe.

PostgreSQL a été conçu depuis le début en conservant à l'esprit qu'il serait potentiellement nécessaire de l'étendre à l'aide d'extensions particulières – la possibilité d'ajouter de nouveaux types, des nouvelles fonctions et des méthodes d'accès à chaud. Grâce à cela, une extension de PostgreSQL peut être développée par une équipe de développement indépendante, bien que le lien soit très fortement lié au coeur de la base de données PostgreSQL.

Pourquoi choisir PostgreSQL ?

Une question que se posent souvent les gens déja familiarisés avec les bases de données libres est : "Pourquoi PostGIS n'a pas été basé sur MySQL ?"

PostgreSQL a :
- prouvé sa fiabilité et son respect de l'intégrité des données (propriétés ACID)
- un support soigneux des standard SQL (respecte la norme SQL92)
- un support pour le développement d'extensions et de nouvelles fonctions
- un modèle de développement communautaire
- pas de limite sur la taille des colonne (les tuples peuvent être "TOAST"és) pour supporter des objets géographiques
- un structure d'index générique (GiST) permettant l'indexation à l'aide d'arbres R
- une facilité d'ajout de fonctions personalisées

Tout ceci combiné, PostgreSQL permet un cheminement simple du développement nécessaire à l'ajout des types spatiaux. Dans le monde propriétaire, seul Illustra (maintenant Informix Universal Server) permet une extension aussi simple. Ceci n'est pas une coïncidence, Illustra est une version propriétaire modifiée du code original de PostgreSQL publié dans les années 1980.

Puisque le cheminement du développement nécessaire à l'ajout de types à PostgreSQL est direct, il semblait naturel de commencer par là. Lorsque MySQL a publié des types de données spatiaux de base dans sa version 4.1, l'équipe de PostGIS a jeté un coup d'oeil dans leur code source et cela a confirmé le choix initial d'utiliser PostgreSQL. Puisque les objets géographiques de MySQL doivent être considérés

comme un cas particulier de chaînes de caractères, le code de MySQL a été diffus dans l'intégralité du code de base. Le développement de PostGIS version 0.1 a pris un mois. Réaliser un projet "MyGIS" 0.1 aurait pris beaucoup plus de temps, c'est sans doute pourquoi il n'a jamais vu le jour.

2.2.2 Pourquoi pas des fichiers Shapefile ?

Les fichiers shapefile (et les autres formats) ont été la manière standard de stocker et d'interagir avec les données spatiales depuis l'origine des SIG. Néanmoins, ces fichiers "plats" ont les inconvénients suivants :

- **Les fichier au formats SIG requièrent un logiciel spécifique pour les lire et les écrire.** Le langage SQL est une abstraction de l'accès aléatoire aux données et à leur analyse. Sans cette abstraction, vous devrez développer l'accès et l'analyse par vos propre moyens.
- **L'accès concurrent aux données peut parfois entraîner un stockage de données corrompues.** Alors qu'il est possible d'écrire du code supplémentaire afin de garantir la cohérence des données, une fois ce problème solutionné et celui de la performance associée, vous aurez re-écrit la partie la plus importante d'un système de base de données. Pourquoi ne pas simplement utiliser une base de données standard dans ce cas ?
- **Les questions compliquées nécessitent des logiciels compliqués pour y répondre.** Les question intéressantes et compliquées (jointures spatiales, aggrégations, etc) qui sont exprimables en une ligne de SQL grâce à la base de données, nécessitent une centaine de lignes de code spécifiques pour y répondre dans le cas de fichiers.

La plupart des utilisateurs de PostGIS ont mis en place des systèmes où diverses applications sont susceptibles d'accéder aux données, et donc d'avoir les méthodes d'accès SQL standard, qui simplifient le déploiement et le développement. Certains utilisateurs travaillent avec de grands jeux de données sous forme de fichiers, qui peuvent être segmentés en plusieurs fichiers, mais dans une base de données ces données peuvent être stockées dans une seule grande table.

En résumé, la combinaison du support de l'accès concurrent, des requêtes complexes spécifiques et de la performance sur de grands jeux de données différencient les bases de données spatiales des systèmes utilisant des fichiers.

2.2.3 Un bref historique de PostGIS

En mai 2001, la société Refractions Research publie la première version de PostGIS. PostGIS 0.1 fournissait les objets, les index et des fonctions utiles. Le résultat était une base de données permettant le stockage et l'accès mais pas encore l'analyse.

Comme le nombre de fonctions augmentait, le besoin d'un principe d'organisation devint clair. La spécification "Simple Features for SQL" (*SFSQL*) publiée par l'Open Geospatial Consortium fournit une telle structure avec des indications pour le nommage des fonctions et les pré-requis.

Avec le support dans PostGIS de simples fonctions d'analyses et de jointures spatiales, Mapserver devint la première application externe permettant de visualiser les données de la base de données.

Au cours de ces dernières années, le nombre de fonctions fournies par PostGIS grandissait, mais leur puissance restait limitée. La plupart des fonctions intéressantes (ex : ST_Intersects(), ST_Buffer(), ST_Union()) étaient difficiles à implémenter. Les écrire en repartant du début promettait des années de travail.

Heureusement un second projet, nommé "Geometry Engine, Open Source" ou GEOS vit le jour. Cette librairie fournit l'ensemble des algorithmes nécessaires à l'implémentation de la spécification *SFSQL* . En se liant à GEOS, PostGIS fournit alors le support complet de la *SFSQL* depuis la version 0.8.

Alors que les capacités de PostGIS grandissaient, un autre problème fit surface : la représentation utilisée pour stocker les géométries n'était pas assez efficace. Pour de petits objets comme les points ou de courtes lignes, les métadonnées dans la représentation occupaient plus de 300% supplémentaires. Pour des raisons de performances, il fut nécessaire de faire faire un régime à la représentation. En réduisant l'entête des métadonnées et les dimensions requises, l'espace supplémentaire fut réduit drastiquement. Dans PostGIS 1.0, cette nouvelle représentation plus rapide et plus légère devint la représentation par défaut.

Les mises à jour récentes de PostGIS ont permis d'étendre la compatibilité avec les standards, d'ajouter les géométries courbes et les signatures de fonctions spécifiées dans la norme ISO *SQL/MM*. Dans un soucis de performance, PostGIS 1.4 a aussi augmenté considérablement la rapidité d'exécution des fonctions de tests sur les géométries.

2.2.4 Qui utilise PostGIS ?

Pour une liste complète des cas d'utilisation, consultez la page web : Cas d'utilisations de PostGIS (en anglais).

Institut Géographique National, France

L'IGN utilise PostGIS pour stocker des cartes topographiques de grande résolution de la France : la "BDUni". La BDUni a plus de 100 millions d'entités, et est maintenue par une équipe de 100 personnes qui vérifie les observations et ajoute quotidiennement de nouvelles données à la base. L'installation de l'IGN utilise le système transactionnel de la base de données pour assurer la consistance durant les phases de mises à jour et utilise un serveur de warm-standby par transfert de journaux afin de conserver un état cohérent en cas de défaillance du système.

GlobeXplorer

GlobeXplorer est un service web fournissant un accès en ligne à une imagerie satellite et photos aériennes de plusieurs petabytes. GlobeXplorer utilise PostGIS pour gérer les métadonnées associées avec le catalogue d'images. Les requêtes pour accéder aux images recherchent d'abord dans le catalogue PostGIS pour récupérer la localisation des images demandées, puis récupèrent ces images et les retournent au client. Lors du proccessus de mise en place de leur système, GlobeXplorer a essayé d'autres systèmes de base de données spatiales mais a conservé PostGIS à cause de la combinaison du prix et de la performance qu'il offre.

2.2.5 Quest-ce qu'une application qui supporte PostGIS ?

PostGIS est devenu une base de données spatiale communément utilisée, et le nombre d'applications tierces qui supportent le stockage ou la récupération des données n'a cessé d'augmenter. Les application qui supportent PostGIS contiennent à la fois des applications libres et des application propriétaires tournant sur un serveur ou localement depuis votre bureau.

La table suivante propose une liste des logiciels qui tirent profit de PostGIS :

Libre/Gratuit	Fermé/Propriétaire

– Chargement/Extraction
  – Shp2Pgsql
  – ogr2ogr
  – Dxf2PostGIS
– Basé sur le web
  – Mapserver
  – GeoServer (Java-based WFS / WMS -server)
  – SharpMap SDK - for ASP.NET 2.0
  – MapGuide Open Source (using FDO)
– Logiciels bureautiques
  – uDig
  – QGIS
  – mezoGIS
  – OpenJUMP
  – OpenEV
  – SharpMap SDK for Microsoft.NET 2.0
  – ZigGIS for ArcGIS/ArcObjects.NET
  – GvSIG
  – GRASS
	– Chargement/Extraction
	  – Safe FME Desktop Translator/Converter
	– Basé sur le web
	  – Ionic Red Spider (now ERDAS)
	  – Cadcorp GeognoSIS
	  – Iwan Mapserver
	  – MapDotNet Server
	  – MapGuide Enterprise (using FDO)
	  – ESRI ArcGIS Server 9.3+
	– Logiciels bureautiques
	  – Cadcorp SIS
	  – Microimages TNTmips GIS
	  – ESRI ArcGIS 9.3+
	  – Manifold
	  – GeoConcept
	  – MapInfo (v10)
	  – AutoCAD Map 3D (using FDO)

CHAPITRE 3

Partie 2 : Installation

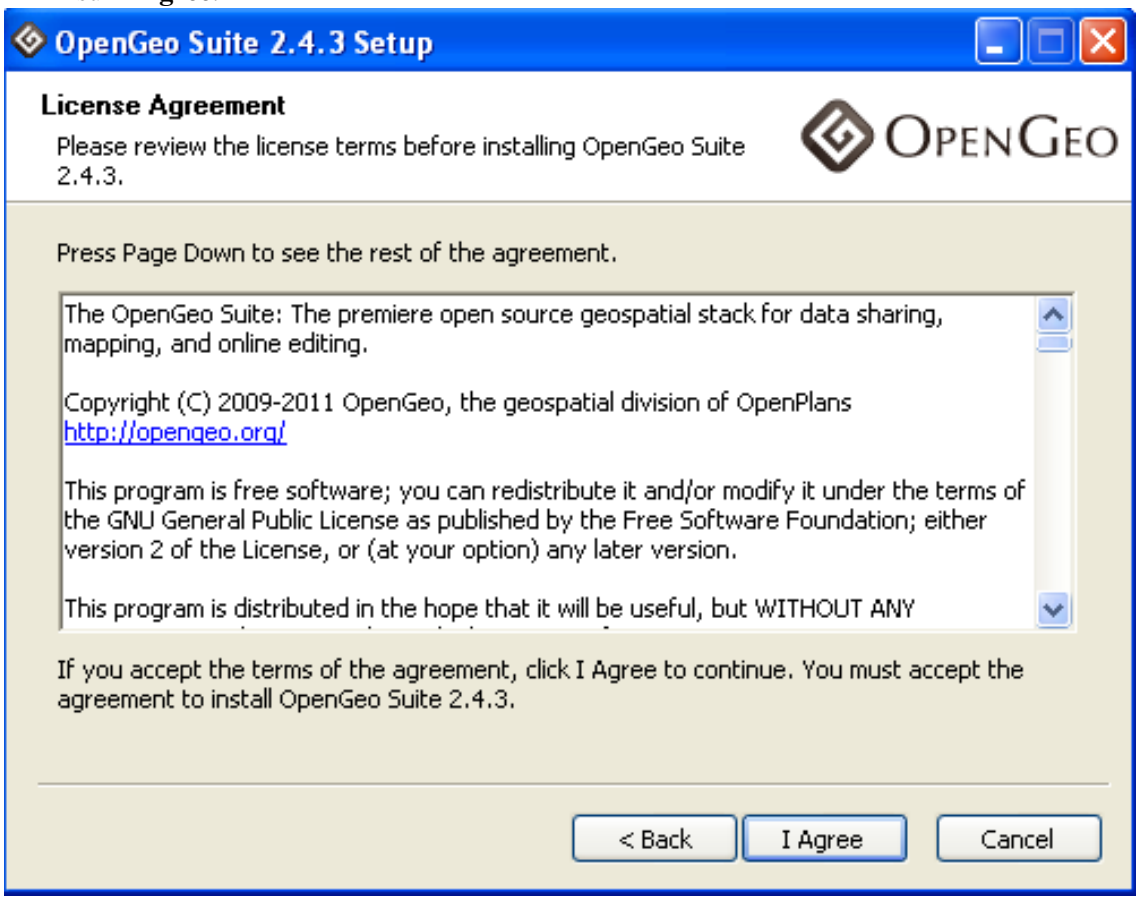
Nous utiliserons OpenGeo Suite comme application d'installation, car celle-ci contient PostGIS/PostgreSQL dans un seul outil d'installation pour Windows, Apple OS/X et Linux. La suite contient aussi GeoServer, OpenLayers et d'autres outils de visualisations sur le web.

Note : Si vous souhaitez installer simplement PostgreSQL, cela peut se faire en téléchargeant directement le code source ou les binaires de PostgreSQL sur le site du projet http ://postgresql.org/download/. Après avoir installé PostgreSQL, utilisez l'outil "StackBuilder" pour ajouter l'extension PostGIS à votre installation.

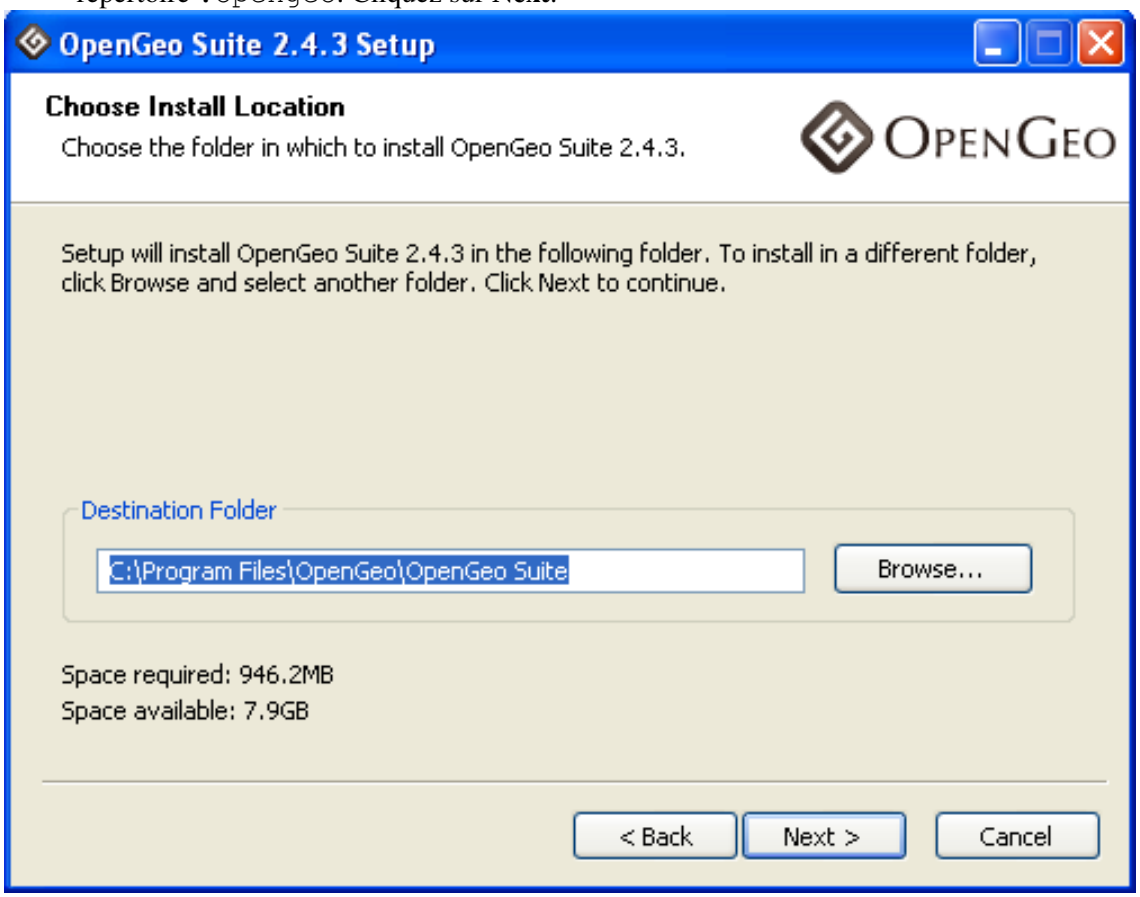
Note : Les indications précises de ce document sont propre à Windows, mais l'installation sous OS/X est largement similaire. Une fois la Suite installée, les instructions relatives au système d'exploitation devraient être identiques.

1. Dans le répertoire `postgisintro\software\` vous trouverez l'installeur de OpenGeo Suite nommé : `opengeosuite-2.4.3.exe` (sur OS/X, `opengeosuite-2.4.3.dmg`). Double cliquez sur cet exécutable pour le lancer.
2. Appréciez le message de courtoisie d'OpenGeo, puis cliquez sur **Next**.

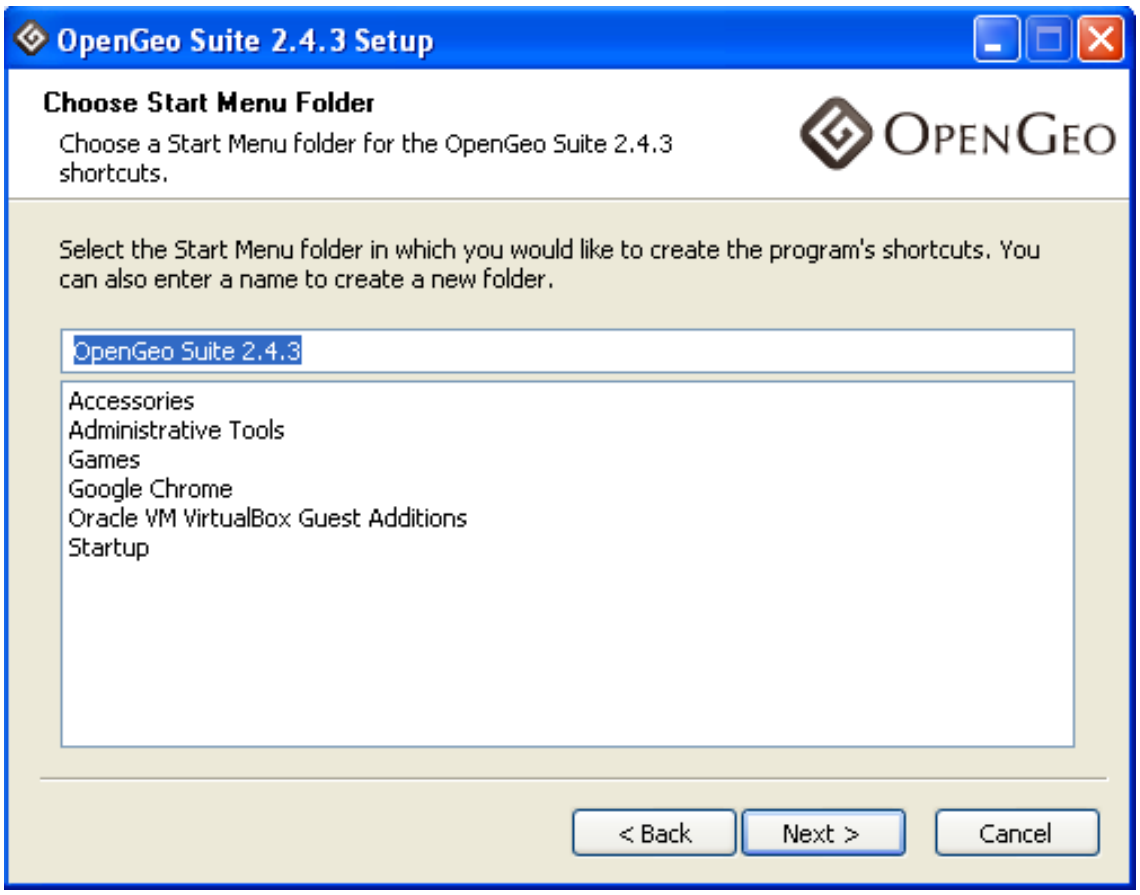
3. OpenGeo Suite est publiée sous licence GPL, ce qui est précisé dans la fenêtre de license. Cliquez sur **I Agree**.



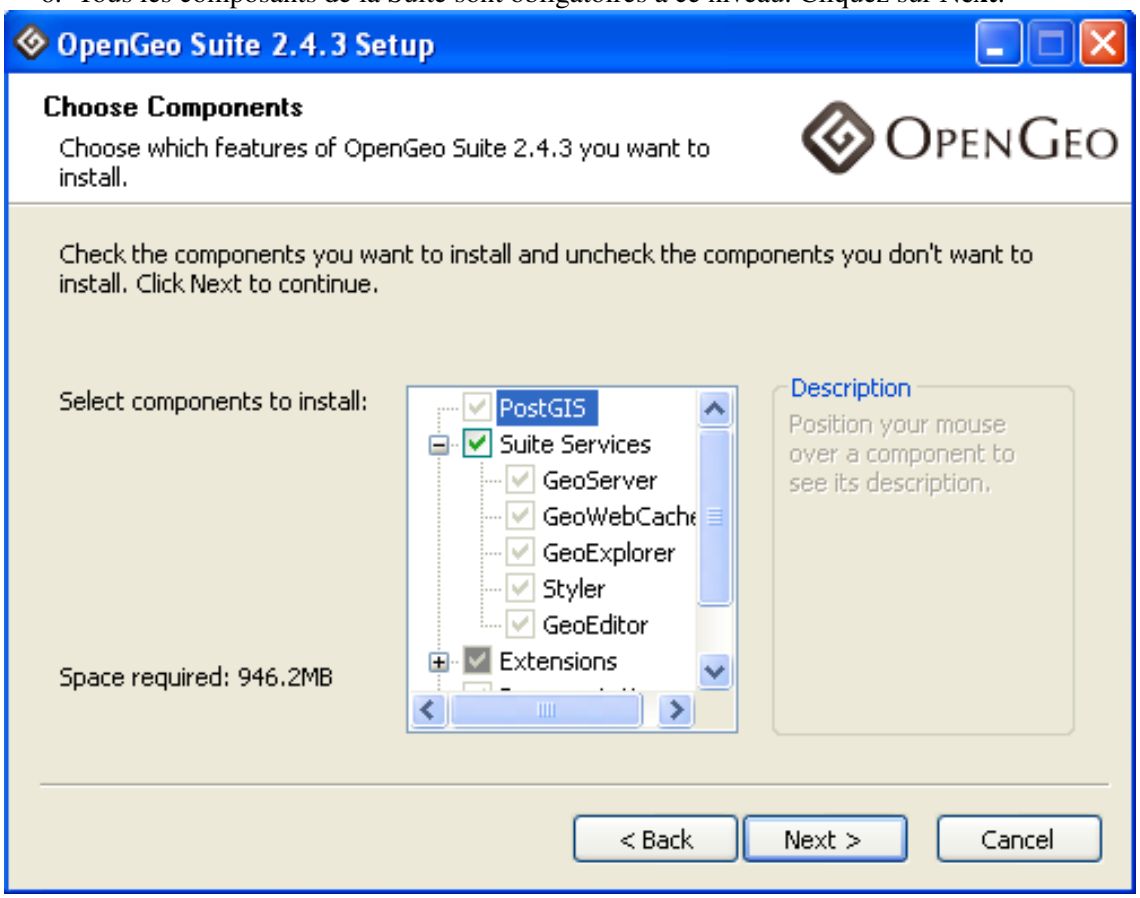
4. Le répertoire où OpenGeo Suite sera installé est généralement le répertoire `C:\Program Files\`. Les données seront placées dans le répertoire `.opengeo`. Cliquez sur **Next**.



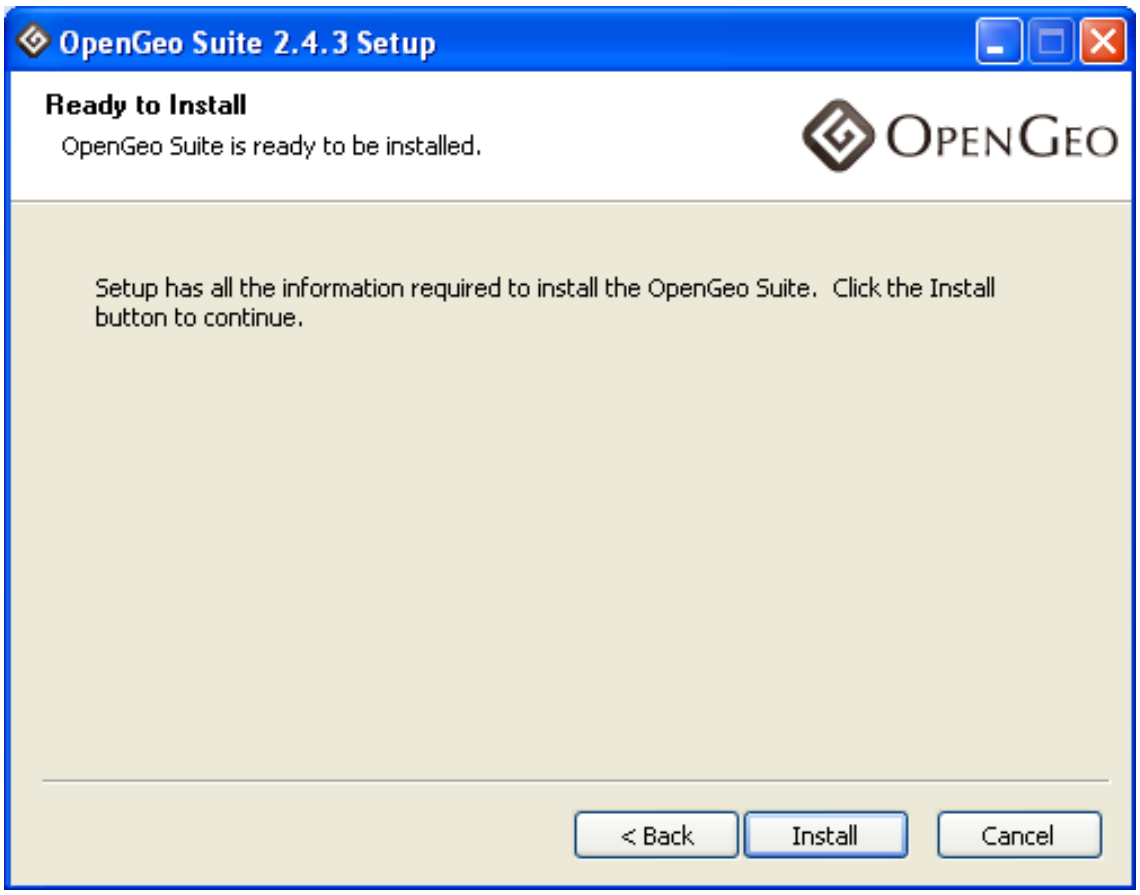
5. L'installeur créera un certain nombre de raccourcis dans le répertoire OpenGeo du menu Démarrer. Cliquez sur **Next**.



6. Tous les composants de la Suite sont obligatoires à ce niveau. Cliquez sur **Next**.



7. Prêt à installer ! Cliquez sur **Install**.



8. Le processus d'installation prendra quelques minutes.

9. Lorsque l'installation est terminée, lancez le Dashboard pour commencer la partie suivante de ces

travaux pratiques ! Cliquez sur **Finish**.

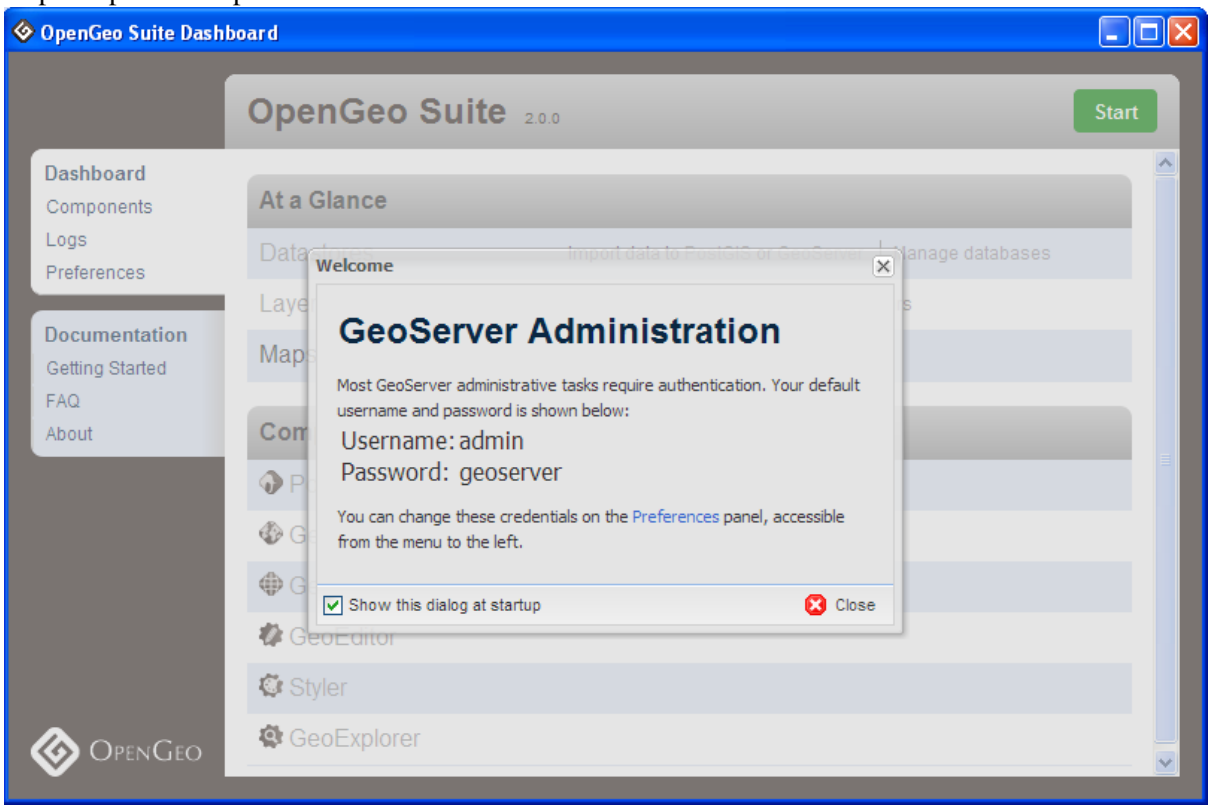
CHAPITRE 4

Partie 3 : Créer une base de données spatiales

4.1 Le Dashboard et PgAdmin

Le “Dashboard” est une application centralisant les accès aux différentes parties de l’openGeo Suite.

Lorsque vous démarrez le dashboard pour la première fois, il vous fournit une indication quand au mot de passe par défaut pour accéder à GeoServer.



Note : La base de données PostGIS a été installée sans la moindre restriction d’accès pour les utilisateurs locaux (les utilisateurs se connectant sur la même machine que celle faisant tourner le serveur de base de données). Cela signifie qu’il acceptera *tout* les mots de passe que vous fournirez. Si vous devez vous

connecter depuis un ordinateur distant, le mot de passe pour l'utilisateur `postgres` a utiliser est : `postgres`.

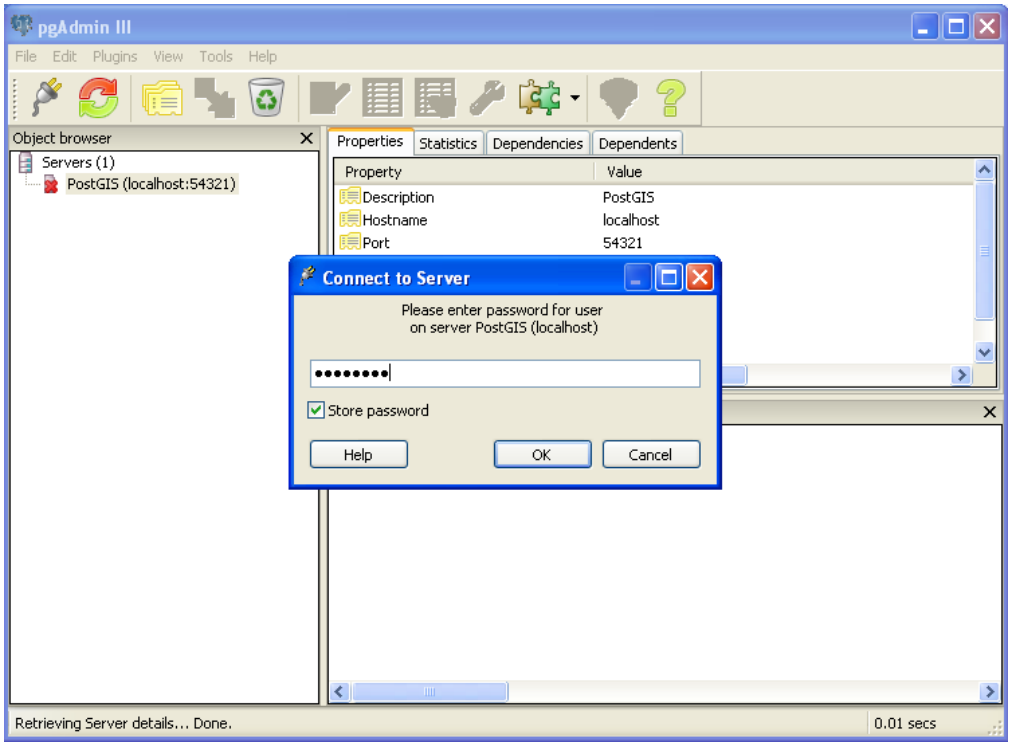
Pour ces travaux pratiques, nous n'utilserons que les parties de la section "PostGIS" du Dashboard.

1. Premièrement, nous devons démarrer le serveur de base de données PostGIS. Cliquez sur le bouton vert **Start** en haut à droite de la fenêtre du Dashboard.
2. La première fois que la Suite se démarre, elle initialise un espace de données et met en place des modèles de bases de données. Ceci peut prendre quelques minutes. Une fois la Suite lancée, vous pouvez cliquer sur l'option **Manage** dans le composant *PostGIS* pour lancer l'outil pgAdmin.

Note : PostgreSQL dispose de nombreux outils d'administration différents. Le premier est psql un outil en ligne de commande permettant de saisir des requêtes SQL. Un autre outil d'administation populaire est l'outil graphique libre et gratuit pgAdmin. Toutes les requêtes exécutées depuis pgAdmin peuvent aussi être utilisées depuis la ligne de commande avec psql.

---

3. Si c'est la première fois que vous lancez pgAdmin, vous devriez avoir une entrée du type **PostGIS (localhost :54321)** déjà configurée dans pgAdmin. Double cliquez sur cet élément, et entrez le mot de passe de votre choix pour vous connecter au serveur.

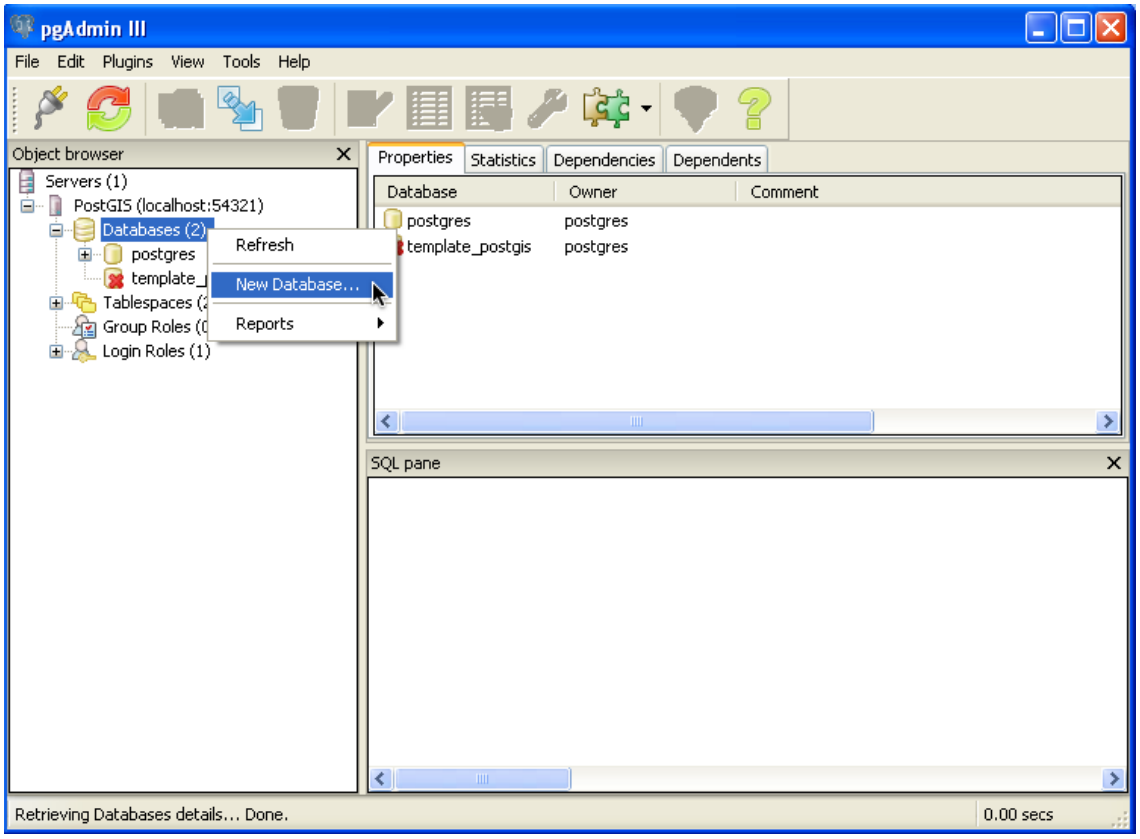


Note : Si vous aviez déjà une installation pgAdmin sur votre ordinateur, vous n'aurez pas l'entrée **(localhost :54321)**. Vous devrez donc créer une nouvelle connexion. Allez dans *File* > *Add Server*, puis enregistrez un nouveau serveur pour **localhost** avec le port **54321** (notez que numéro de port n'est pas standard) afin de vous connecter au serveur PostGIS installé à l'aide de l'OpenGeo Suite.

4.2 Créer une base de données

PostgreSQL fournit ce que l'on appelle des modèles de bases de données qui peuvent être utilisés lors de la création d'une nouvelle base. Cette nouvelle base contiendra alors une copie de tout ce qui est présent dans le modèle. Lorsque vous installez PostGIS, une base de données appelée `template_postgis` a été crée. Si nous utilisons `template_postgis` comme modèle lors de la création de notre nouvelle base, la nouvelle base sera une base de données spatiales.

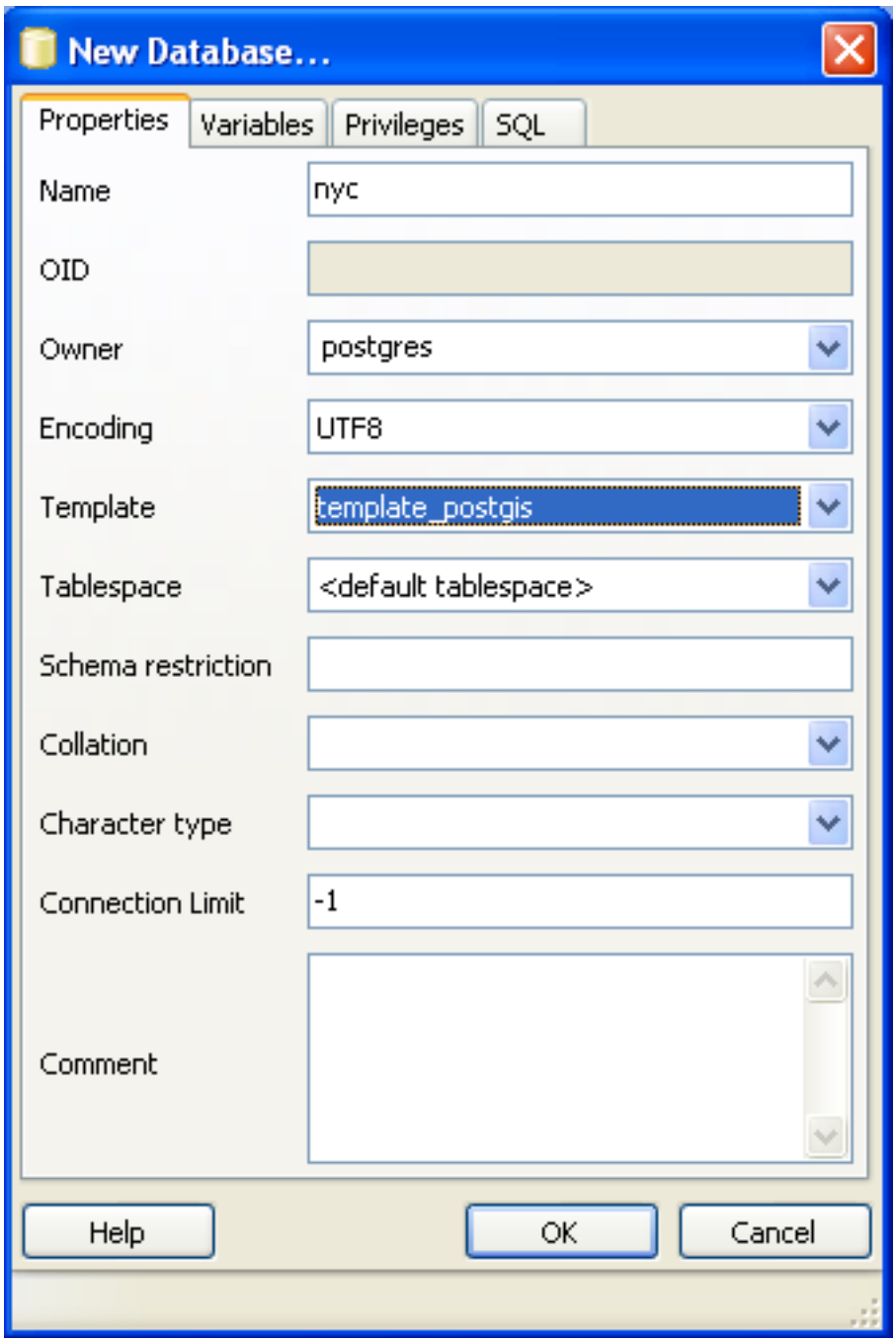
1. Ouvrez l'arbre des bases de données et regardez quelles sont les bases de données disponibles. La base `postgres` est la base de l'utilisateur (par défaut l'utilisateur postgres, donc pas très intéressante pour nous). La base `template_postgis` est celle que nous utiliserons pour créer des bases de données spatiales.
2. Cliquez avec le clic droit sur l'élément `Databases` et sélectionnez `New Database`.



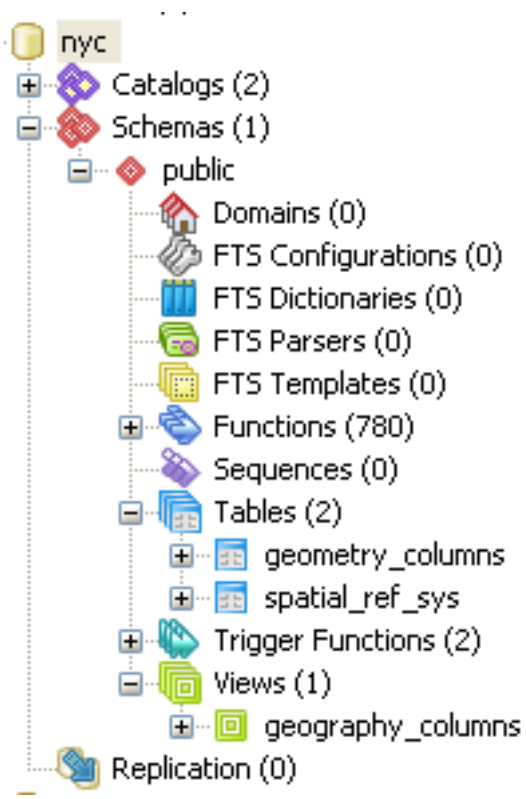
Note : Si vous recevez un message d'erreur indiquant que la base de données (`template_postgis`) est utilisée par d'autre utilisateurs, cela signifie que vous l'avez activé par inadvertance. Utilisez alors le clic droit sur l'élément `PostGIS (localhost:54321)` puis sélectionnez `Disconnect`. Double cliquez sur le même élément pour vous reconnecter et essayez à nouveau.

3. Remplissez le formulaire `New Database` puis cliquez sur **OK**.

Name	`nyc`
Owner	`postgres`
Encoding	`UTF8`
Template	`template_postgis`



4. Sélectionnez la nouvelle base de données `nyc` et ouvrez-la pour consulter son contenu. Vous verrez le schéma `public`, et sous cela un ensemble de tables de métadonnées spécifiques à PostGIS – `geometry_columns` et `spatial_ref_sys`.



5. Cliquez sur le bouton SQL query comme présenté ci-dessous (ou allez dans *Tools* > *Query Tool*).

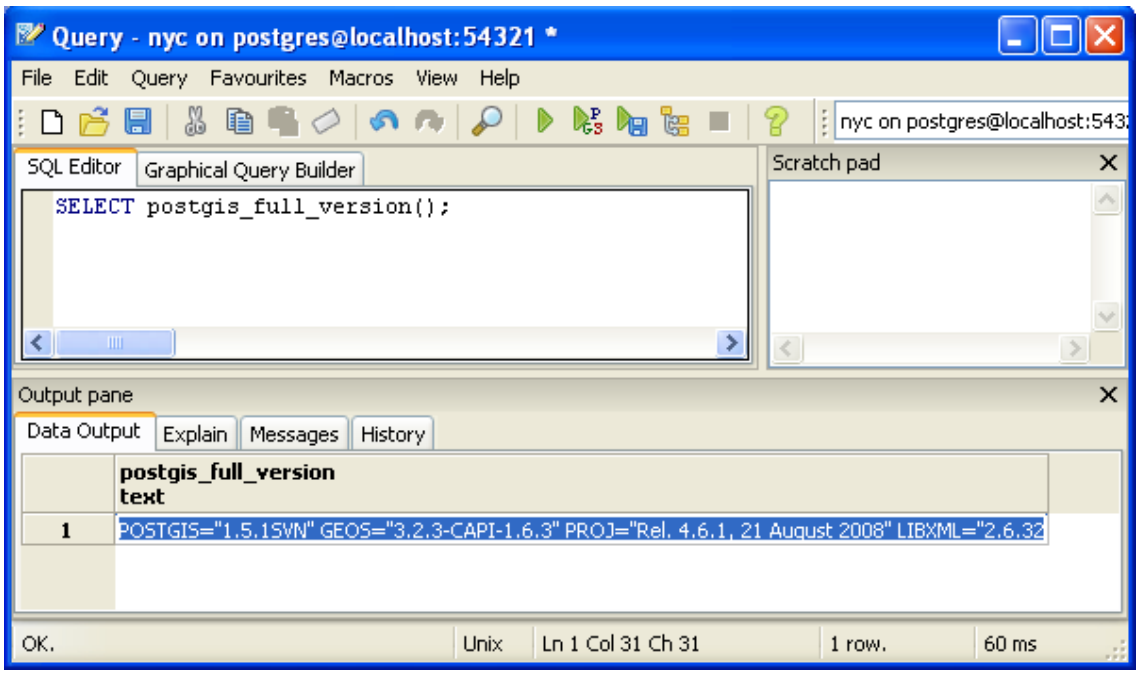
6. Saisissez la requête suivante dans le champ prévu à cet effet :

```
SELECT postgis_full_version();
```

Note : C'est notre première requête SQL. `postgis_full_version()` est une fonction d'administration qui renvoie le numéro de version et les options de configuration utilisées lors de la compilation.

---

7. Cliquez sur le bouton **Play** dans la barre d'outils (ou utilisez la touche **F5**) pour "exécuter la requête." La requête retournera la chaîne de caractères suivante, confirmant que PostGIS est correctement activé dans la base de données.



Vous venez de créer une base de données PostGIS avec succès !

4.3 Liste des fonctions

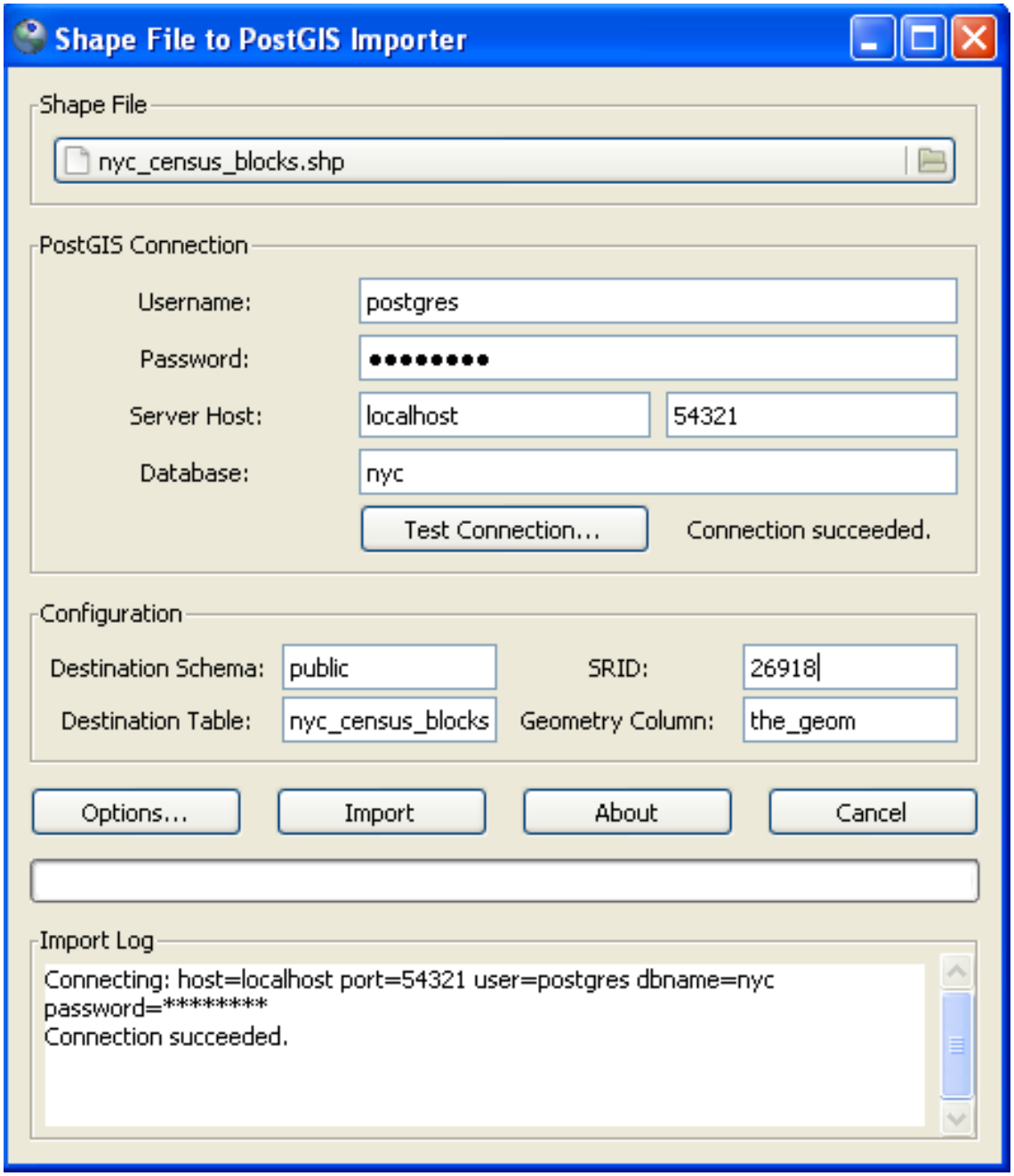
PostGIS_Full_Version : Retourne les informations complètes relatives à la version et aux options de compilation de PostGIS.

CHAPITRE 5

Partie 4 : Charger des données spatiales

Supporté par une grande variété de librairies et d'applications, PostGIS fournit de nombreux outils pour charger des données. Cette partie traitera uniquement du chargement basique de données, c'est à dire le chargement de fichiers Shapefile (.shp) en utilisant l'outil dédié de PostGIS.

1. Premièrement, retournez sur le Dashboard et cliquez sur le lien **Import shapefiles** de la section PostGIS. L'interface d'import de données Shapefile pgShapeLoader se lance.



2. Ensuite, ouvrez le navigateur de fichier *Shape File* puis dans le répertoire file :*postgisintro**data* sélectionnez le fichier `nyc_census_blocks.shp`.
3. Saisissez les détails de la section *connexion PostGIS* et cliquez sur le bouton **Test Connection...**.

Username	`postgres`
Password	`postgres`
Server Host	`localhost 54321`
Database	`nyc`

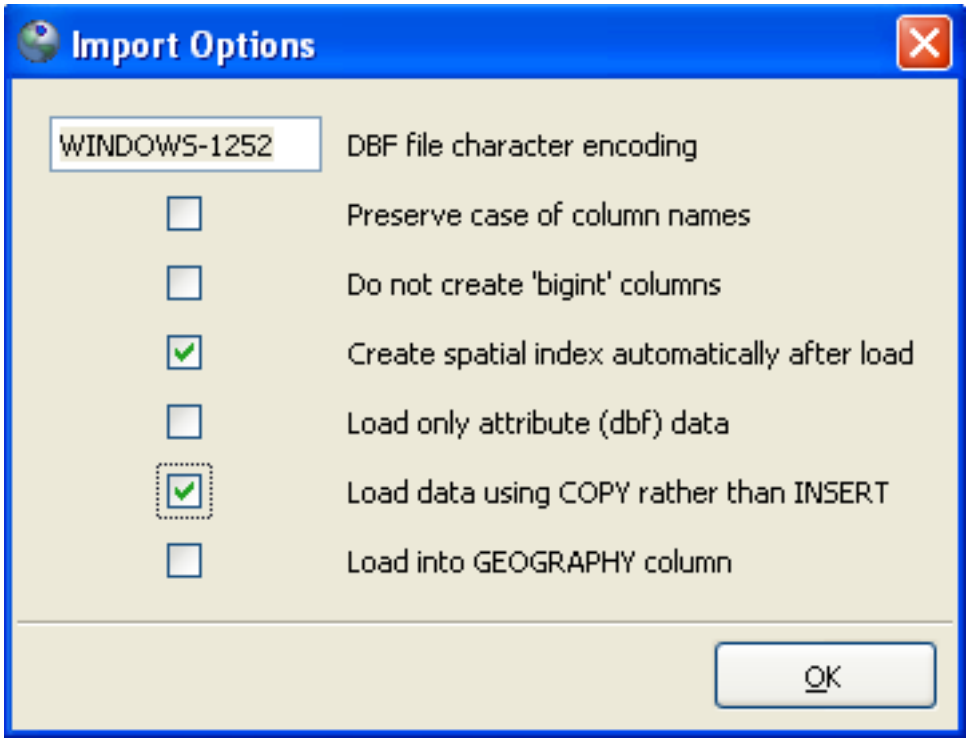
Note : Affecter le numéro de port **54321** est très important ! Le serveur PostGIS d'OpenGeo utilise ce port et non le port par défaut (5432).

---

1. Saisissez les détails de la section *Configuration*.

Destination Schema	`public`
SRID	`26918`
Destination Table	`nyc_census_blocks`
Geometry Column	`the_geom`

2. Cliquez sur le bouton **Options** et sélectionnez "Load data using COPY rather than INSERT." Ce qui implique que le chargement des données sera plus rapide.



3. Pour finir, cliquez sur le bouton **Import** et regardez l'importation s'exécuter. Cela peut prendre plusieurs minutes pour charger, mais ce fichier est le plus gros que nous aurons à charger.
4. Repétez la méthode afin d'importer les autres données présentes dans le répertoire data. Hormis le nom du fichier et le nom de la table de sortie, les autres paramètres de pgShapeLoader devrait rester les mêmes :
   - `nyc_streets.shp`
   - `nyc_neighborhoods.shp`
   - `nyc_subway_stations.shp`
5. Lorsque tous les fichiers sont chargés, cliquez sur le bouton "Refresh" de pgAdmin pour mettre à jour l'arbre affiché. Vous devriez voir vos quatre nouvellles tables affichées dans la section **Tables** de l'arbre.

5.1 Shapefile ? Qu'est-ce que c'est ?

Il est possible que vous vous demandiez "Qu'est-ce que c'est ce shapefile ?" On utilise communément le terme "Shapefile" pour parler d'un ensemble de fichiers d'extension `.shp`, `.shx`, `.dbf`, ou autre ayant un nom commun (ex : nyc_census_blocks). Le fichier Shapefile est en réalité le fichier d'extension `.shp`, mais ce fichier seul n'est pas complet sans ses fichiers associés.

Fichiers obligatoires :

- `.shp` — les formes ; les entités géographiques elle-mêmes
- `.shx` — l'index de formes ; un index basé sur les positions des entités géographiques
- `.dbf` — les attributs ; les données attributaires associées à chaque forme, au format dBase III

Les fichiers optionnels possibles :

- `.prj` — la projection ; le système de coordonnées et l'information de projection, un fichier texte décrivant la projection utilisant le format texte bien connu (WKT)

Afin d'utiliser un fichier Shapefile dans PostGIS, vous devez le convertir en une série de requêtes SQL. En utilisant pgShapeLoader, un Shapefile est converti en une table que PostgreSQL peut comprendre.

5.2 SRID 26918 ? Qu'est que c'est ?

La plupart des paramètres de l'importation de données sont explicites mais même les professionnels du SIG peuvent rencontrer des difficulté à propos du **SRID**.

"SRID" signifie "IDentifiant de Référence Spatiale". Il définit tous les paramètres de nos données, telles les coordonnées géographiques et la projection. Un SRID est pratique car il encapsule sous la forme d'un nombre toutes les informations à propos de la projection de la carte (ce qui peut être très compliqué).

Vou pouvez consulter la définition de la projection de la carte en consultant la base de données en ligne suivante :

http ://spatialreference.org/ref/epsg/26918/

ou directement depuis PostGIS en interrogeant la table `spatial_ref_sys`.

```
SELECT srtext FROM spatial_ref_sys WHERE srid = 26918;
```

Note : La table `spatial_ref_sys` de PostGIS est une table standard OGC qui définit tous les systèmes de référence spatiale connus par la base de données. Les données livrées avec PostGIS, contiennent 3000 systèmes de référence spatiale et précisent les informations nécessaires à la transformation ou la reprojection.

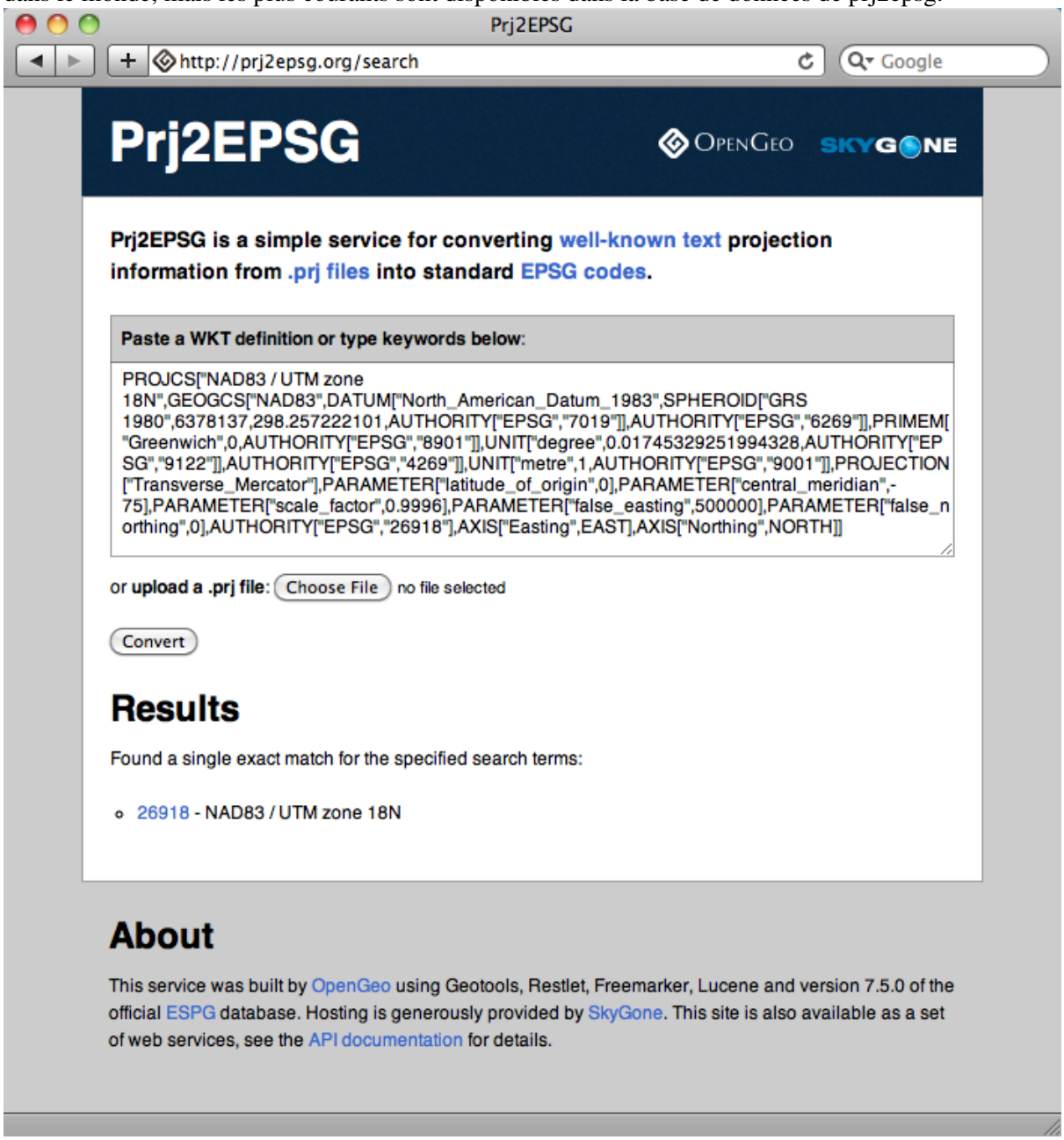
Dans les deux cas, vous obtiendrez une représentation du système de référence spatiale **26918** (affichée sur plusieurs lignes ici pour plus de clarté).

```
PROJCS["NAD83 / UTM zone 18N",
  GEOGCS["NAD83",
    DATUM["North_American_Datum_1983",
      SPHEROID["GRS 1980",6378137,298.257222101,AUTHORITY["EPSG","7019"]],
      AUTHORITY["EPSG","6269"]],
    PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.01745329251994328,AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4269"]],
  UNIT["metre",1,AUTHORITY["EPSG","9001"]],
  PROJECTION["Transverse_Mercator"],
  PARAMETER["latitude_of_origin",0],
  PARAMETER["central_meridian",-75],
  PARAMETER["scale_factor",0.9996],
  PARAMETER["false_easting",500000],
  PARAMETER["false_northing",0],
  AUTHORITY["EPSG","26918"],
  AXIS["Easting",EAST],
  AXIS["Northing",NORTH]]
```

Si vous ouvrez le fichier `nyc_neighborhoods.prj` du répertoire data, vous verrez la même définition.

Un problème auquel se confronte la plupart des débutants en PostGIS est de savoir quel SRID il doit utiliser pour ses données. Tout ce qu'ils ont c'est un fichier `.prj`. Mais comment un humain peut-il reconnaitre le numéro de SRID correct en lisant le contenu du fichier `.prj` ?

La réponse simple est d'utiliser un ordinateur. Copiez le contenu du fichier `.prj` dans le formulaire du site http ://prj2epsg.org. Cela vous donnera le nombre (ou la liste de nombres) qui correspond le plus à votre définition de projection. Il n'y a pas de nombre pour *toutes* les projections de cartes existantes dans le monde, mais les plus courants sont disponibles dans la base de données de prj2epsg.



Les données que vous recevez des agences locales de l'Etat - comme la ville de New York - utilisent la plupart du temps des projections locales notées "state plane" ou "UTM". Dans notre cas, la projection est "Universal Transverse Mercator (UTM) Zone 18 North" soit EPSG :26918.

5.3 Les choses à essayer : rendre spatiale une base de données existante

Vous avez déjà vu comment créer une base de données en utilisant le modèle `postgis_template` depuis pgAdmin. Néanmoins, lorsque vous installé depuis les sources ou que vous ajoutez le module PostGIS à une base existante, il n'est pas toujours approprié de créer une nouvelle base de données en utilisant le modèle PostGIS.

Votre tâche consiste dans cette section à créer une base de données et à ajouter les types et les fonctions PostGIS ensuite. Les script SQL nécessaires - `postgis.sql` et `spatial_ref_sys.sql` - se trouvent dans le répertoire `contrib` de votre installation de PostgreSQL. Pour vous guider, vous pouvez consulter la documentation PostGIS expliquant comment installer PostGIS [1].

Note : N'oubliez pas saisir le nom de l'utilisateur et le numéro de port lorsque vous créez une base de données en ligne de commande.

5.4 Les choses à essayer : visualiser des données avec uDig

uDig, (User-friendly Desktop Internet GIS) est un outil bureautique de visualisation/édition SIG permettant de visualiser rapidement ses données. Vous pouvez visualiser un grand nombre de formats différents dont les Shapefiles et les bases de données PostGIS. Son interface graphique vous permet d'explorer vos données facilement mais aussi de les tester et les styler rapidement.

Utilisez cette application pour vous connecter à votre base de données PostGIS. L'application est contenue dans le répertoire `software`.

1. "Chapter 2.5. Installation" PostGIS Documentation. Mai 2010 <http ://postgis.org/documentation/manual-1.5/ch02.html#id2786223>

CHAPITRE 6

Partie 5 : A propos de nos données

Les données utilisées dans ces travaux pratiques sont quatre shapefiles de la ville de New York, et une table attributaire des variables socio-démographiques de la ville. Nous les avons chargés sous forme de tables PostGIS et nous ajouterons les données socio-démographiques plus tard.

Cette partie fournit le nombre d'enregistrements et les attributs de chacun de nos ensembles de données. Ces valeurs attributaires et les relations sont essentielles pour nos futures analyses.

Pour visualiser la nature de vos tables depuis pgAdmin, cliquez avec le bouton droit sur une table et sélectionnez **Properties**. Vous trouverez un résumé des propriétés de la table, incluant la liste des attributs d'une table dans l'onglet **Columns**.

6.1 nyc_census_blocks

Un bloc recensé est la plus petite entité géographique pour laquelle un recensement est rapporté. Toutes les couches représentant les niveaux supérieurs (régions, zones de métro, comtés) peuvent être contruites à partir de ces blocs. Nous avons attaché des données démographiques aux blocs.

Nombre d'enregistrements : 36592

blkid	Un code à 15 chiffres qui permet d'identifier de manière unique chaque bloc **block**. Eg : 360050001009000
popn_total	Nombre total de personnes dans le bloc
popn_white	Nombre de personnes se déclarant comme de couleur blanche
popn_black	Nombre de personnes se déclarant comme de couleur noire
popn_nativ	Nombre de personnes se déclarant comme natif d'Amérique du nord
popn_asian	Nombre de personnes se déclarant comme asiatique
popn_other	Nombre de personnes se déclarant comme faisant partie d'une autre catégorie
hous_total	Nombre de pièces dans le bloc
hous_own	Nombre de propriétaires occupant le bloc
hous_rent	Nombre de locataires occupant le bloc
boroname	Nom du quartier (Manhattan, The Bronx, Brooklyn, Staten Island, Queens)
the_geom	Polygone représentant les contours d'un bloc

Note : Pour disposer des données d'un recensement dans votre SIG, vous avez besoin de joindre deux informations : Les données socio-démographiques et les limites géographiques des blocs/quartiers. Il

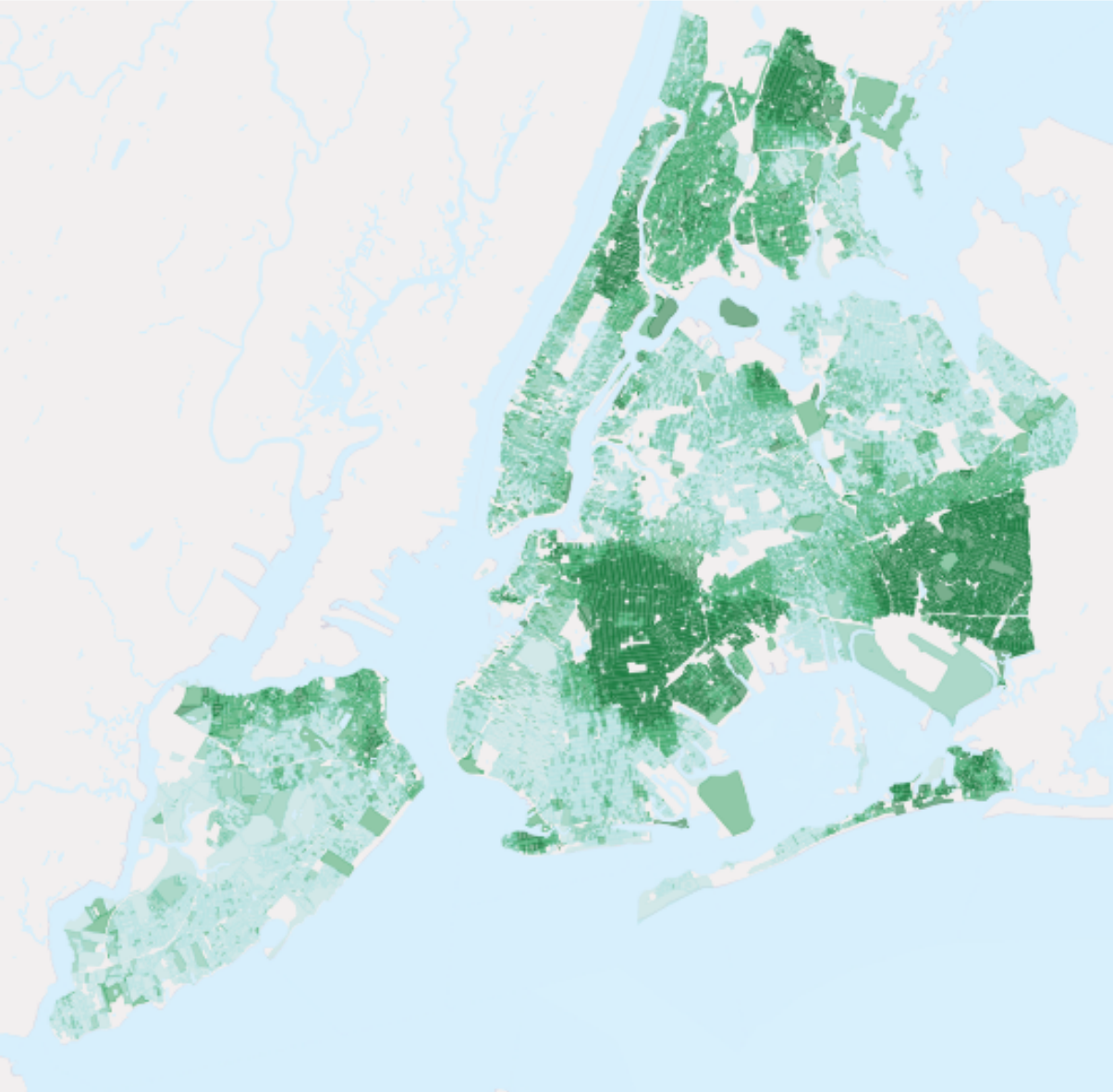


FIGURE 6.1 – *Pourcentage de la population qui est de couleur noire*

existe plusieurs moyen de se les procurer, dans notre cas, elles ont été récupérées sur le site Internet du Census Bureau's American FactFinder.

6.2 nyc_neighborhoods

Les quartiers de New York

Nombre d'enregistrements : 129

name	Nom du quartier
boroname	Nom de la section dans New York (Manhattan, The Bronx, Brooklyn, Staten Island, Queens)
the_geom	Limite polygonale du quartier

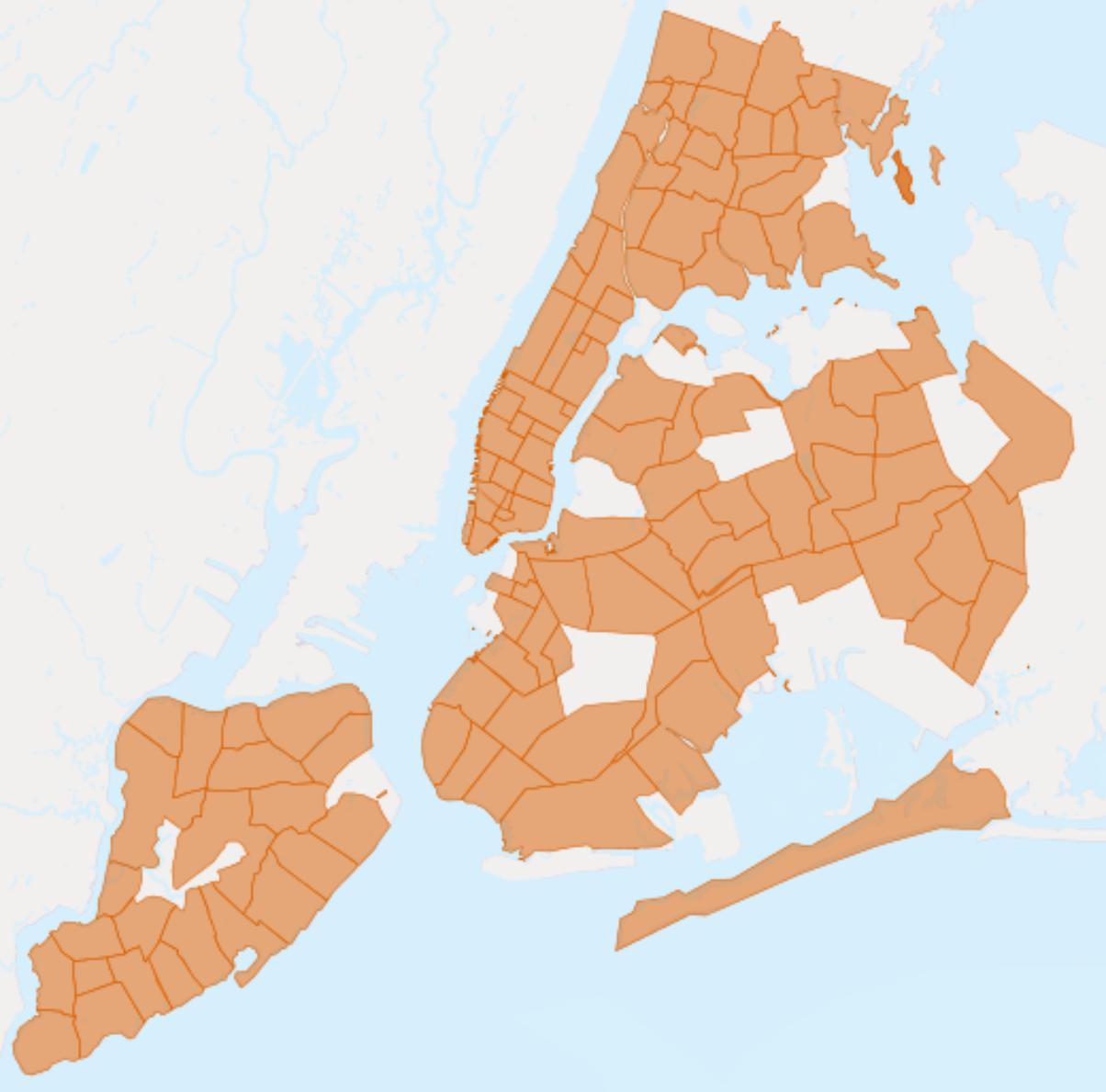


FIGURE 6.2 – *Les quartiers de New York*

6.3 nyc_streets

Les rues de New York

Nombre d'enregistrements : 19091

name	Nom de la rue
oneway	Est-ce que la rue est à sens unique ? "yes" = yes, "" = no
type	Type de voie (Cf : primary, secondary, residential, motorway)
the_geom	Ligne du centre de la rue.

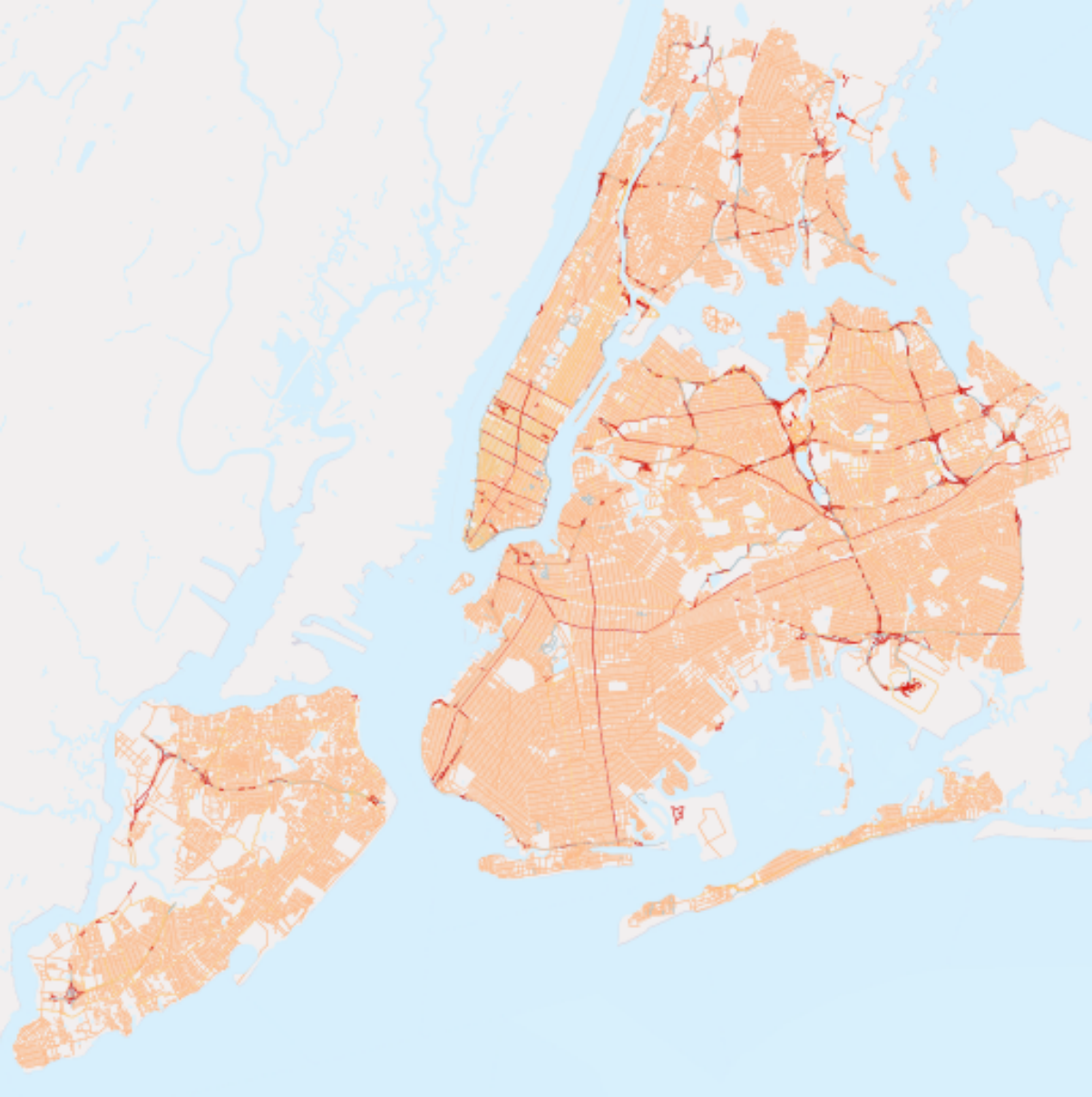


FIGURE 6.3 – *Les rues de New York (les rues principales apparaissent en rouge)*

6.4 nyc_subway_stations

Les stations de métro de New York

Nombre d'enregistrements : 491

name	Nom de la station
borough	Nom de la section dans New York (Manhattan, The Bronx, Brooklyn, Staten Island, Queens)
routes	Lignes de métro passant par cette station
transfers	Lignes de métro accessibles depuis cette station
express	Stations ou le train express s'arrête, "express" = yes, "" = no
the_geom	Localisation ponctuelle de la station

FIGURE 6.4 – *Localisation ponctuelle des stations de métro de New York*

6.5 nyc_census_sociodata

Données socio-démographiques de la ville de New York

Note : La donnée `nyc_census_sociodata` est une table attributaire. Nous devrons nous connecter aux géométries correspondant à la zone du recensement avant de conduire toute analyse spatiale .

tractid	Un code à 11 chiffre qui identifie chaque secteur de recensement. **tract**. Eg : 36005000100
transit_total	Nombre de travailleurs dans le secteur
transit_public	Nombre de travailleurs dans le secteur utilisant les transports en commun
transit_private	Nombre de travailleurs dans le secteur utilisant un véhicule privé
transit_other	Nombre de travailleurs dans le secteur utilisant un autre moyen de transport
transit_time_mins	Nombre total de minutes passées dans les transports par l'ensemble des travailleurs du secteur (minutes)
family_count	Nombre de familles dans le secteur
family_income_median	Revenu médian par famille du secteur (dollars)
family_income_aggregate	Revenu total de toutes les familles du secteur (dollars)
edu_total	Nombre de personnes ayant un parcours scolaire
edu_no_highschool_dipl	Nombre de personnes n'ayant pas de diplôme d'éducation secondaire
edu_highschool_dipl	Nombre de personnes ayant un diplôme d'éducation secondaire
edu_college_dipl	Nombre de personnes ayant un diplôme de lycée
edu_graduate_dipl	Nombre de personnes ayant un diplôme de collège

CHAPITRE 7

Partie 6 : Requêtes SQL simples

SQL, pour "Structured Query Language", définit la manière d'importer et d'interroger des données dans une base. Vous avez déjà rédigé du SQL lorsque nous avons créé notre première base de données.

Rappel :

```
SELECT postgis_full_version();
```

Maintenant que nous avons chargé des données dans notre base, essayons d'utiliser SQL pour les interroger. Par exemple,

> "Quels sont les noms des quartiers de la ville de New York ?"

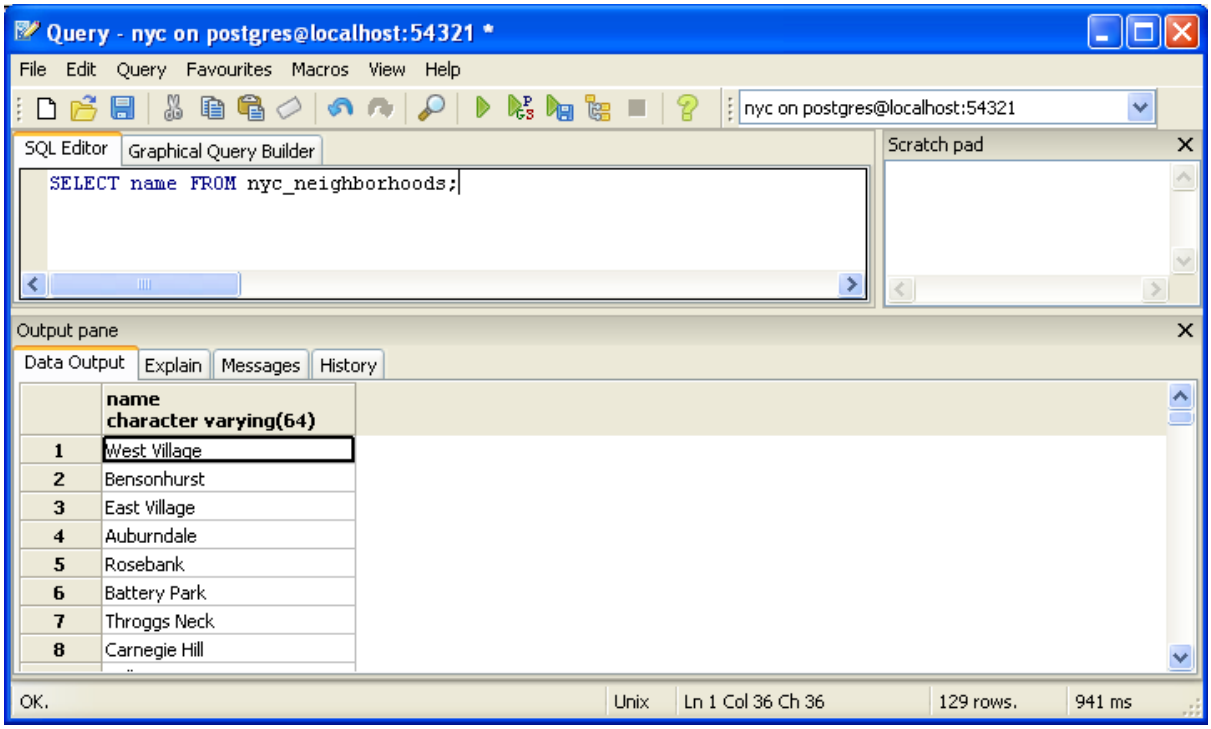
Ouvrez une fenêtre SQL depuis pgAdmin en cliquant sur le bouton SQL

Puis saisissez la requête suivante dans la fenêtre

```
SELECT name FROM nyc_neighborhoods;
```

et cliquez sur le bouton **Execute Query** (le triangle vert).

La requête s'exécutera pendant quelques (mili)secondes et retournera 129 résultats.



Mais que s'est-il exactement passé ici ? Pour le comprendre, commençons par présenter les quatre types de requêtes du SQL :

- `SELECT`, retourne des lignes en réponse à une requête
- `INSERT`, ajoute des lignes dans une table
- `UPDATE`, modifie des lignes existantes d'une table
- `DELETE`, supprime des lignes d'une table

Nous travaillerons principalement avec des requêtes de type `SELECT` afin d'interroger les tables en utilisant des fonctions spatiales.

7.1 Requête de type SELECT

Une requête de type Select est généralement de la forme :

> SELECT colonnes FROM données WHERE conditions ;

Note : Pour une description exhaustive des paramètres possible d'une requête `SELECT`, consultez la documentation de PostgresSQL.

Les `colonnes` sont soit des noms de colonnes, soit des fonctions utilisant les valeurs des colonnes. Les `données` sont soit une table seule, soit plusieurs tables reliées ensemble en réalisant une jointure sur une clef ou une autre condition. Les `conditions` représentent le filtre qui restreint le nombre de lignes à retourner.

> "Quel sont les noms des quartiers de Brooklyn ?"

Nous retournons à notre table `nyc_neighborhoods` avec le filtre en main. La table contient tous les quartiers de New York et nous voulons uniquement ceux de Brooklyn.

```
SELECT name
  FROM nyc_neighborhoods
  WHERE boroname = 'Brooklyn';
```

La requête prendra à nouveau quelque (milli)secondes et retournera les 23 éléments résultants.

Parfois, nous aurons besoin d'appliquer des fonctions sur le résultat d'une de nos requêtes. Par exemple,

> "Quel est le nombre de lettres dans les noms des quartiers de Brooklyn ?"

Heureusement PostgreSQL fournit une fonction calculant la longueur d'une chaîne de caractères : **char_length(string)**.

```
SELECT char_length(name)
  FROM nyc_neighborhoods
  WHERE boroname = 'Brooklyn';
```

Bien souvent, nous sommes moins interessés par une ligne particulière que par un calcul statistique sur l'ensemble résultant. Donc, connaître la longueur des noms de quartiers est moins intéressant que de calculer la moyenne de ces longueurs. Les fonctions qui renvoient un résultat unique en utilisant un ensemble de valeurs sont appelées des "fonctions d'aggrégations".

PostgreSQL fournit un ensemble de fonctions d'aggrégations, parmi lesquelles **avg()** pour calculer la moyenne, and **stddev()** pour l'écart type.

> "Quel est le nombre moyen et l'écart type du nombre de lettres dans le nom des quartier de Brooklyn ?"

```
SELECT avg(char_length(name)), stddev(char_length(name))
  FROM nyc_neighborhoods
  WHERE boroname = 'Brooklyn';
```

```
         avg          |       stddev
----------------------+--------------------
 11.7391304347826087 | 3.9105613559407395
```

Les fonctions d'agrégation dans notre dernier exemple sont appliquées à chaque ligne de l'ensemble des résultats. Comment faire si nous voulons rassembler des données ? Pour cela, nous utilisons la clause `GROUP BY`. Les fonctions d'agrégation ont souvent besoin d'une clause `GROUP BY` pour regrouper les éléments en utilisant une ou plusieurs colonnes.

> "Quel est la moyenne du nombre de caractères des noms de quartiers et l'écart-type du nombre de caractères des noms de quartiers, renvoyé par section de New York ?"

```
SELECT boroname, avg(char_length(name)), stddev(char_length(name))
  FROM nyc_neighborhoods
  GROUP BY boroname;
```

Nous ajoutons la colonne `boroname` dans le résultat afin de pouvoir déterminer quelle valeur statistique s'applique à quelle section. Dans une requête agrégée, vous pouvez seulement retourner les colonnes qui sont (a) membre de la clause de regroupement ou (b) des fonctions d'agrégation.

```
   boroname     |         avg          |       stddev
----------------+----------------------+--------------------
 Brooklyn       | 11.7391304347826087 | 3.9105613559407395
 Manhattan      | 11.8214285714285714 | 4.3123729948325257
 The Bronx      | 12.0416666666666667 | 3.6651017740975152
 Queens         | 11.6666666666666667 | 5.0057438272815975
 Staten Island  | 12.2916666666666667 | 5.2043390480959474
```

7.2 Liste de fonctions

avg(expression) : fonction d'agrégation de PostgreSQL qui retourne la valeur moyenne d'une colonne.

char_length(string) : fonction s'appliquant aux chaînes de caractère de PostgreSQL qui retourne le nombre de lettres dans une chaîne.

stddev(expression) : fonction d'aggrégation de PostgreSQL qui retourne l'écart type d'un ensemble de valeurs.

CHAPITRE 8

Partie 7 : Exercices simples de SQL

En utilisant la table `nyc_census_blocks`, répondez au questions suivantes (et n'allez pas directement aux réponses !).

Vous trouverez ci-dessous des informations utiles pour commencer. Référez-vous à la partie *A propos des nos données* pour la définition de notre table `nyc_census_blocks`.

blkid	Un code à 15 chiffres qui définit de manière unique chaque **bloc** ressencé . Ex : 360050001009000
popn_total	Nombre total de personnes dans un bloc ressensé
popn_white	Nombre de personnes se déclarant "blancs"
popn_black	Nombre de personnes se déclarant "noirs"
popn_nativ	Nombre de personnes se déclarant comme "nés aux états-unis"
popn_asian	Nombre de personne se déclarant comme "asiatiques"
popn_other	Nombre de personne se déclarant d'une autre catégorie
hous_total	Nombre de pièces appartements
hous_own	Nombre de propriétaires occupant les appartements
hous_rent	Nombre de locations disponibles
boroname	Nom du quartier de New York. Manhattan, The Bronx, Brooklyn, Staten Island, Queens
the_geom	Polygone délimitant le bloc

Ici se trouvent certaines des fonctions d'aggrégation qui vous seront utiles pour répondre aux questions :
- avg() - la moyenne des valeurs dans un ensemble d'enregistrements
- sum() - la somme des valeurs d'un ensemble d'enregistrements
- count() - le nombre d'éléments contenus dans un ensemble d'enregistrements.

Maintenant les questions :
- **"Quelle est la population de la ville de New York ?"**

```
SELECT Sum(popn_total) AS population
  FROM nyc_census_blocks;
```

```
8008278
```

Note : Qu'est-ce que ce `AS` dans la requête ? vous pouvez donner un nom à une table ou à des colonnes en utilisant un alias. Les alias permettent de rendre les requêtes plus simple à écrire et à lire. Donc au lieu que notre colonne résultat soit nommée `sum` nous utilisons le **AS** pour la renommer en `population`.

– **"Quelle est la population du Bronx ?"**

```
SELECT Sum(popn_total) AS population
  FROM nyc_census_blocks
  WHERE boroname = 'The Bronx';
```

```
1332650
```

– **"Quelle est en moyenne le nombre de personnes vivant dans chaque appartement de la ville de New York ?"**

```
SELECT Sum(popn_total)/Sum(hous_total) AS popn_per_house
  FROM nyc_census_blocks;
```

```
2.6503540522400804
```

– **"Pour chaque quartier, quel est le pourcentage de population blanche ?"**

```
SELECT
    boroname,
    100 * Sum(popn_white)/Sum(popn_total) AS white_pct
  FROM nyc_census_blocks
  GROUP BY boroname;
```

```
   boroname     |      white_pct
---------------+---------------------
 Brooklyn       | 41.2005552206888663
 The Bronx      | 29.8655310846808990
 Manhattan      | 54.3594013771837665
 Queens         | 44.0806610271290794
 Staten Island  | 77.5968611401579346
```

8.1 Liste des fonctions

avg(expression) : fonction d'aggrégation de PostgreSQL qui renvoie la moyenne d'un ensemble de nombres.

count(expression) : une fonction d'aggrégation de PostgreSQL qui retourne le nombre d'éléments dans un ensemble.

sum(expression) : une fonction d'aggrégation de PostgreSQL qui retourne la somme des valeurs numériques d'un ensemble.

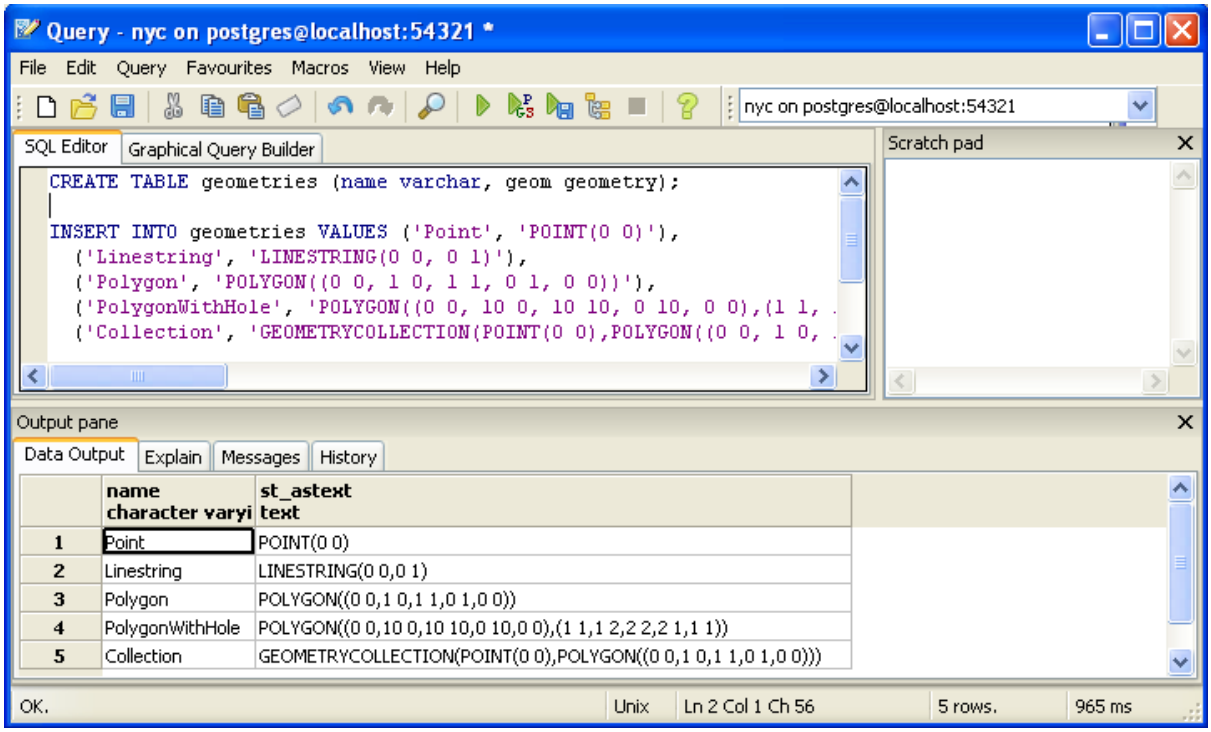
CHAPITRE 9

Partie 8 : Les géometries

9.1 Introduction

Dans *une partie précédente* nous avons chargé différentes données. Avant de commencer à jouer avec, commençons par regarder quelques exemples simples. Depuis pgAdmin, choisissez de nouveau la base de donnée **nyc** et ouvrez l'outil de requêtage SQL. Copiez cette exemple de code SQL (après avoir supprimé le contenu présent par défaut si nécessaire) puis exécutez-le.

```
CREATE TABLE geometries (name varchar, geom geometry);

INSERT INTO geometries VALUES
  ('Point', 'POINT(0 0)'),
  ('Linestring', 'LINESTRING(0 0, 1 1, 2 1, 2 2)'),
  ('Polygon', 'POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))'),
  ('PolygonWithHole', 'POLYGON((0 0, 10 0, 10 10, 0 10, 0 0),(1 1, 1 2, 2 2, 2 1, 1 1))'
  ('Collection', 'GEOMETRYCOLLECTION(POINT(2 0),POLYGON((0 0, 1 0, 1 1, 0 1, 0 0)))');

SELECT Populate_Geometry_Columns();

SELECT name, ST_AsText(geom) FROM geometries;
```

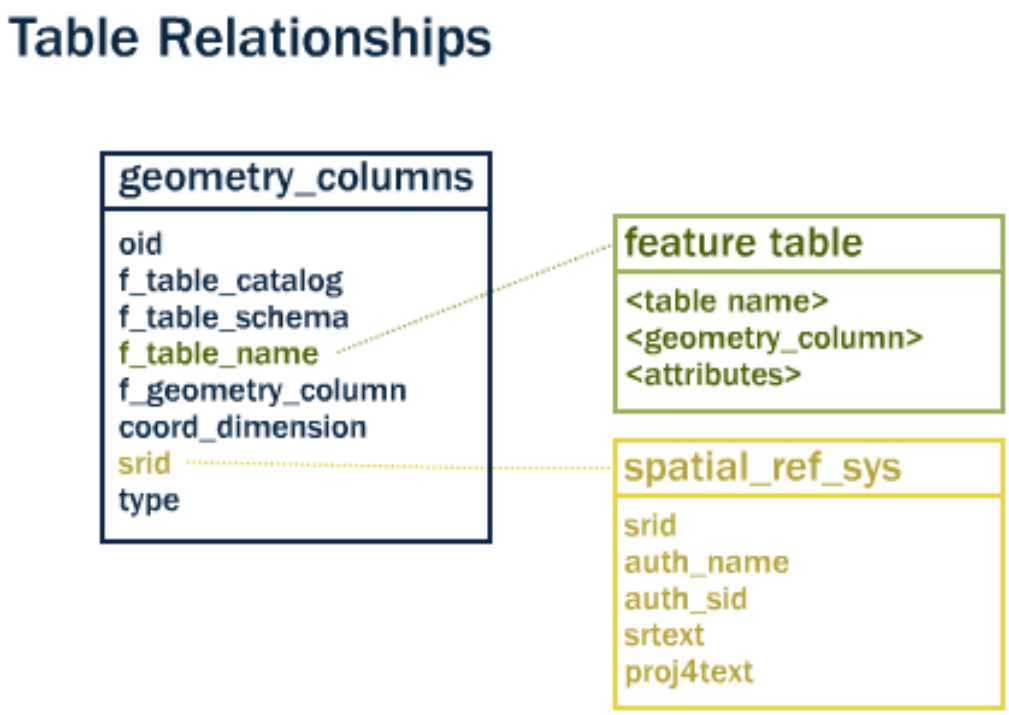


L'exemple ci-dessus créé une table (**geometries**) puis y insère cinq géométries : un point, une ligne, un polygone, un polygone avec un trou, et une collection. Les lignes insérées sont sélectionnées et affichées dans le tableau de sortie.

9.2 Les tables de métadonnées

Dans le respect de la spécification Simple Features for SQL (*SFSQL*), PostGIS fournit deux tables pour récupérer et s'informer sur les types de géométries disponibles dans une base de données spécifique.

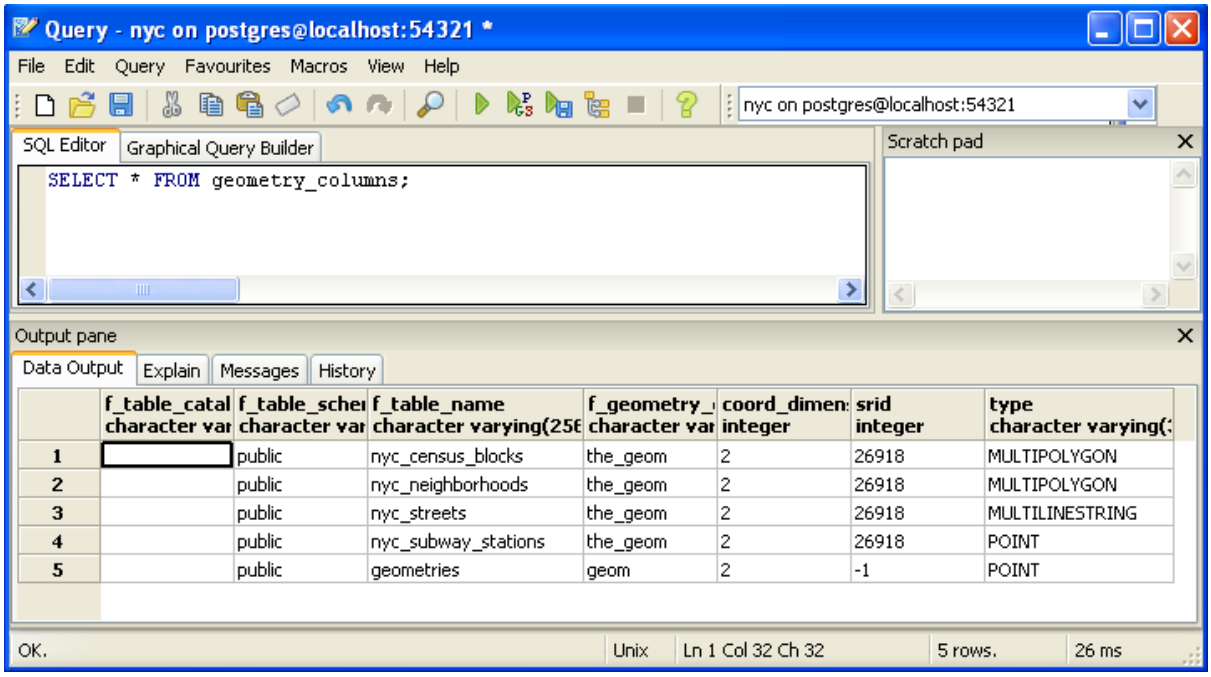
– La première table, `spatial_ref_sys`, définit tous les systèmes de projection connus de la base de données et sera décrite plus en détails plus tard.
– La seconde table, `geometry_columns`, fournit une liste de toutes les "entités" (définit comme un objet avec un attribut géométrique) et les détails de base relatives à ces entités.



Dans l'exemple fournit en introduction, la fonction **Populate_Geometry_Columns()** détecte toutes les colonnes de la base de données qui contiennent des géométries et met à jour la table `geometry_columns` pour y inclure leurs références.

Regardons maintenant la table `geometry_columns` de notre base de données. Copiez cette commande dans la fenêtre de requêtage :

```
SELECT * FROM geometry_columns;
```



– `f_table_catalog`, `f_table_schema`, et `f_table_name` fournissent le nom complet de la table contenant une géométrie donnée. Étant donné que PostgreSQL n'utilise pas de catalogues, `f_table_catalog` est toujours vide.
– `f_geometry_column` est le nom de la colonne qui contient la géométrie – pour les tables ayant plusieurs colonnes géométriques, il y a un enregistrement dans cette table pour chacune.
– `coord_dimension` et `srid` définissent respectivement la dimension de la géométrie (en 2-, 3- or 4-dimensions) et le système de référence spatiale qui fait référence à la table `spatial_ref_sys`.
– La colonne `type` définit le type de géométrie comme décrit plus tôt, nous avons déjà vu les points et les lignes.

En interrogeant cette table, les clients SIG et les libraires peuvent déterminer quoi attendre lors de la récupération des données et peuvent réaliser les opération de reprojection, transformation ou rendu sans avoir à inspecter chaque géométrie.

9.3 Réprésenter des objets du monde réel

La spécification Simple Features for SQL (*SFSQL*), le standard ayant guidé le développement de PostGIS, définit comment les objets du monde réel doivent être représentés. En considérant une forme continue à une seule résolution fixe, nous obtenons une piètre représentation des objets. SFSQL considère uniquement les représentations en 2 dimensions. PostGIS a étendu cela en ajoutant les représentation en 3 et 4 dimensions. Plus récemment, la spécification SQL-Multimedia Part 3 (*SQL/MM*) a officiellement définit sa propre représentation.

Notre table exemple contient différents types de géométries. Nous pouvons récupérer les informations de chaque objet en utilisant les fonctions qui lisent les métadonnées de la géométrie.
– **ST_GeometryType(geometry)** retourne le type de la géométrie
– **ST_NDims(geometry)** retourne le nombre de dimensions d'une géométrie
– **ST_SRID(geometry)** retourne l'identifiant de référence spatiale de la géométrie

```
SELECT name, ST_GeometryType(geom), ST_NDims(geom), ST_SRID(geom)
  FROM geometries;
```

```
      name        |    st_geometrytype     | st_ndims | st_srid
-----------------+------------------------+----------+---------
 Point            | ST_Point               |        2 |      -1
 Polygon          | ST_Polygon             |        2 |      -1
 PolygonWithHole  | ST_Polygon             |        2 |      -1
 Collection       | ST_GeometryCollection  |        2 |      -1
 Linestring       | ST_LineString          |        2 |      -1
```

9.3.1 Les points

Un **point** représente une localisation unique sur la Terre. Ce point est représenté par une seule coordonnée (incluant soit 2, 3 ou 4 dimensions). Les points sont utilisés pour représenter des objets lorsque

le détail exact du contour n'est pas important à une échelle donnée. Par exemple, les villes sur une carte du monde peuvent être décrites sous la forme de points alors qu'une carte régionale utiliserait une représentation polygonale des villes.

```
SELECT ST_AsText(geom)
  FROM geometries
  WHERE name = 'Point';
```

```
POINT(0 0)
```

Certaines des fonctions spécifiques pour travailler avec les points sont :
– **ST_X(geometry)** retourne la composante X
– **ST_Y(geometry)** retourne la composante Y
Donc, nous pouvons lire les coordonnées d'un point de la manière suivante :

```
SELECT ST_X(geom), ST_Y(geom)
  FROM geometries
  WHERE name = 'Point';
```

La table des stations de métro de la ville de New York (`nyc_subway_stations`) est un ensemble de données représenté sous la forme de points. La requête SQL suivante renverra la géométrie associée à un point (dans la colonne **ST_AsText**).

```
SELECT name, ST_AsText(the_geom)
  FROM nyc_subway_stations
  LIMIT 1;
```

9.3.2 Les lignes

Une **ligne** est un chemin entre plusieurs points. Elle prend la forme d'un tableau ordonné composé de deux (ou plusieurs) points. Les routes et les rivières sont typiquement représentées sous la forme de lignes. Une ligne est dite **fermée** si elle commence et finit en un même point. Elle est dite **simple** si elle ne se coupe pas ou ne se touche pas elle-même (sauf à ses extrémités si elle est fermée). Une ligne peut être à la fois **fermée** et **simple**.

Le réseau des rues de New York (`nyc_streets`) a été chargé auparavant. Cet ensemble de données contient les détails comme le nom et le type des rues. Une rue du monde réel pourrait être constituée de plusieurs lignes, chacune représentant une segment de routes avec ses différents attributs.

La requête SQL suivante retourne la géométrie associée à une ligne (dans la colonne **ST_AsText**) :

```
SELECT ST_AsText(geom)
  FROM geometries
  WHERE name = 'Linestring';
```

```
LINESTRING(0 0, 1 1, 2 1, 2 2)
```

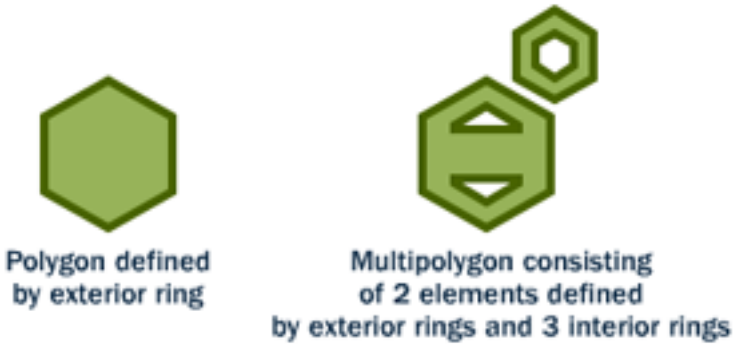
Les fonctions spatiales permettant de travailler avec les lignes sont les suivantes :
- **ST_Length(geometry)** retourne la longueur d'une ligne
- **ST_StartPoint(geometry)** retourne le premier point d'une ligne
- **ST_EndPoint(geometry)** retourne le dernier point d'une ligne
- **ST_NPoints(geometry)** retourne le nombre de points dans une ligne

Donc, la longueur de notre ligne est :

```
SELECT ST_Length(geom)
  FROM geometries
  WHERE name = 'Linestring';
```

```
3.41421356237309
```

9.3.3 Les polygones



Un polygone est représenté comme une zone. Le contour externe du polygone est représenté par une ligne simple et fermée. Les trous sont représentés de la même manière.

Les polygones sont utilisés pour représenter les objets dont les tailles et la forme sont importants. Les limites de villes, les parcs, les bâtiments ou les cours d'eau sont habituellement représentés par des polygones lorsque l'échelle est suffisament élevée pour pouvoir distinguer leurs zones. Les routes et les rivières peuvent parfois être représentées comme des polygones.

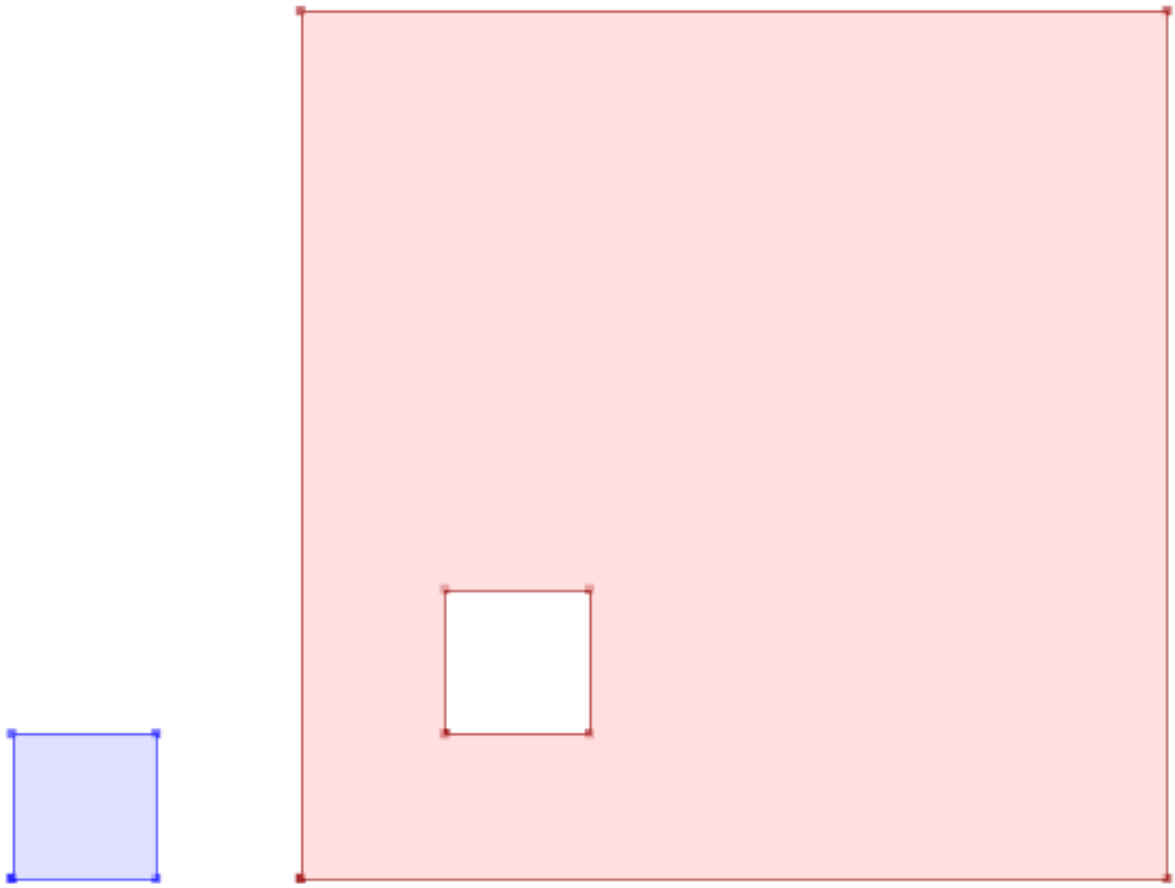
La requête SQL suivante retournera la géométrie associée à un polygone (dans la colonne **ST_AsText**).

```
SELECT ST_AsText(geom)
  FROM geometries
  WHERE name LIKE 'Polygon%';
```

Note : Plutôt que d'utiliser le signe = dans notre clause `WHERE`, nous avons utilisé l'opérateur `LIKE` pour pouvoir définir notre comparaison. Vous auriez sans doute voulu utiliser le symbole `*` pour exprimer "n'importe quelle valeur" mais en SQL vous devez utiliser : `%` et l'opérateur `LIKE` pour informer le système que cette comparaison doit être possible.

```
POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))
POLYGON((0 0, 10 0, 10 10, 0 10, 0 0),(1 1, 1 2, 2 2, 2 1, 1 1))
```

Le premier polygone a seulement une ligne. Le second a un “trou”. La plupart des systèmes de rendu graphique supportent le concept de “polygone”, mais les systèmes SIG sont les seuls à accepter que les polygones puissent contenir des trous.



Certaines des fonctions spatiales spécifiques de traitement des polygones sont :
– **ST_Area(geometry)** retourne l’aire d’un polygone
– **ST_NRings(geometry)** retourne le nombre de contours (habituellement 1, plus lorsqu’il y a des trous)
– **ST_ExteriorRing(geometry)** retourne le contour extérieur
– **ST_InteriorRingN(geometry,n)** retourne le contour intérieur numéro n
– **ST_Perimeter(geometry)** retourne la longueur de tous les contours
Nous pouvons calculer l’aire de nos polygones en utilisant la fonction area :

```
SELECT name, ST_Area(geom)
  FROM geometries
  WHERE name LIKE 'Polygon%';
```

```
Polygon            1
PolygonWithHole    99
```

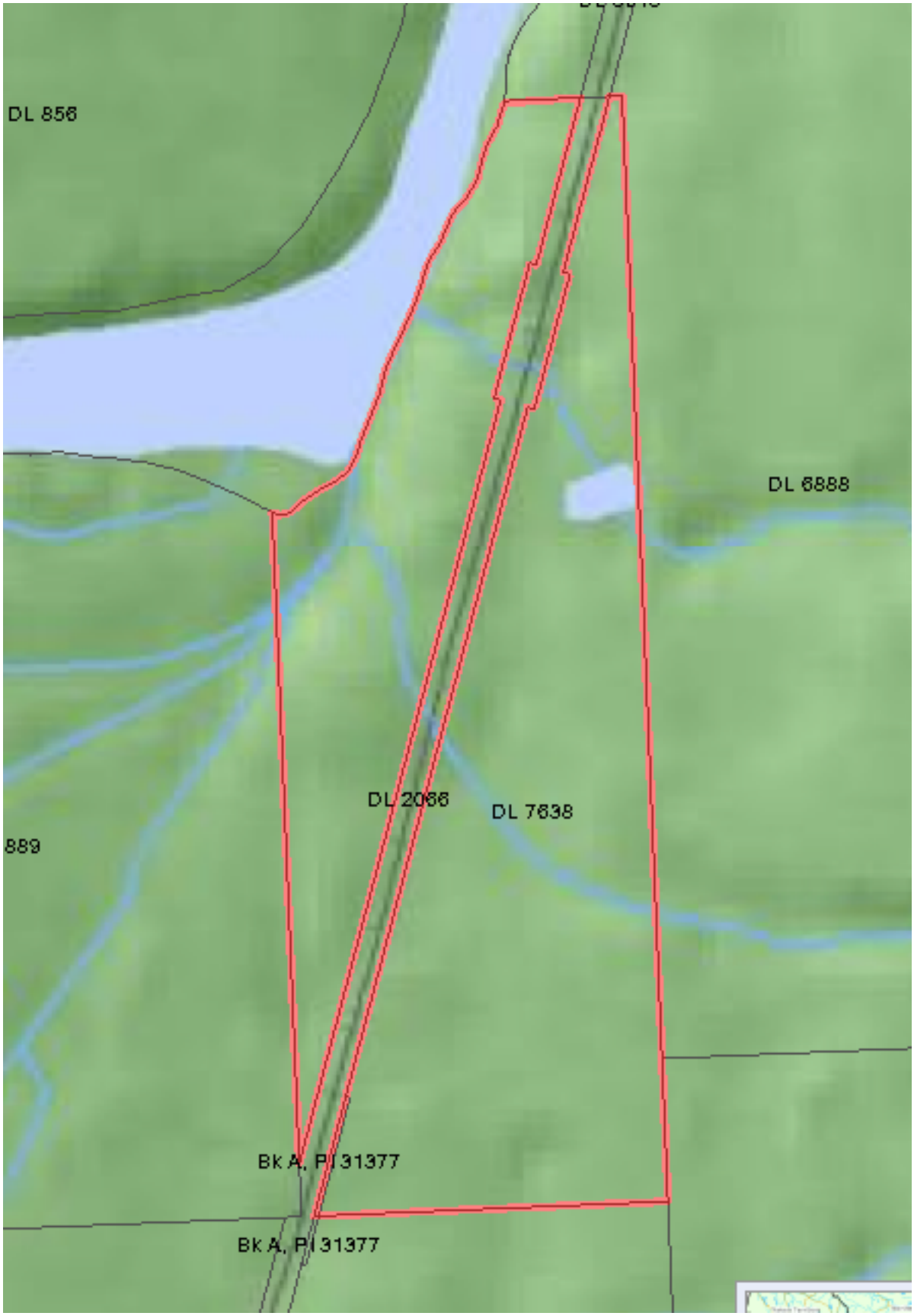
Remarquez que le polygone contenant un trou a une aire égale à l’aire du contour externe (un carré de 10 sur 10) moins l’aire du trou (un carré de 1 sur 1).

9.3.4 Les collections

Il y a quatre types de collections, qui regroupent ensemble plusieurs géométries simples.
– **MultiPoint**, une collection de points

– **MultiLineString**, une collection de lignes
– **MultiPolygon**, une collection de polygones
– **GeometryCollection**, une collection hétérogène de n'importe quelle géométrie (dont d'autre collections)

Les collections sont un concept présents dans les logiciels SIG plus que dans les applications de rendu graphique génériques. Elles sont utiles pour directement modéliser les objets du monde réel comme des objet spatiaux. Par exemple, comment modéliser une parcelle qui a été coupée par un chemin ? Comme un **MultiPolygon**, ayant une partie de chaque coté du chemin.



Notre collection exemple contient un polygone et un point :

```
SELECT name, ST_AsText(geom)
  FROM geometries
  WHERE name = 'Collection';

GEOMETRYCOLLECTION(POINT(2 0),POLYGON((0 0, 1 0, 1 1, 0 1, 0 0)))
```

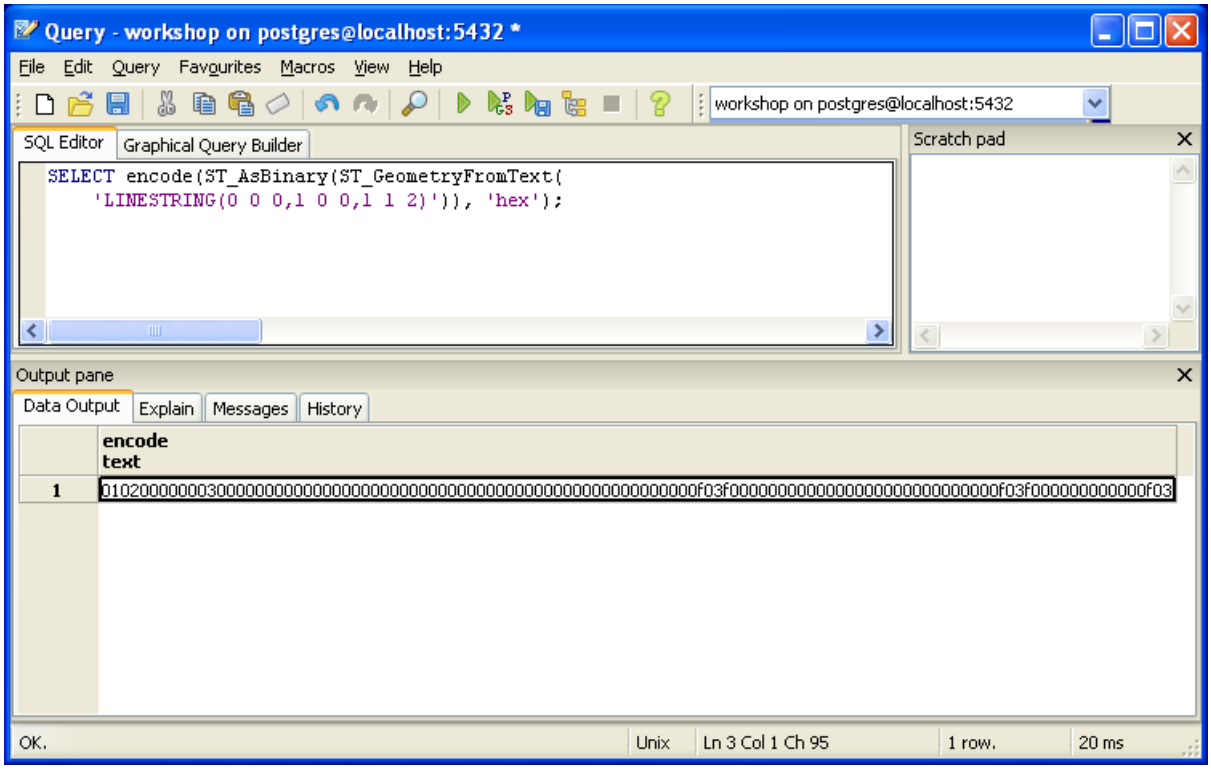
Certaines des fonctions spatiales spécifiques à la manipulation des collections sont :
- **ST_NumGeometries(geometry)** retourne le nombre de composantes d'une collection
- **ST_GeometryN(geometry,n)** retourne une composante spécifique
- **ST_Area(geometry)** retourne l'aire totale des composantes de type polygone
- **ST_Length(geometry)** retourne la longueur totale des composantes de type ligne

9.4 Entré / Sortie des géométries

Dans la base de données, les géométries sont stockées dans un format utilisé uniquement par le programme PostGIS. Afin que des programmes externes puissent insérer et récupérer les données utiles, elles ont besoin d'être converties dans un format compris par l'application. Heureusement, PostGIS supporte un grand nombre de formats en entrée et en sortie :
- Format texte bien connu (Well-known text *WKT*)
  - **ST_GeomFromText(text)** retourne une `geometry`
  - **ST_AsText(geometry)** retourne le `texte`
  - **ST_AsEWKT(geometry)** retourne le `texte`
- Format binaire bien connu (Well-known binary *WKB*)
  - **ST_GeomFromWKB(bytea)** retourne `geometry`
  - **ST_AsBinary(geometry)** retourne `bytea`
  - **ST_AsEWKB(geometry)** retourne `bytea`
- Geographic Mark-up Language (*GML*)
  - **ST_GeomFromGML(text)** retourne `geometry`
  - **ST_AsGML(geometry)** retourne `text`
- Keyhole Mark-up Language (*KML*)
  - **ST_GeomFromKML(text)** retourne `geometry`
  - **ST_AsKML(geometry)** retourne `text`
- *GeoJSON*
  - **ST_AsGeoJSON(geometry)** retourne `text`
- Scalable Vector Graphics (*SVG*)
  - **ST_AsSVG(geometry)** retourne `text`

La requête SQL suivante montre un exemple de représentation en *WKB* (l'appel à **encode()** est requis pour convertir le format binaire en ASCII pour l'afficher) :

```
SELECT encode(
  ST_AsBinary(ST_GeometryFromText('LINESTRING(0 0 0,1 0 0,1 1 2)')),
  'hex');
```

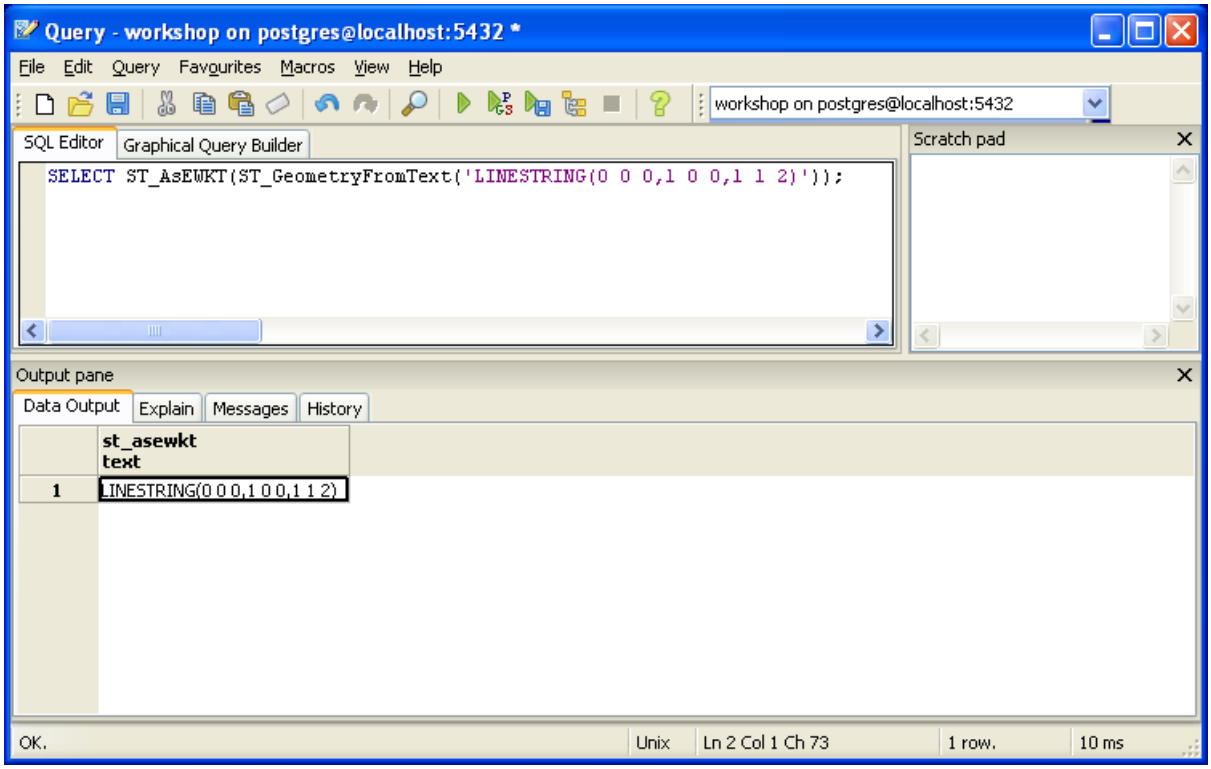


Dans le reste de ces travaux pratiques, nous utiliserons principalement le format WKT pour que vous puissiez lire et comprendre les géométries que nous voyons. Néanmoins, pour la plupart des traitement actuels, comme la visualisation des données dans une application SIG, le transfert de données à des services web, ou l'exécution distante de traitements, le format WKB est un format de choix.

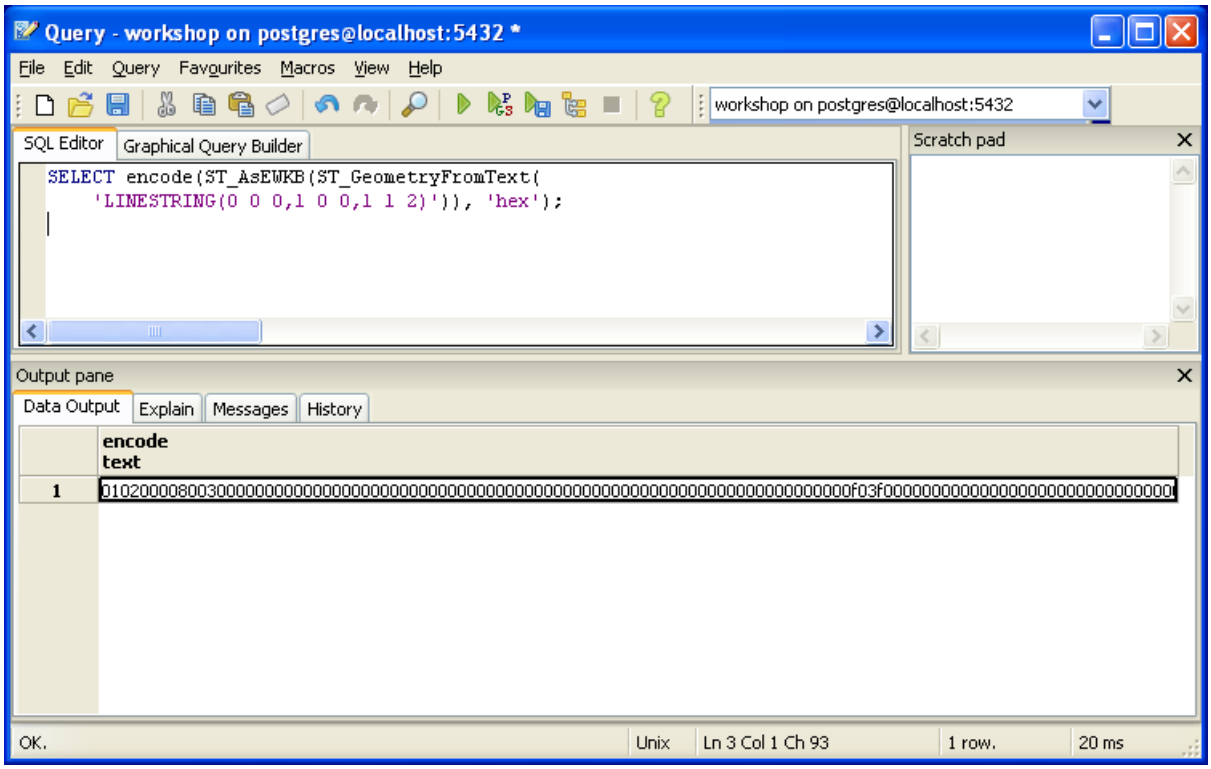
Puisque les formats WKT et le WKB sont définis dans la spécification *SFSQL*, ils ne prennent pas en compte les géométries à 3 ou 4 dimensions. C'est pour cette raison que PostGIS définit les formats Extended Well Known Text (EWKT) et Extended Well Known Binary (EWKB). Cela permet de gérer de façon similaire aux formats WKT et WKB les dimensions ajoutées.

Voici un exemple de ligne 3D au format WKT :

```
SELECT ST_AsEWKT(ST_GeometryFromText('LINESTRING(0 0 0,1 0 0,1 1 2)'));
```

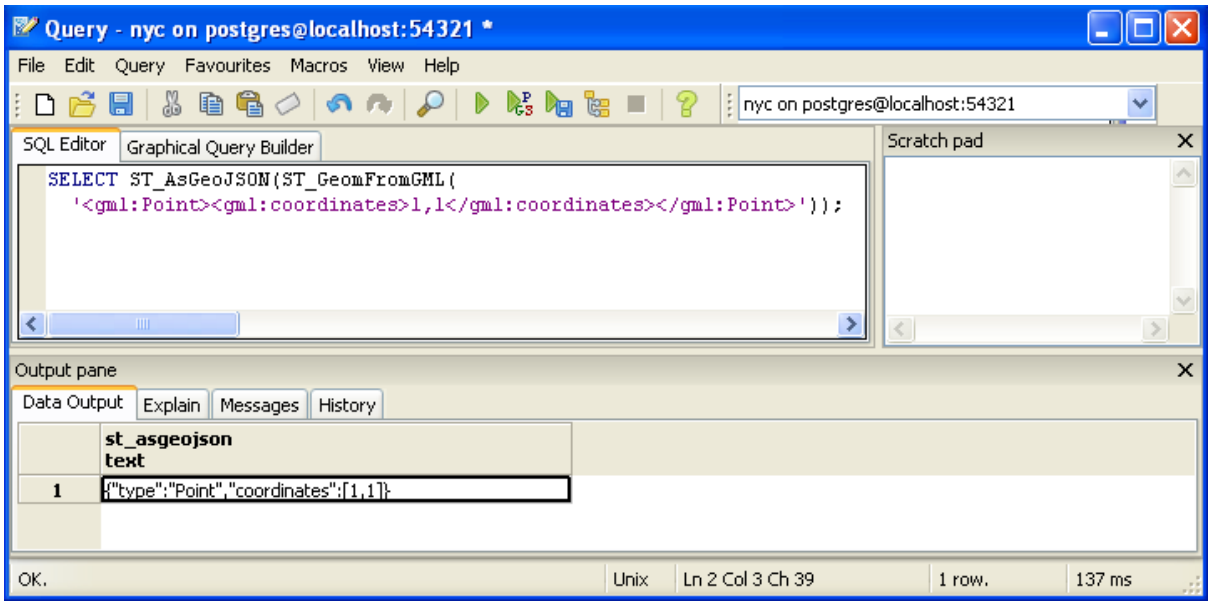



```
SELECT encode(ST_AsEWKB(ST_GeometryFromText(
    'LINESTRING(0 0 0,1 0 0,1 1 2)')), 'hex');
```



En plus de pouvoir générer les différents formats en sortie (WKT, WKB, GML, KML, JSON, SVG), PostGIS permet aussi de lire 4 de ces formats (WKT, WKB, GML, KML). La plupart des applications utilisent des fonctions créant des géométries à l'aide du format WKT ou WKB, mais les autres marchent aussi. Voici un exemple qui lit du GML et retourne du JSON :

```
SELECT ST_AsGeoJSON(ST_GeomFromGML('<gml:Point><gml:coordinates>1,1</gml:coordinates></g
```



9.5 Liste des fonctions

Populate_Geometry_Columns : s'assure que les colonnes géométriques ont les contraintes spatiales appropriées et qu'elles sont présentes dans la table geometry_columns.

ST_Area : retourne l'aire de la surface si c'est un polygone ou un multi-polygone. Pour le type "geometry" l'aire est dans l'unité du SRID. Pour les "geography" l'aire est en mètres carrés.

ST_AsText : retourne la représentation de la geometry/geography au format Well-Known Text (WKT) sans métadonnée correspondant au SRID.

ST_AsBinary : retourne la représentation de la geometry/geography au format Well-Known Binary (WKB) sans métadonnée correspondant u SRID.

ST_EndPoint : retourne le dernier point d'une ligne.

ST_AsEWKB : retourne la représentation de la géometrie au format Well-Known Binary (WKB) avec la métadonnée SRID.

ST_AsEWKT : retourne la représentation de la géometrie au format Well-Known Text (WKT) avec la métadonnée SRID.

ST_AsGeoJSON : retourne la géométrie au format GeoJSON.

ST_AsGML : retourne la géométrie au format GML version 2 ou 3.

ST_AsKML : retourne la géométrie au format KML. Nombreuses variantes. Par défaut : version=2 et precision=15.

ST_AsSVG : retourne la géométrie au format SVG.

ST_ExteriorRing : retourne une ligne représentant le contour extérieur du polygone. Retourne NULL si la géométrie n'est pas un polygone. Ne fonctionne pas avec les multi-polygones.

ST_GeometryN : retourne la nième composante si la géométrie est du type GEOMETRYCOLLECTION, MULTIPOINT, MULTILINESTRING, MULTICURVE ou MULTIPOLYGON. Sinon, retourne NULL.

ST_GeomFromGML : prend en entrée une représentation GML de la géométrie et retourne un object PostGIS de type geometry.

ST_GeomFromKML : prend en entrée une représentation KML de la géométrie et retourne un object PostGIS de type geometry.

ST_GeomFromText : retourne une valeur de type ST_Geometry à partir d'une représentation au format Well-Known Text (WKT).

ST_GeomFromWKB : retourne une valeur de type ST_Geometry à partir d'une représentation au format Well-Known Binary (WKB).

ST_GeometryType : retourne le type de géométrie de la valeur de type ST_Geometry.

ST_InteriorRingN : retourne le nième contour intérieur d'un polygone. Retourne NULL si la géométrie n'est pas un polygone ou si N est hors des limites.

ST_Length : retourne la longueur en 2-dimensions si c'est une ligne ou une multi-lignes. Les objets de type geometry sont dans l'unité du système de référence spatiale et les objets de type geography sont en mètres (sphéroïde par défaut).

ST_NDims : retourne le nombre de dimensions d'une géométrie. Les valeurs possibles sont : 2,3 ou 4.

ST_NPoints : retourne le nombre de points dans une géométrie.

ST_NRings : si la géométrie est un polygone ou un multi-polygone, retourne le nombre de contours.

ST_NumGeometries : si la géométrie est du type GEOMETRYCOLLECTION (ou MULTI*) retourne le nombre de géométries, sinon retourne NULL.

ST_Perimeter : retourne la longueur du contour extérieur d'une valeur de type ST_Surface ou ST_MultiSurface (polygone, multi-polygone).

ST_SRID : retourne l'identifiant du système de référence spatiale définit dans la table spatial_ref_sys d'un objet de type ST_Geometry.

ST_StartPoint : retourne le premier point d'une ligne.

ST_X : retourne la coordonnée X d'un point, ou NULL si non présent. L'argument passé doit être un point.

ST_Y : retourne la coordonnée Y d'un point, ou NULL si non présent. L'argument passé doit être un point.

CHAPITRE 10

Partie 9 : Exercices sur les géométries

Voici un petit rappel de toutes les fonction que nous avons abordé jusqu'à présent. Elles devraient être utiles pour les exercices !

- **sum(expression)** agrégation retournant la somme d'un ensemble
- **count(expression)** agrégation retournant le nombre d'éléments d'un ensemble
- **ST_GeometryType(geometry)** retourne le type de la géométrie
- **ST_NDims(geometry)** retourne le nombre de dimensions
- **ST_SRID(geometry)** retourne l'identifiant du système de référence spatiale
- **ST_X(point)** retourne la coordonnée X
- **ST_Y(point)** retourne la coordonnée Y
- **ST_Length(linestring)** retourne la longueur d'une ligne
- **ST_StartPoint(geometry)** retourne le premier point d'une ligne
- **ST_EndPoint(geometry)** retourne le dernier point d'une ligne
- **ST_NPoints(geometry)** retourne le nombre de points d'une ligne
- **ST_Area(geometry)** retourne l'aire d'un polygone
- **ST_NRings(geometry)** retourne le nombre de contours (1 ou plus si il y a des trous)
- **ST_ExteriorRing(polygon)** retourne le contour extérieur (ligne) d'un polygone
- **ST_InteriorRingN(polygon, integer)** retourne le contour intérieur (ligne) d'un polygone
- **ST_Perimeter(geometry)** retourne la longueur de tous les contours
- **ST_NumGeometries(multi/geomcollection)** retourne le nombre de composantes dans une collection
- **ST_GeometryN(geometry, integer)** retourne la nième entité de la collection
- **ST_GeomFromText(text)** retourne `geometry`
- **ST_AsText(geometry)** retourne WKT `text`
- **ST_AsEWKT(geometry)** retourne EWKT `text`
- **ST_GeomFromWKB(bytea)** retourne `geometry`
- **ST_AsBinary(geometry)** retourne WKB `bytea`
- **ST_AsEWKB(geometry)** retourne EWKB `bytea`
- **ST_GeomFromGML(text)** retourne `geometry`
- **ST_AsGML(geometry)** retourne GML `text`
- **ST_GeomFromKML(text)** retourne `geometry`
- **ST_AsKML(geometry)** retourne KML `text`
- **ST_AsGeoJSON(geometry)** retourne JSON `text`
- **ST_AsSVG(geometry)** retourne SVG `text`

Souvenez-vous aussi des tables disponibles :

- `nyc_census_blocks`
  - name, popn_total, boroname, the_geom

- `nyc_streets`
  - name, type, the_geom
- `nyc_subway_stations`
  - name, the_geom
- `nyc_neighborhoods`
  - name, boroname, the_geom

10.1 Exercices

- **"Quelle est l'aire du quartier 'West Village' ?"**

```
SELECT ST_Area(the_geom)
  FROM nyc_neighborhoods
  WHERE name = 'West Village';
```

```
1044614.53027344
```

Note : L'aire est donnée en mètres carrés. Pour obtenir l'aire en hectare, divisez par 10000. Pour obtenir l'aire en acres, divisez par 4047.

- **"Quelle est l'aire de Manhattan en acres ?"** (Astuce : `nyc_census_blocks` et `nyc_neighborhoods` ont toutes les deux le champ `boroname`.)

```
SELECT Sum(ST_Area(the_geom)) / 4047
  FROM nyc_neighborhoods
  WHERE boroname = 'Manhattan';
```

```
13965.3201224118
```

or...

```
SELECT Sum(ST_Area(the_geom)) / 4047
  FROM nyc_census_blocks
  WHERE boroname = 'Manhattan';
```

```
14572.1575543757
```

- **"Combien de blocs de la ville de New York ont des trous ?"**

```
SELECT Count(*)
  FROM nyc_census_blocks
  WHERE ST_NRings(the_geom) > 1;
```

```
66
```

- **"Quel est la longueur totale des rues (en kilomètres) dans la ville de New York ?"** (Astuce : l'unité de mesure des données spatiales est le mètre, il y a 1000 mètres dans un kilomètre.)

```
SELECT Sum(ST_Length(the_geom)) / 1000
  FROM nyc_streets;
```

```
10418.9047172
```

- **"Quelle est la longueur de 'Columbus Cir' (Columbus Circle) ?"**

```
SELECT ST_Length(the_geom)
  FROM nyc_streets
  WHERE name = 'Columbus Cir';
```

```
308.34199
```

– **"Quelle est le contour de 'West Village' au format JSON ?"**

```
SELECT ST_AsGeoJSON(the_geom)
  FROM nyc_neighborhoods
  WHERE name = 'West Village';
```

```
{"type":"MultiPolygon","coordinates":
 [[[[583263.2776595836,4509242.6260239873],
    [583276.81990686338,4509378.825446927], ...
    [583263.2776595836,4509242.6260239873]]]]}
```

La géométrie de type "MultiPolygon", intéressant !

– **"Combien de polygones sont dans le multi-polygone 'West Village' ?"**

```
SELECT ST_NumGeometries(the_geom)
  FROM nyc_neighborhoods
  WHERE name = 'West Village';
```

```
1
```

Note : Il n'est pas rare de trouver des éléments de type multi-polygone ne contenant qu'un seul polygone dans des tables. L'utilisation du type multi-polygone permet d'utiliser une seule table pour y stocker des géométries simples et multiples sans mélanger les types.

– **"Quel est la longueur des rues de la ville de New York, suivant leur type ?"**

```
SELECT type, Sum(ST_Length(the_geom)) AS length
 FROM nyc_streets
 GROUP BY type
 ORDER BY length DESC;
```

```
                      type                        |      length
--------------------------------------------------+------------------
 residential                                      | 8629870.33786606
 motorway                                         | 403622.478126363
 tertiary                                         | 360394.879051303
 motorway_link                                    | 294261.419479668
 secondary                                        | 276264.303897926
 unclassified                                     | 166936.371604458
 primary                                          | 135034.233017947
 footway                                          | 71798.4878378096
 service                                          |  28337.635038596
 trunk                                            | 20353.5819826076
 cycleway                                         | 8863.75144825929
 pedestrian                                       | 4867.05032825026
 construction                                     | 4803.08162103562
 residential; motorway_link                       | 3661.57506293745
 trunk_link                                       | 3202.18981240201
 primary_link                                     | 2492.57457083536
 living_street                                    | 1894.63905457332
```

```
primary; residential; motorway_link; residential | 1367.76576941335
undefined                                        |  380.53861910346
steps                                            | 282.745221342127
motorway_link; residential                       |  215.07778911517
```

Note : La clause `ORDER BY length DESC` ordonne le résultat par la valeur des longueurs dans l'ordre décroissant. Le résultat avec la plus grande valeur se retrouve au début la liste de résultats.

CHAPITRE 11

Partie 10 : Les relations spatiales

Jusqu'à présent, nous avons utilisé uniquement des fonctions qui permettent de mesurer (**ST_Area**, **ST_Length**), de sérialiser (**ST_GeomFromText**) ou désérialiser (**ST_AsGML**) des géométries. Ces fonctions sont toutes utilisées sur une géométrie à la fois.

Les base de données spatiales sont puissantes car elle ne se contentent pas de stocker les géométries, elle peuvent aussi vérifier les *relations entre les géométries*.

Pour les questions comme "Quel est le plus proche garage à vélo près du parc ?" ou "Ou est l'intersection du métro avec telle rue ?", nous devrons comparer les géométries représentant les garages à vélo, les rues et les lignes de métro.

Le standard de l'OGC définit l'ensemble de fonctions suivantes pour comparer les géométries.

11.1 ST_Equals

ST_Equals(geometry A, geometry B) teste l'égalité spatiale de deux géométries.

ST_Equals retourne TRUE si les deux géométries sont du même type et ont des coordonnées x.y identiques.

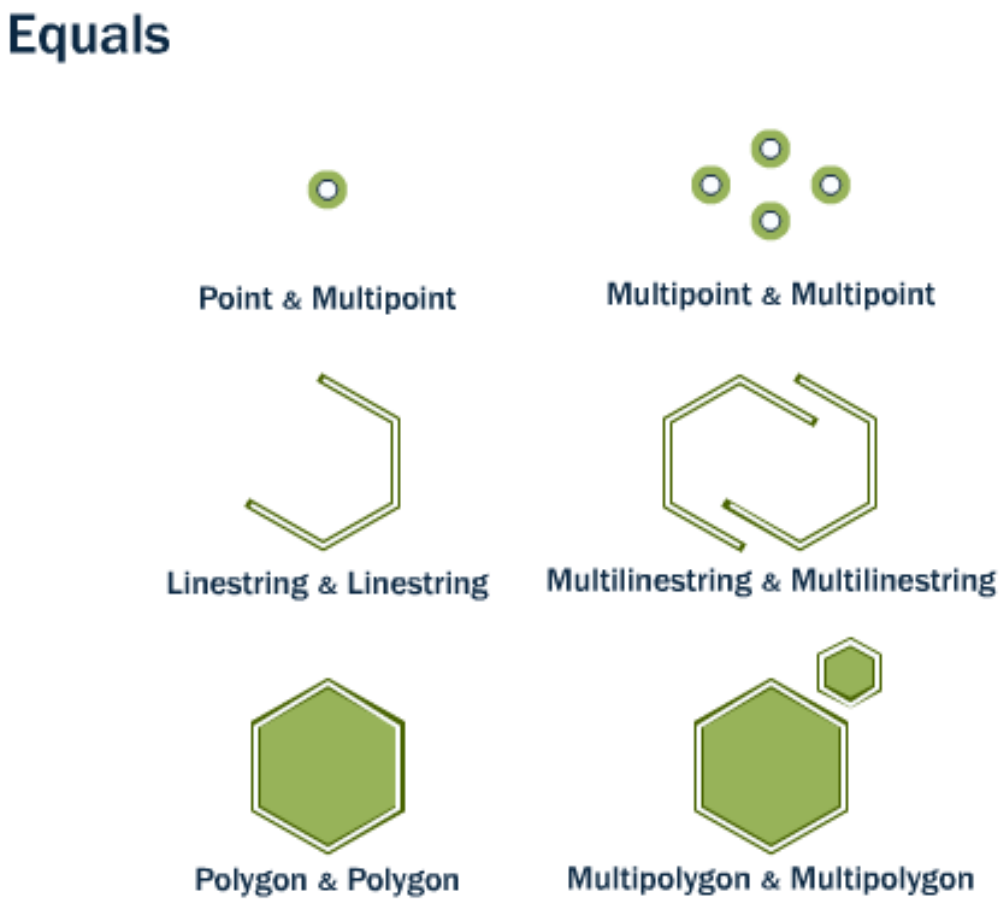
Premièrement, essayons de récupérer la représentation d'un point de notre table `nyc_subway_stations`. Nous ne prendrons que l'entrée : 'Broad St'.

```
SELECT name, the_geom, ST_AsText(the_geom)
FROM nyc_subway_stations
WHERE name = 'Broad St';
```

```
   name    |                      the_geom                      |       st_astext
----------+----------------------------------------------------+-----------------------
 Broad St | 0101000020266900000EEBD4CF27CF2141BC17D69516315141 | POINT(583571 4506714)
```

Maintenant, copiez / collez la valeur affichée pour tester la fonction **ST_Equals** :

```
SELECT name
FROM nyc_subway_stations
WHERE ST_Equals(the_geom, '0101000020266900000EEBD4CF27CF2141BC17D69516315141');
```

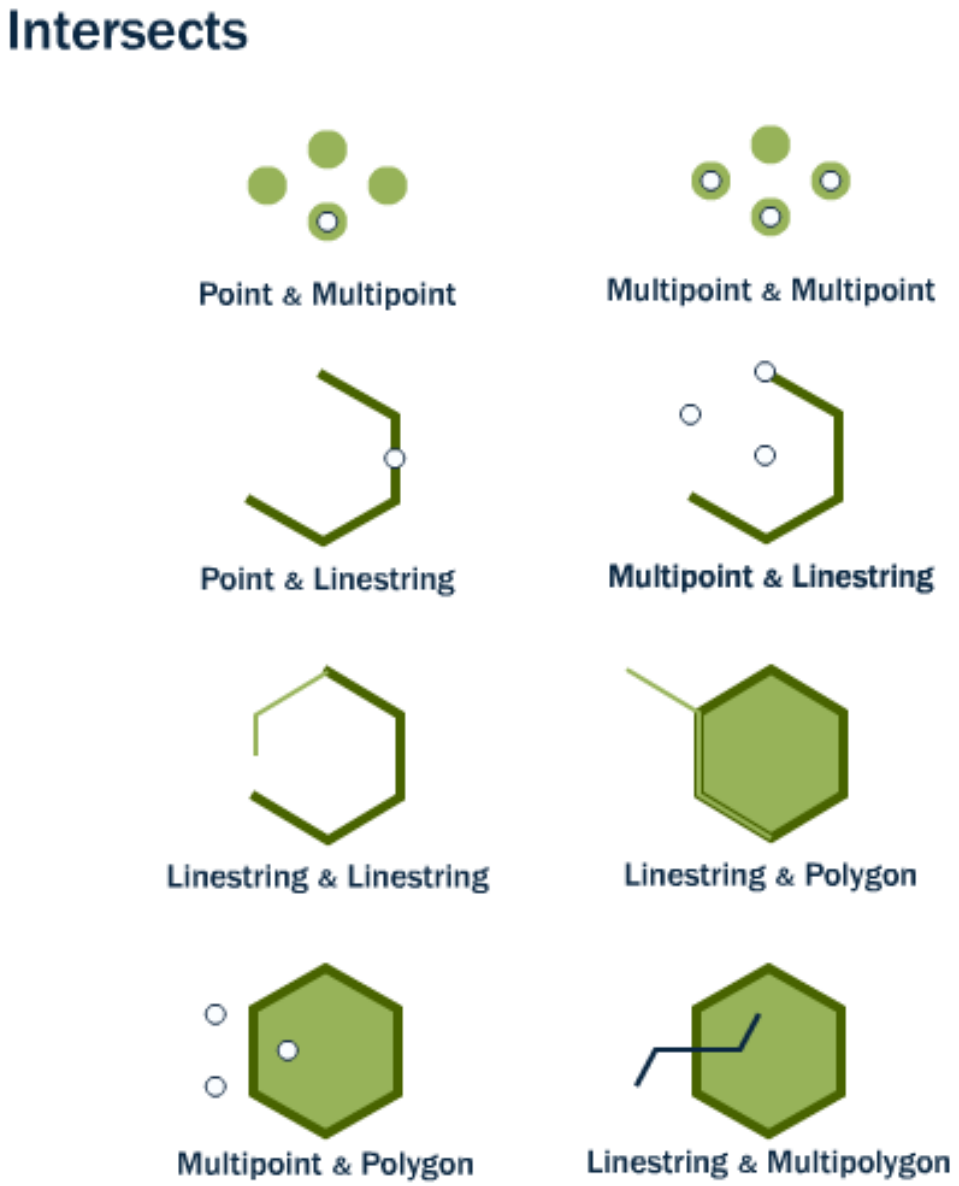

Equals
Point & Multipoint
Multipoint & Multipoint
Linestring & Linestring
Multilinestring & Multilinestring
Polygon & Polygon
Multipolygon & Multipolygon

```
Broad St
```

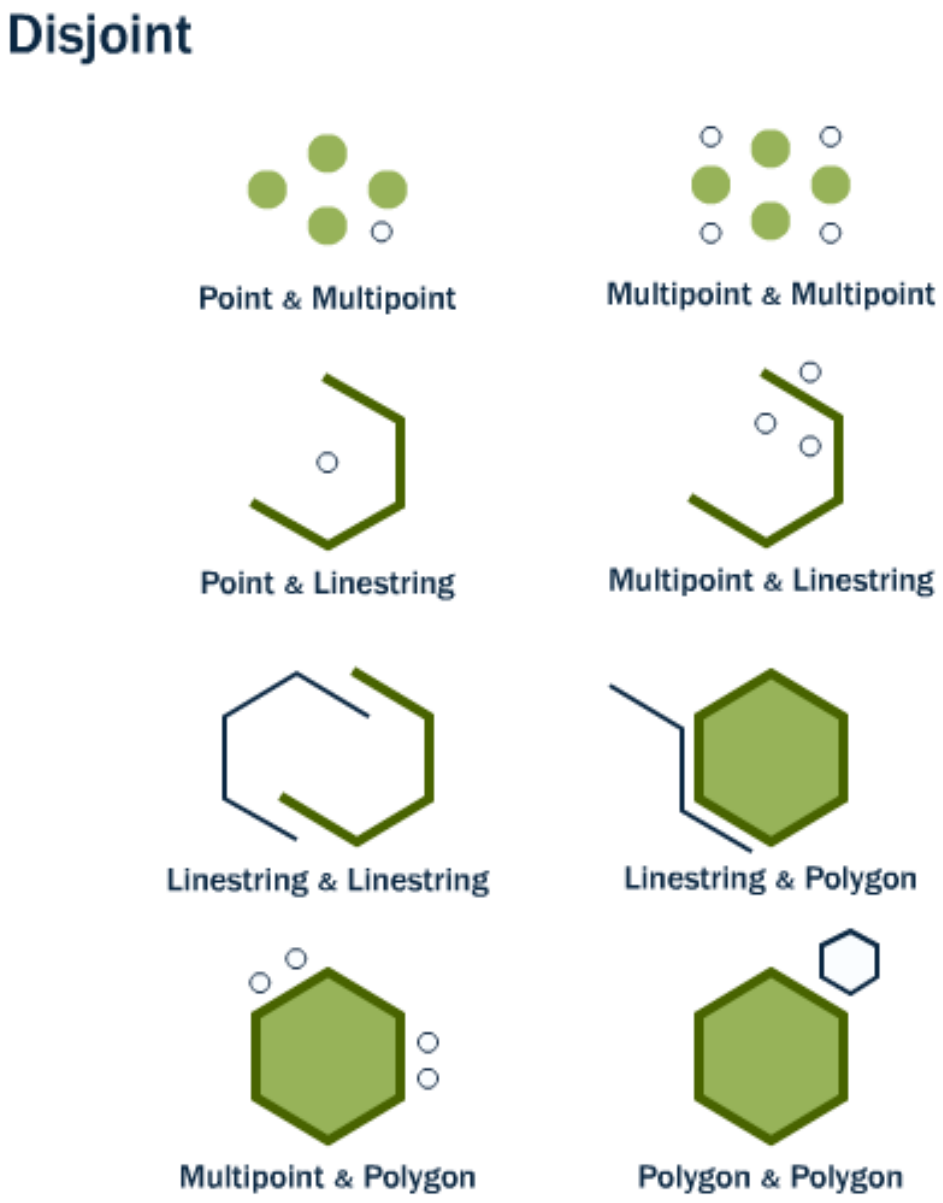
Note : La représentation du point n'est pas vraiment compréhensible (`0101000020266900000EEBD4CF27CF2141BC17D69516315141`) mais c'est exactement la représentation des coordonnées. Pour tester l'égalité, l'utilisation de ce format est nécessaire.

11.2 ST_Intersects, ST_Disjoint, ST_Crosses et ST_Overlaps

ST_Intersects, **ST_Crosses**, et **ST_Overlaps** teste si l'intérieur des géométries s'intersecte, se croise ou se chevauche.



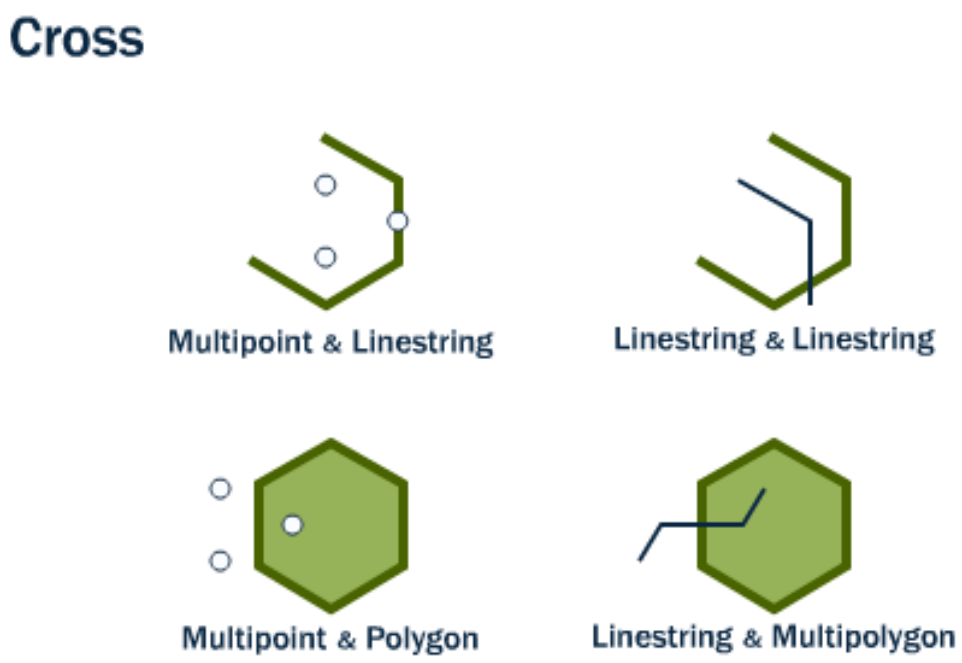
ST_Intersects(geometry A, geometry B) retourne t (TRUE) si l'intersection ne renvoie pas un ensemble vide de résultats. Intersects retourne le résultat exactement inverse de la fonction disjoint.



L'opposé de ST_Intersects est **ST_Disjoint(geometry A , geometry B)**. Si deux géométries sont disjointes, elle ne s'intersectent pas et vice-versa. En fait, il est souvent plus efficace de tester si deux géométries ne s'intersectent pas que de tester si elles sont disjointes du fait que le test d'intersection peut être spatialement indexé alors que le test disjoint ne le peut pas.

Pour les comparaisons de couples de types multipoint/polygon, multipoint/linestring, linestring/linestring, linestring/polygon, et linestring/multipolygon, **ST_Crosses(geometry A, geometry B)** retourne t (TRUE) si les résultats de l'intersection sont à l'intérieur des deux géométries.

ST_Overlaps(geometry A, geometry B) compare deux géométries de même dimension et retourne TRUE si leur intersection est une géométrie différente des deux fournies mais de même dimension.



Essayons de prendre la station de métro de Broad Street et de déterminer son voisinage en utilisant la fonction **ST_Intersects** :

```
SELECT name, boroname
FROM nyc_neighborhoods
WHERE ST_Intersects(the_geom, '0101000020266900000EEBD4CF27CF2141BC17D69516315141');
```

```
        name        | boroname
--------------------+-----------
 Financial District | Manhattan
```

11.3 ST_Touches

ST_Touches teste si deux géométries se touchent en leur contours extérieurs, mais leur contours intérieurs ne s'intersectent pas

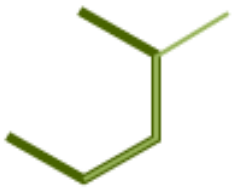
ST_Touches(geometry A, geometry B) retourne TRUE soit si les contours des géométries s'intersectent ou si l'un des contours intérieurs de l'une intersecte le contour extérieur de l'autre.

11.4 ST_Within et ST_Contains

ST_Within et **ST_Contains** teste si une géométrie est totalement incluse dans l'autre.

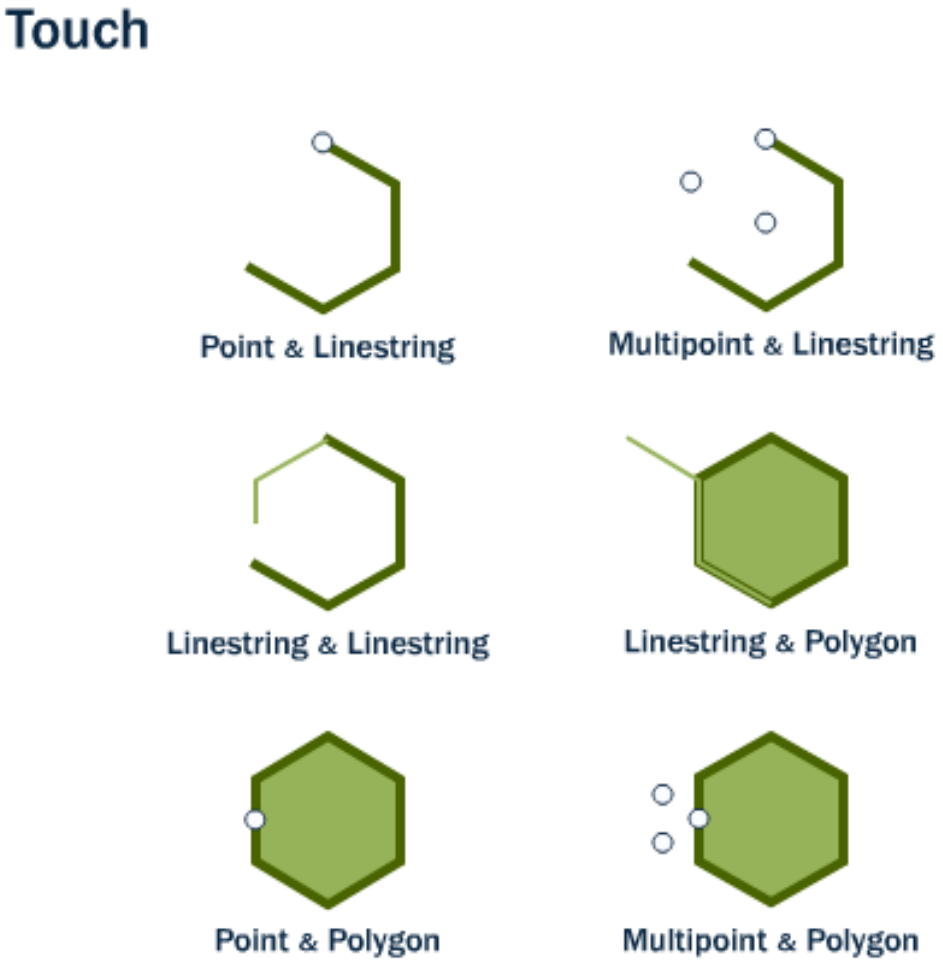
Overlap

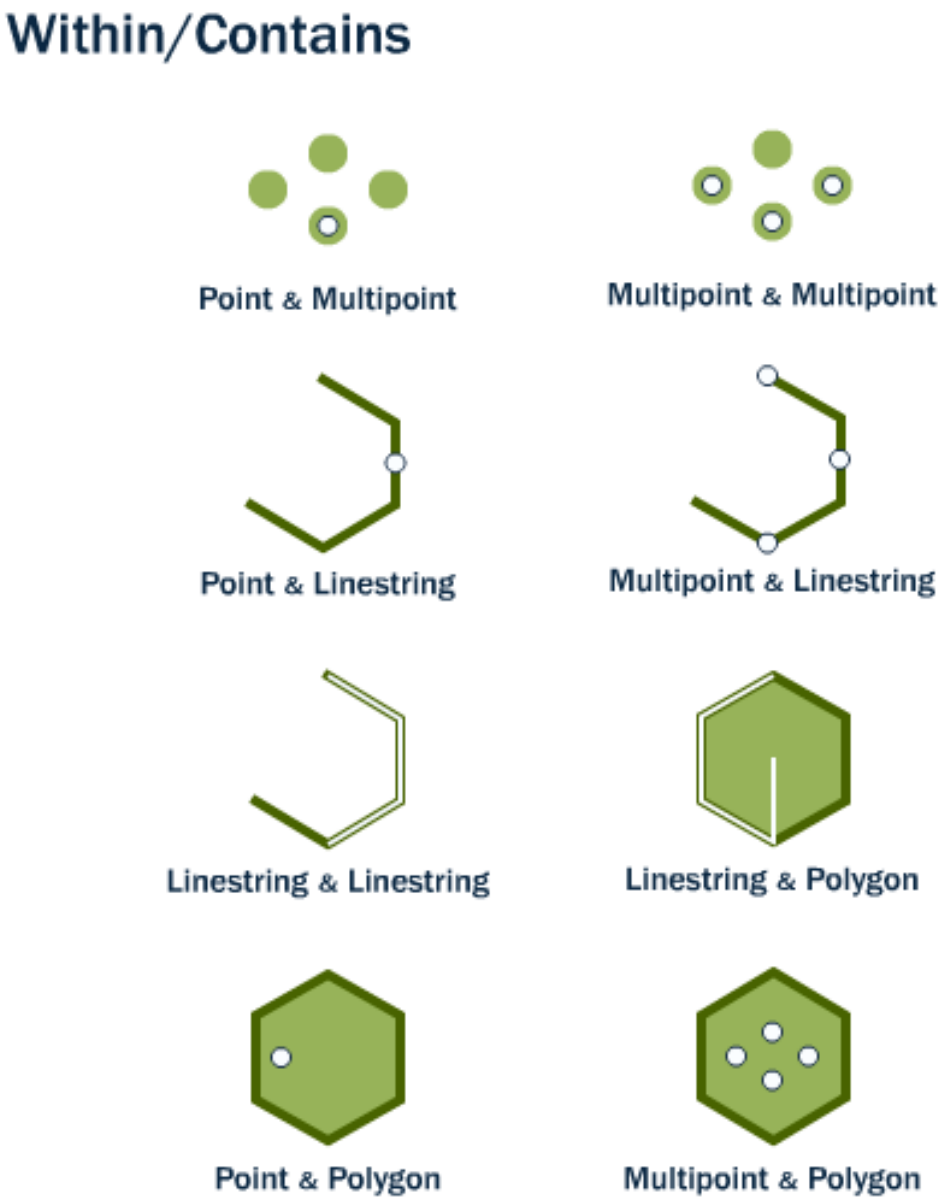
Multipoint & Multipoint



Linestring & Linestring

Polygon & Polygon


Touch
Point & Linestring
Multipoint & Linestring
Linestring & Linestring
Linestring & Polygon
Point & Polygon
Multipoint & Polygon


Within/Contains
Point & Multipoint
Multipoint & Multipoint
Point & Linestring
Multipoint & Linestring
Linestring & Linestring
Linestring & Polygon
Point & Polygon
Multipoint & Polygon

ST_Within(geometry A , geometry B) retourne TRUE si la première géométrie est complètement contenue dans l'autre. ST_Within teste l'exact opposé au résultat de ST_Contains.

ST_Contains(geometry A, geometry B) retourne TRUE si la seconde géométrie est complètement contenue dans la première géométrie.

11.5 ST_Distance et ST_DWithin

Une question fréquente dans le domaine du SIG est "trouver tous les éléments qui se trouvent à une distance X de cet autre élément".

La fonction **ST_Distance(geometry A, geometry B)** calcule la *plus courte* distance entre deux géométries. Cela est pratique pour récupérer la distance entre les objets.

```
SELECT ST_Distance(
  ST_GeometryFromText('POINT(0 5)'),
  ST_GeometryFromText('LINESTRING(-2 2, 2 2)'));
```

```
3
```

Pour tester si deux objets sont à la même distance d'un autre, la fonction **ST_DWithin** fournit un test tirant profit des index. Cela est très utile pour répondre a une question telle que : "Combien d'arbres se situent dans un buffer de 500 mètres autour de cette route ?". Vous n'avez pas à calculer le buffer, vous avez simplement besoin de tester la distance entre les géométries.

En utilisant de nouveau notre station de métro Broad Street, nous pouvons trouver les rues voisines (à 10 mètres de) de la station :

```
SELECT name
FROM nyc_streets
WHERE ST_DWithin(
        the_geom,
        '0101000020266900000EEBD4CF27CF2141BC17D69516315141',
        10
      );

     name
--------------
   Wall St
   Broad St
   Nassau St
```

Nous pouvons vérifier la réponse sur une carte. La station Broad St est actuellement à l'intersection des rues Wall, Broad et Nassau.

11.6 Liste des fonctions

ST_Contains(geometry A, geometry B) : retourne TRUE si aucun des points de B n'est à l'extérieur de

A, et au moins un point de l'intérieur de B est à l'intérieur de A.

ST_Crosses(geometry A, geometry B) : retourne TRUE si la géométrie A a certains, mais pas la totalité, de ses points à l'intérieur de B.

ST_Disjoint(geometry A , geometry B) : retourne TRUE si les géométries ne s'intersectent pas - elles n'ont aucun point en commun.

ST_Distance(geometry A, geometry B) : retourne la distance cartésienne en 2 dimensions minimum entre deux géométries dans l'unité de la projection.

ST_DWithin(geometry A, geometry B, radius) : retourne TRUE si les géométries sont distante (radius) l'une de l'autre.

ST_Equals(geometry A, geometry B) : retourne TRUE si les géométries fournies représentent la même géométrie. L'ordre des entités n'est pas pris en compte.

ST_Intersects(geometry A, geometry B) : retourne TRUE si les géométries s'intersectent - (ont un espace en commun) et FALSE si elles n'en ont pas (elles sont disjointes).

ST_Overlaps(geometry A, geometry B) : retourne TRUE si les géométries ont un espace en commun, sont de la même dimension, mais ne sont pas complètement contenues l'une dans l'autre.

ST_Touches(geometry A, geometry B) : retourne TRUE si les géométries ont au moins un point en commun, mais leur intérieurs ne s'intersectent pas.

ST_Within(geometry A , geometry B) : retourne TRUE si la géométrie A est complètement à l'intérieur de B

CHAPITRE 12

Partie 11 : Exercices sur les relations spatiales

Voici un rappel des fonctions que nous avons vu dans les parties précédentes. Elles seront utiles pour les exercices !

- **sum(expression)** agrégation retournant la somme d'un ensemble
- **count(expression)** agrégation retournant le nombre d'éléments d'un ensemble
- **ST_Contains(geometry A, geometry B)** retourne TRUE si la géométrie A contient la géométrie B
- **ST_Crosses(geometry A, geometry B)** retourne TRUE si la géométrie A croise la géométrie B
- **ST_Disjoint(geometry A , geometry B)** retourne TRUE si les géométries ne s'intersectent pas
- **ST_Distance(geometry A, geometry B)** retourne la distance minimum entre deux géométries
- **ST_DWithin(geometry A, geometry B, radius)** retourne TRUE si la A est distante d'au plus radius de B
- **ST_Equals(geometry A, geometry B)** retourne TRUE si A est la même géométrie que B
- **ST_Intersects(geometry A, geometry B)** retourne TRUE si A intersecte B
- **ST_Overlaps(geometry A, geometry B)** retourne TRUE si A et B on un espace en commun, mais ne sont pas complètement incluses l'un dans l'autre.
- **ST_Touches(geometry A, geometry B)** retourne TRUE si le contour extérieur de A touche B
- **ST_Within(geometry A, geometry B)** retourne TRUE si A est hors de B

Souvenez-vous les tables à votre disposition :

- `nyc_census_blocks`
  - name, popn_total, boroname, the_geom
- `nyc_streets`
  - name, type, the_geom
- `nyc_subway_stations`
  - name, the_geom
- `nyc_neighborhoods`
  - name, boroname, the_geom

12.1 Exercices

- **"Quelle est la valeur géométrique de la rue nommée 'Atlantic Commons' ?"**

```
SELECT the_geom
  FROM nyc_streets
  WHERE name = 'Atlantic Commons';

0105000020266900000100000001020000000200000093235673BE82141F319CD89A22E514170E30E0ADF
```

– **"Quels sont les quartiers et villes qui sont dans Atlantic Commons ?"**

```
SELECT name, boroname
FROM nyc_neighborhoods
WHERE ST_Intersects(
  the_geom,
  '0105000020266900000100000001020000000200000093235673BE82141F319CD89A22E514170E30E0
);

    name     | boroname
------------+----------
 Fort Green | Brooklyn
```

– **"Quelles rues touchent Atlantic Commons ?"**

```
SELECT name
FROM nyc_streets
WHERE ST_Touches(
  the_geom,
  '0105000020266900000100000001020000000200000093235673BE82141F319CD89A22E514170E30E0
);

     name
---------------
 S Oxford St
 Cumberland St
```

– **"Approximativement combien de personnes vivent dans (ou dans une zone de 50 mètres autour d') Atlantic Commons ?"**

```
SELECT Sum(popn_total)
  FROM nyc_census_blocks
  WHERE ST_DWithin(
   the_geom,
   '0105000020266900000100000001020000000200000009323567 3BE82141F319CD89A22E514170E30E
   50
   );
```

```
1186
```

CHAPITRE 13

Partie 12 : Les jointures spatiales

Les jointures spatiales sont la cerise sur le gâteau des base de données spatiales. Elles vous pemettent de combiner les informations de plusieurs tables en utilisant une relation spatiale comme clause de jointure. La plupart des "analyses SIG standards" peuvent être exprimées à l'aide de jointures spatiales.

Dans la partie précédente, nous avons utilisé les relations spatiales en utilisant deux étapes dans nos requêtes : nous avons dans un premier temps extrait la station de métro "Broad St" puis nous avons utilisé ce résultat dans nos autres requêtes pour répondre aux questions comme "dans quel quartier se situe la station 'Broad St' ?"

En utilisant les jointures spatiales, nous pouvons répondre aux questions en une seule étape, récupérant les informations relatives à la station de métro et le quartier la contenant :

```
SELECT
  subways.name AS subway_name,
  neighborhoods.name AS neighborhood_name,
  neighborhoods.boroname AS borough
FROM nyc_neighborhoods AS neighborhoods
JOIN nyc_subway_stations AS subways
ON ST_Contains(neighborhoods.the_geom, subways.the_geom)
WHERE subways.name = 'Broad St';
```

```
 subway_name | neighborhood_name   |  borough
-------------+---------------------+-----------
 Broad St    | Financial District | Manhattan
```

Nous avons pu regrouper chaque station de métro avec le quartier auquel elle appartient, mais dans ce cas nous n'en voulions qu'une. Chaque fonction qui envoit un résultat du type vrai/faux peut être utilisée pour joindre spatialement deux tables, mais la plupart du temps on utilise : **ST_Intersects**, **ST_Contains**, et **ST_DWithin**.

13.1 Jointure et regroupement

La combinaison de `JOIN` avec `GROUP BY` fournit le type d'analyse qui est couramment utilisé dans les systèmes SIG.

Par exemple : **Quelle est la population et la répartition raciale du quartier de Manhattan ?** Ici nous avons une question qui combine les informations relatives à la population recensée et les contours des quartiers, or nous ne voulons qu'un seul quartier, celui de Manhattan.

```
SELECT
  neighborhoods.name AS neighborhood_name,
  Sum(census.popn_total) AS population,
  Round(100.0 * Sum(census.popn_white) / Sum(census.popn_total),1) AS white_pct,
  Round(100.0 * Sum(census.popn_black) / Sum(census.popn_total),1) AS black_pct
FROM nyc_neighborhoods AS neighborhoods
JOIN nyc_census_blocks AS census
ON ST_Intersects(neighborhoods.the_geom, census.the_geom)
WHERE neighborhoods.boroname = 'Manhattan'
GROUP BY neighborhoods.name
ORDER BY white_pct DESC;
```

```
  neighborhood_name   | population | white_pct | black_pct
----------------------+------------+-----------+-----------
 Carnegie Hill        |      19909 |      91.6 |       1.5
 North Sutton Area    |      21413 |      90.3 |       1.2
 West Village         |      27141 |      88.1 |       2.7
 Upper East Side      |     201301 |      87.8 |       2.5
 Greenwich Village    |      57047 |      84.1 |       3.3
 Soho                 |      15371 |      84.1 |       3.3
 Murray Hill          |      27669 |      79.2 |       2.3
 Gramercy             |      97264 |      77.8 |       5.6
 Central Park         |      49284 |      77.8 |      10.4
 Tribeca              |      13601 |      77.2 |       5.5
 Midtown              |      70412 |      75.9 |       5.1
 Chelsea              |      51773 |      74.7 |       7.4
 Battery Park         |       9928 |      74.1 |       4.9
 Upper West Side      |     212499 |      73.3 |      10.4
 Financial District   |      17279 |      71.3 |       5.3
 Clinton              |      26347 |      64.6 |      10.3
 East Village         |      77448 |      61.4 |       9.7
 Garment District     |       6900 |      51.1 |       8.6
 Morningside Heights  |      41499 |      50.2 |      24.8
 Little Italy         |      14178 |      39.4 |       1.2
 Yorkville            |      57800 |      31.2 |      33.3
 Inwood               |      50922 |      29.3 |      14.9
 Lower East Side      |     104690 |      28.3 |       9.0
 Washington Heights   |     187198 |      26.9 |      16.3
 East Harlem          |      62279 |      20.2 |      46.2
 Hamilton Heights     |      71133 |      14.6 |      41.1
 Chinatown            |      18195 |      10.3 |       4.2
 Harlem               |     125501 |       5.7 |      80.5
```

Que ce passe-t-il ici ? Voici ce qui se passe (l'ordre d'évaluation est optimisé par la base de données) :

1. La clause `JOIN` crée une table virtuelle qui contient les colonnes à la fois des quartiers et des recensements (tables neighborhoods et census).
2. La clause `WHERE` filtre la table virtuelle pour ne conserver que la ligne correspondant à Manhattan.
3. Les lignes restantes sont regroupées par le nom du quartier et sont utilisées par la fonction d'agrégation : **Sum()** pour réaliser la somme des valeurs de la population.
4. Après un peu d'arithmétique et de formatage (ex : `GROUP BY`, `ORDER BY`)) sur le nombres finaux, notre requête calcule les pourcentages.

Note : La clause `JOIN` combine deux parties `FROM`. Par défaut, nous utilisons un jointure du type :`INNER JOIN`, mais il existe quatres autres types de jointures. Pour de plus amples informations à ce sujet, consultez la partie type_jointure de la page de la documentation officielle de PostgreSQL.

Nous pouvons aussi utiliser le test de la distance dans notre clef de jointure, pour créer une regroupement de "tous les éléments dans un certain rayon". Essayons d'analyser la géographie raciale de New York en utilisant les requêtes de distance.

Premièrement, essayons d'obtenir la répartition raciale de la ville.

```
SELECT
  100.0 * Sum(popn_white) / Sum(popn_total) AS white_pct,
  100.0 * Sum(popn_black) / Sum(popn_total) AS black_pct,
  Sum(popn_total) AS popn_total
FROM nyc_census_blocks;
```

```
      white_pct       |       black_pct       | popn_total
----------------------+-----------------------+------------
 44.6586020115685295 | 26.5945063345703034 |    8008278
```

Donc, 8M de personnes dans New York, environ 44% sont "blancs" et 26% sont "noirs".

Duke Ellington chantait que "You / must take the A-train / To / go to Sugar Hill way up in Harlem." Comme nous l'avons vu précédemment, Harlem est de très loin le quartier ou se trouve la plus grande concentration d'africains-américains de Manhattan (80.5%). Est-il toujours vrai qu'il faut prendre le train A dont Duke parlait dans sa chanson ?

Premièrement, le contenu du champ `routes` de la table `nyc_subway_stations` va nous servir à récupérer le train A. Les valeurs de ce champs sont un peu complexes.

```
SELECT DISTINCT routes FROM nyc_subway_stations;
```

```
A,C,G
4,5
D,F,N,Q
5
E,F
E,J,Z
R,W
```

Note : Le mot clef `DISTINCT` permet d'éliminer les répétitions de lignes de notre résultat. Dans ce mot clef, notre requête renverrait 491 résultats au lieu de 73.

Donc pour trouver le train A, nous allons demander toutes les lignes ayant pour `routes` la valeur 'A'. Nous pouvons faire cela de différentes manières, mais nous utiliserons aujourd'hui le fait que la fonction **strpos(routes,'A')** retourne un entier différent de 0 si la lettre 'A' se trouve dans la valeur du champ route.

```
SELECT DISTINCT routes
FROM nyc_subway_stations AS subways
WHERE strpos(subways.routes,'A') > 0;
```

```
A,B,C
A,C
A
A,C,G
A,C,E,L
A,S
A,C,F
A,B,C,D
A,C,E
```

Essayons de regrouper la répartition raciale dans un rayon de 200 mètres de la ligne du train A.

```
SELECT
  100.0 * Sum(popn_white) / Sum(popn_total) AS white_pct,
  100.0 * Sum(popn_black) / Sum(popn_total) AS black_pct,
  Sum(popn_total) AS popn_total
FROM nyc_census_blocks AS census
JOIN nyc_subway_stations AS subways
ON ST_DWithin(census.the_geom, subways.the_geom, 200)
WHERE strpos(subways.routes,'A') > 0;
```

```
      white_pct       |      black_pct       | popn_total
----------------------+----------------------+------------
 42.0805466940877366 | 23.0936148851067964 |     185259
```

La répartition raciale le long de la ligne du train A n'est pas radicalement différente de la répartition générale de la ville de New York.

13.2 Jointures avancées

Dans la dernière partie nous avons vu que le train A n'est pas utilisé par des populations si éloignées de la répartition totale du reste de la ville. Y-a-t-il des trains qui passent par des parties de la ville qui ne sont pas dans la moyenne de la répartition raciale ?

Pour répondre à cette question, nous ajouterons une nouvelle jointure à notre requête, de telle manière que nous puissions calculer simultanément la répartition raciale de plusieurs lignes de métro à la fois. Pour faire ceci, nous créerons une table qui permettra d'énumérer toutes les lignes que nous voulons regrouper.

```
CREATE TABLE subway_lines ( route char(1) );
INSERT INTO subway_lines (route) VALUES
  ('A'),('B'),('C'),('D'),('E'),('F'),('G'),
  ('J'),('L'),('M'),('N'),('Q'),('R'),('S'),
  ('Z'),('1'),('2'),('3'),('4'),('5'),('6'),
  ('7');
```

Maintenant nous pouvons joindre les tables des lignes de métro à notre requête précédente.

```
SELECT
  lines.route,
  Round(100.0 * Sum(popn_white) / Sum(popn_total), 1) AS white_pct,
  Round(100.0 * Sum(popn_black) / Sum(popn_total), 1) AS black_pct,
  Sum(popn_total) AS popn_total
FROM nyc_census_blocks AS census
JOIN nyc_subway_stations AS subways
```

```
ON ST_DWithin(census.the_geom, subways.the_geom, 200)
JOIN subway_lines AS lines
ON strpos(subways.routes, lines.route) > 0
GROUP BY lines.route
ORDER BY black_pct DESC;
```

```
 route | white_pct | black_pct | popn_total
-------+-----------+-----------+------------
 S     |      30.1 |      59.5 |      32730
 3     |      34.3 |      51.8 |     201888
 2     |      33.6 |      45.5 |     535414
 5     |      32.1 |      45.1 |     407324
 C     |      41.3 |      35.9 |     430194
 4     |      34.7 |      30.9 |     328292
 B     |      36.1 |      30.6 |     261186
 Q     |      52.9 |      26.3 |     259820
 J     |      29.5 |      23.6 |     126764
 A     |      42.1 |      23.1 |     370518
 Z     |      29.5 |      21.5 |      81493
 D     |      39.8 |      20.9 |     233855
 G     |      44.8 |      20.0 |     138602
 L     |      53.9 |      17.1 |     104140
 6     |      52.7 |      16.3 |     257769
 1     |      54.8 |      12.6 |     659028
 F     |      60.0 |       8.6 |     438212
 M     |      50.0 |       7.8 |     166721
 E     |      69.4 |       5.3 |      86118
 R     |      57.7 |       4.8 |     389124
 7     |      42.4 |       3.8 |     107543
```

Comme précédemment, les jointures créent une table virtuelle de toutes les combinaisons possibles et disponibles à l'aide des contraintes de type `JOIN ON`. Ces lignes sont ensuite utilisées dans le regroupement ``GROUP`. La magie spatiale tient dans l'utilisation de la fonction `ST_DWithin` qui s'assure que les blocs sont suffisamment proches des lignes de métros incluses dans le calcul.

13.3 Liste de fonctions

ST_Contains(geometry A, geometry B) : retourne TRUE si et seulement si aucun point de B est à l'extérieur de A, et si au moins un point à l'intérieur de B est à l'intérieur de A.

ST_DWithin(geometry A, geometry B, radius) : retourne TRUE si les géométries sont distantes du rayon donné.

ST_Intersects(geometry A, geometry B) : retourne TRUE si les géométries/géographies "s'intersectent spatialement" (partage une portion de l'espace) et FALSE sinon (elles sont disjointes).

round(v numeric, s integer) : fonction de PostgreSQL qui arrondit à s décimales.

strpos(chaîne, sous-chaîne) : fonction de chaîne de caractères de PostgreSQL qui retourne la position de la sous-chaine.

sum(expression) : fonction d'agrégation de PostgreSQL qui retourne la somme d'un ensemble de valeurs.

CHAPITRE 14

Partie 13 : Exercices sur jointures spatiales

Voici un petit rappel de certaines des fonctions vues précédemment. Elles seront utiles pour les exercices !

- **sum(expression)** agrégation retournant la somme d'un ensemble
- **count(expression)** agrégation retournant le nombre d'éléments d'un ensemble
- **ST_Area(geometry)** retourne l'aire d'un polygone
- **ST_AsText(geometry)** retourne un texte WKT
- **ST_Contains(geometry A, geometry B)** retourne TRUE si la géométrie A contient la géométrie B
- **ST_Distance(geometry A, geometry B)** retourne la distance minimum entre deux géométries
- **ST_DWithin(geometry A, geometry B, radius)** retourne TRUE si la A est distante d'au plus radius de B
- **ST_GeomFromText(text)** retourne une géométrie
- **ST_Intersects(geometry A, geometry B)** retourne TRUE si la géométrie A intersecte la géométrie B
- **ST_Length(linestring)** retourne la longueur d'une ligne
- **ST_Touches(geometry A, geometry B)** retourne TRUE si le contour extérieur de A touche B
- **ST_Within(geometry A, geometry B)** retourne TRUE si A est hors de B

Souvenez-vous aussi des tables à votre disposition :

- `nyc_census_blocks`
  - name, popn_total, boroname, the_geom
- `nyc_streets`
  - name, type, the_geom
- `nyc_subway_stations`
  - name, routes, the_geom
- `nyc_neighborhoods`
  - name, boroname, the_geom

14.1 Exercices

- **"Quelle station de métro se situe dans le quartier 'Little Italy' ? Quelle est l'itinéraire de métro à emprunter ?"**

```
SELECT s.name, s.routes
FROM nyc_subway_stations AS s
```

```
JOIN nyc_neighborhoods AS n
ON ST_Contains(n.the_geom, s.the_geom)
WHERE n.name = 'Little Italy';
```

```
   name     | routes
-----------+--------
 Spring St | 6
```

– **"Quels sont les quartiers desservis pas le train numéro 6 ?"** (Astuce : la colonne `routes` de la table `nyc_subway_stations` dispose des valeurs suivantes : 'B,D,6,V' et 'C,6')

```
SELECT DISTINCT n.name, n.boroname
FROM nyc_subway_stations AS s
JOIN nyc_neighborhoods AS n
ON ST_Contains(n.the_geom, s.the_geom)
WHERE strpos(s.routes,'6') > 0;
```

```
        name         | boroname
--------------------+-----------
 Midtown             | Manhattan
 Hunts Point         | The Bronx
 Gramercy            | Manhattan
 Little Italy        | Manhattan
 Financial District  | Manhattan
 South Bronx         | The Bronx
 Yorkville           | Manhattan
 Murray Hill         | Manhattan
 Mott Haven          | The Bronx
 Upper East Side     | Manhattan
 Chinatown           | Manhattan
 East Harlem         | Manhattan
 Greenwich Village   | Manhattan
 Parkchester         | The Bronx
 Soundview           | The Bronx
```

Note : Nous avons utilisé le mot clef `DISTINCT` pour supprimer les répétitions dans notre ensemble de résultats où il y avait plus d'une seule station de métro dans le quartier.

– **"Après le 11 septembre, le quartier de 'Battery Park' était interdit d'accès pendant plusieurs jours. Combien de personnes ont dû être évacuées ?"**

```
SELECT Sum(popn_total)
FROM nyc_neighborhoods AS n
JOIN nyc_census_blocks AS c
ON ST_Intersects(n.the_geom, c.the_geom)
WHERE n.name = 'Battery Park';
```

```
9928
```

– **"Quelle est la densité de population (personne / km^2) des quartiers de 'Upper West Side' et de 'Upper East Side' ?"** (Astuce : il y a 1000000 m^2 dans un km^2.)

```
SELECT
  n.name,
  Sum(c.popn_total) / (ST_Area(n.the_geom) / 1000000.0) AS popn_per_sqkm
FROM nyc_census_blocks AS c
JOIN nyc_neighborhoods AS n
```

```
ON ST_Intersects(c.the_geom, n.the_geom)
WHERE n.name = 'Upper West Side'
OR n.name = 'Upper East Side'
GROUP BY n.name, n.the_geom;
```

```
      name        |  popn_per_sqkm
-----------------+------------------
 Upper East Side | 47943.3590089405
 Upper West Side | 39729.5779474286
```

CHAPITRE 15

Partie 14 : L'indexation spatiale

Rapellez-vous que l'indexation spatiale est l'une des trois fonctionnalités clés d'une base de données spatiales. Les index permettent l'utilisation de grandes quantités de données dans une base. Sans l'indexation, chaque recherche d'entité nécessitera d'accéder séquentiellement à tous les enregistrements de la base de données. L'indexation accélère les recherches en organisant les données dans des arbres de recherche qui peuvent être parcourus efficacement pour retrouver une entité particulière.

L'indexation spatiale l'un des plus grands atouts de PostGIS. Dans les exemples précédents, nous avons construit nos jointures spatiales en comparant la totalité des tables. Ceci peut parfois s'avérer très coûteux : réaliser la jointure de deux tables de 10000 enregistrements sans indexation nécessitera de comparer 100000000 valeurs, les comparaisons requises ne seront plus que 20000 avec l'indexation.

Lorsque nous avons chargé la table `nyc_census_blocks`, l'outil pgShapeLoader crée automatiquement un index spatial appelé `nyc_census_blocks_the_geom_gist`.

Pour démontrer combien il est important d'indexer ses données pour la performance des requêtes, essayons de requêter notre table `nyc_census_blocks` **sans** utiliser notre index.

La première étape consiste à supprimer l'index.

```
DROP INDEX nyc_census_blocks_the_geom_gist;
```

Note : La commande `DROP INDEX` supprime un index existant de la base de données. Pour de plus amples informations à ce sujet, consultez la documentation officielle de PostgreSQL.

Maintenant, regardons le temps d'exécution dans le coin en bas à droite de l'interface de requêtage de pgAdmin, puis lançons la commande suivante. Notre requête recherche les blocs de la rue Broad.

```
SELECT blocks.blkid
FROM nyc_census_blocks blocks
JOIN nyc_subway_stations subways
ON ST_Contains(blocks.the_geom, subways.the_geom)
WHERE subways.name = 'Broad St';

      blkid
-----------------
 360610007003006
```

La table `nyc_census_blocks` est très petite (seulement quelque milliers d'enregistrements) donc même sans l'index, la requête prends **55 ms** sur l'ordinateur de test.

Maintenant remettons en place l'index et lançons de nouveau la requête.

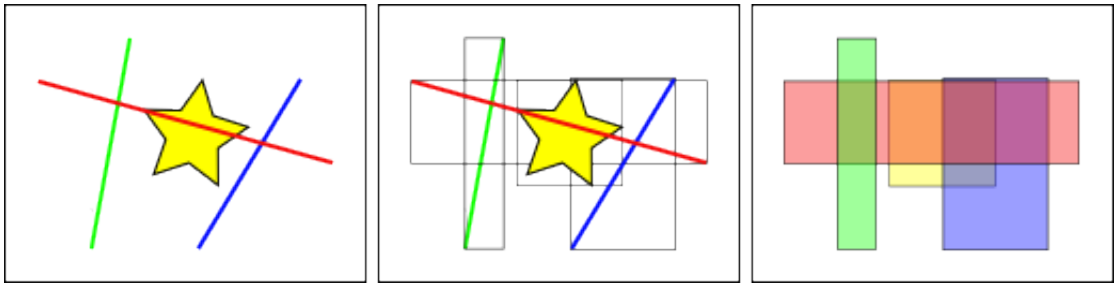
```
CREATE INDEX nyc_census_blocks_the_geom_gist ON nyc_census_blocks USING GIST (the_geom);
```

Note : l'utilisation de la clause `USING GIST` spécifie à PostgreSQL de créer une structure (GIST) pour cet index. Si vous recevez un message d'erreur ressemblant à `ERROR: index row requires 11340 bytes, maximum size is 8191` lors de la création, cela signifie sans doute que vous avez omis la clause `USING GIST`.

Sur l'ordinateur de test le temps d'exécution se réduit à **9 ms**. Plus votre table est grande, plus la différence de temps d'exécution pour une requête utilisant les index augmentera.

15.1 Comment les index spatiaux fonctionnent

Les index des bases de données standards créent des arbres hiérarchiques basés sur les valeurs des colonnes à indexer. Les index spatiaux sont un peu différents - ils ne sont pas capables d'indexer des entités géométriques elles-même mais ils indexent leur étendues.



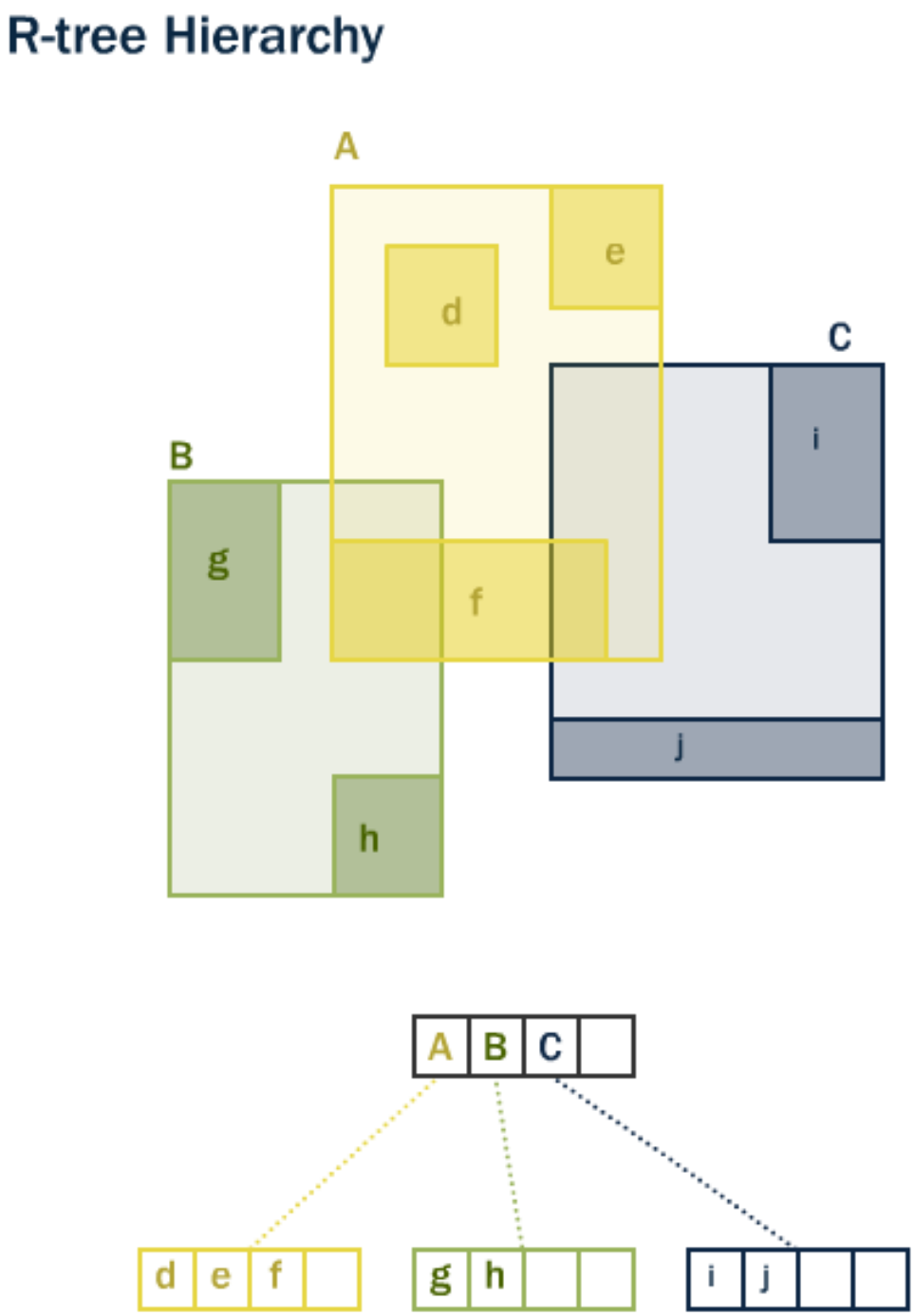
Dans la figure ci-dessus, le nombre de lignes qui intersectent l'étoile jaune est *unique*, la ligne rouge. Mais l'étendue des entités qui intersectent la boîte jaune sont *deux*, la boîte rouge et la boîte bleue.

La manière dont les bases de données répondent de manière efficace à la question "Quelles lignes intersectent l'étoile jaune ?" correspond premièrement à répondre à la question "Quelle étendue intersecte l'étendue jaune" en utilisant les index (ce qui est très rapide) puis à calculer le résultat exact de la question "Quelles lignes intersectent l'étoile jaune ?" **seulement en utilisant les entités retournées par le premier test**.

Pour de grandes tables, il y a un système en "deux étapes" d'évaluation en utilisant dans un premier temps l'approximation à l'aide d'index, puis en réalisant le test exact sur une quantité bien moins importante de données ce qui réduit drastiquement le temps de calcul nécessaire à cette deuxième étape.

PotGIS et Oracle Spatial partage la même notion d'index structuré sous la forme "d'arbres R"[1]. Les arbres R classent les données sous forme de rectangles, de sous-rectangles etc. Cette structure d'index gère automatiquement la densité et la taille des objets.

1. http ://postgis.org/support/rtree.pdf



15.2 Requête avec seulement des index

La plupart des fonctions utilisées par PostGIS (**ST_Contains**, **ST_Intersects**, **ST_DWithin**, etc) prennent en compte les index automatiquement. Mais certaines fonctions (comme par exemple : **ST_Relate**) ne les utilisent pas.

Pour utiliser une recherche par étendue utilisant les index (et pas de filtres), vous pouvez utiliser l'opérateur **&&**. Pour les géométries, l'opérateur **&&** signifie "l'étendue recouvre ou touche" de la même

manière que l'opérateur **=** sur des entiers signifie que les valeurs sont égales.

Essayons de comparer une requête avec seulement un index pour la population du quartier 'West Village'. En utilisant la commande **&&** notre requête ressemble à cela :

```
SELECT Sum(popn_total)
FROM nyc_neighborhoods neighborhoods
JOIN nyc_census_blocks blocks
ON neighborhoods.the_geom && blocks.the_geom
WHERE neighborhoods.name = 'West Village';
```

```
50325
```

Maintenant essayons la même requête en utilisant la fonction plus précise **ST_Intersects**.

```
SELECT Sum(popn_total)
FROM nyc_neighborhoods neighborhoods
JOIN nyc_census_blocks blocks
ON ST_Intersects(neighborhoods.the_geom, blocks.the_geom)
WHERE neighborhoods.name = 'West Village';
```

```
27141
```

Un plus faible nombre de résultats ! La première requête nous renvoie tous les blocs qui intersectent l'étendue du quartier, la seconde nous renvoie seulement les blocs qui intersectent le quartier lui-même.

15.3 Analyse

Le planificateur de requête de PostgreSQL choisit intelligemment d'utiliser ou non les index pour réaliser une requête. Il n'est pas toujours plus rapide d'utiliser un index pour réaliser une recherche : si la recherche doit renvoyer l'ensemble des enregistrements d'une table, parcourir l'index pour récupérer chaque valeur sera plus lent que de parcourir linéairement l'ensemble de la table.

Afin de savoir dans quelle situation il est nécessaire d'utiliser les index (lire une petite partie de la table plutôt qu'une grande partie), PostgreSQL conserve des statistiques relatives à la distribution des données dans chaque colonne indexée. Par défaut, PostgreSQL rassemble les statistiques sur une base régulière. Néanmoins, si vous changez dramatiquement le contenu de vos tables dans une période courte, les statistiques ne seront alors plus à jour.

Pour vous assurez que les statistiques correspondent bien au contenu de la table actuelle, il est courant d'utiliser la commande `ANALYZE` après un grand nombre de modifications ou de suppression de vos données. Cela force le système de gestion des statistiques à récupérer l'ensemble des données des colonnes indexées.

La commande `ANALYZE` demande à PostgreSQL de parcourir la table et de mettre à jour les statistiques utilisées par le planificateur de requêtes (la planification des requêtes sera traité ultérieurement).

```
ANALYZE nyc_census_blocks;
```

15.4 Néttoyage

Il est souvent stressant de constater que la simple création d'un index n'est pas suffisant pour que PostgreSQL l'utilise efficacement. Le nettoyage doit être réalisé après qu'un index soit créé ou après un grand nombre de requêtes UDATE, INSERT ou DELETE ait été réalisé sur une table. La commande `VACUUM` demande à PostgreSQL de récupérer chaque espace non utilisé dans les pages de la table qui sont laissées en l'état lors des requêtes UPDATE ou DELETE à cause du modèle d'estampillage multi-versions.

Le nettoyage des données est tellement important pour une utilisation efficace du serveur de base de données PostgreSQL qu'il existe maintenant une option "autovacuum".

Activée par défaut, le processus autovacuum nettoie (récupère l'espace libre) et analyse (met à jour les statistiques) vos tables suivant un intervalle donné déterminé par l'activité des bases de données. Bien que cela fonctionne avec les bases de données hautement transactionnelles, il n'est pas supportable de devoir attendre que le processus autovacuum se lance lors de la mise à jour ou la suppression massive de données. Dans ce cas, il faut lancer la commande `VACUUM` manuellement.

Le nettoyage et l'analyse de la base de données peuvent être réalisés séparément si nécessaire. Utiliser la commande `VACUUM` ne mettra pas à jour les statistiques alors que lancer la commande `ANALYZE` ne récupèrera pas l'espace libre des lignes d'une table. Chacune de ces commandes peut être lancée sur l'intégralité de la base de données, sur une table ou sur une seule colonne.

```
VACUUM ANALYZE nyc_census_blocks;
```

15.5 Liste des fonctions

geometry_a && geometry_b : retourne TRUE si l'étendue de A chevauche celle de B.

geometry_a = geometry_b : retourne TRUE si l'étendue de A est la même que celle de B.

ST_Intersects(geometry_a, geometry_b) : retourne TRUE si la géométrie *a* "intersecte spatialement" la géométrie '*b*- (si elles ont une partie en commun) et FALSE sinon (elles sont disjointes).

CHAPITRE 16

Partie 15 : Projections des données

La Terre n'est pas plate et il n'y a pas de moyen simple de la poser à plat sur une carte en papier (ou l'écran d'un ordinateur). Certaines projections préservent les aires, donc tous les objets ont des tailles relatives aux autres, d'autre projections conservent les angles (conformes) comme la projection Mercator. Certaines projections tentent de minimiser la distorsion des différents paramètres. Le point commun entre toutes les projections est qu'elles transforment le monde (sphérique) en un système plat de coordonnées cartésiennes, et le choix de la projection dépend de ce que vous souhaitez faire avec vos données.

Nous avons déjà rencontré des projections, lorsque nous avons chargé les données de la ville de Ney York .Rappelez-vous qu'elles utilisaient le SRID 26918. Parfois, vous aurez malgré tout besoin de transformer et de reprojeter vos données d'un système de projection à un autre, en utilisant la fonction **ST_Transform(geometry, srid)**. Pour manipuler les identifiants de système de référence spatiale à partir d'une géométrie, PostGIS fournit les fonctions **ST_SRID(geometry)** et **ST_SetSRID(geometry, srid)**.

Nous pouvons vérifier le SRID de nos données avec la commande **ST_SRID** :

```
SELECT ST_SRID(the_geom) FROM nyc_streets LIMIT 1;
```

```
26918
```

Et quelle est la définition du "26918" ? Comme nous l'avons vu lors de la partie "*chargement des données*", la définition se trouve dans la table `spatial_ref_sys`. En fait, **deux** définitions sont présentes. La définition au format *WKT* dans la colonne `srtext`

```
SELECT * FROM spatial_ref_sys WHERE srid = 26918;
```

En fait, pour les calculs internes de re-projection, c'est le contenu de la colonne `proj4text` qui est utilisé. Pour notre projection 26918, voici la définition au format proj.4 :

```
SELECT proj4text FROM spatial_ref_sys WHERE srid = 26918;
```

```
+proj=utm +zone=18 +ellps=GRS80 +datum=NAD83 +units=m +no_defs
```

En pratique, les deux colonnes `srtext` et `proj4text` sont importantes : la colonne `srtext` est utilisée par les applications externes comme GeoServer, uDig <udig.refractions.net>'_, FME et autres, alors que la colonne `proj4text` est principalement utilisée par PostGIS en interne.

16.1 Comparaison de données

Combinés, une coordonnée et un SRID définissent une position sur le globe. Sans le SRID, une coordonnée est juste une notion abstraite. Un système de coordonnées "cartésiennes" est définit comme un système de coordonnées "plat" sur la surface de la Terre. Puisque les fonctions de PostGIS utilisent cette surface plane, les opérations de comparaison nécessitent que l'ensemble des objets géométriques soient représentés dans le même système, ayant le même SRID.

Si vous utilisé des géométries avec différents SRID vous obtiendrez une erreur comme celle-ci :

```
SELECT ST_Equals(
         ST_GeomFromText('POINT(0 0)', 4326),
         ST_GeomFromText('POINT(0 0)', 26918)
         );

ERROR:  Operation on two geometries with different SRIDs
CONTEXT:  SQL function "st_equals" statement 1
```

Note : Faites attention de pas utiliser la transformation à la volée à l'aide de **ST_Transform** trop souvent. Les index spatiaux sont construits en utilisant le SRID inclus dans les géométries. Si la comparaison est faite avec un SRID différent, les index spatiaux ne seront pas (la plupart du temps) utilisés. Il est reconnu qu'il vaut mieux choisir **un SRID** pour toutes les tables de votre base de données. N'utilisez la fonction de tranformation que lorsque vous lisez ou écrivez les données depuis une application externe.

16.2 Transformer les données

Si vous retournez à la définition au format proj4 du SRID 26918, vous pouvez voir que notre projection actuelle est de type UTM zone 18 (Universal Transvers Mercator), avec le mètre comme unité de mesure.

```
+proj=utm +zone=18 +ellps=GRS80 +datum=NAD83 +units=m +no_defs
```

Essayons de convertir certaines données de notre système de projection dans un système de coordonnées géographiques connu comme "longitude/latitude".

Pour convertir les données d'un SRID à l'autre, nous devons dans un premier temps vérifier que nos géométries ont un SRID valide. Une fois que nous avons vérifié cela, nous devons ensuite trouver le SRID dans lequel nous souhaitons re-projeter. En d'autre terme, quel est le SRID des coordonnées géographiques ?

Le SRID le plus connu pour les coordonnées géographiques est le 4326, qui correspond au couple "longitude/latitude sur la sphéroïde WGS84". Vous pouvez voir sa définition sur le site http ://spatialreference.org.

> http ://spatialreference.org/ref/epsg/4326/

Vous pouvez aussi récupérer les définitions dans la table `spatial_ref_sys` :

```
SELECT srtext FROM spatial_ref_sys WHERE srid = 4326;
```

```
GEOGCS["WGS 84",
  DATUM["WGS_1984",
    SPHEROID["WGS 84",6378137,298.257223563,AUTHORITY["EPSG","7030"]],
    AUTHORITY["EPSG","6326"]],
  PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],
  UNIT["degree",0.01745329251994328,AUTHORITY["EPSG","9122"]],
  AUTHORITY["EPSG","4326"]]
```

Essayons de convertir les cordonnées de la station 'Broad St' :

```
SELECT ST_AsText(ST_Transform(the_geom,4326))
FROM nyc_subway_stations
WHERE name = 'Broad St';

POINT(-74.0106714688735 40.7071048155841)
```

Si vous chargez les données ou créez une nouvelle géométrie sans spécifier de SRID, la valeur du SRID prendra alors la valeur -1. Rappelez-vous que dans les *Partie 8 : Les géometries*, lorsque nous avons créé nos tables géométriques nous n'avions pas spécifié un SRID. Si nous interrogeons la base, nous devons nous attendre à ce que toutes les tables préfixées par `nyc_` aient le SRID 26918, alors que la table `geometries` aura la valeur -1 par défaut.

Pour visualiser la table d'assignation des SRID, interrogez la table `geometry_columns` de la base de données.

```
SELECT f_table_name AS name, srid
FROM geometry_columns;

        name          | srid
----------------------+-------
 nyc_census_blocks    | 26918
 nyc_neighborhoods    | 26918
 nyc_streets          | 26918
 nyc_subway_stations  | 26918
 geometries           |    -1
```

Néanmoins, si vous connaissez le SRID de vos données, vous pouvez l'affecter par la suite en utilisant la fonction **ST_SetSRID** sur les géométries. Ensuite vous pourrez les transformer dans d'autres systèmes de projections.

```
SELECT ST_AsText(
 ST_Transform(
   ST_SetSRID(geom,26918),
 4326)
)
FROM geometries;
```

16.3 Liste des fonctions

ST_AsText : retourne la représentation au format Well-Known Text (WKT) sans la métadonnée SRID.

ST_SetSRID(geometry, srid) : affecte une valeur au SRID d'une géométrie.

ST_SRID(geometry) : retourne l'identifiant du système de référence spatiale d'un objet ST_Geometry comme défini dans la table spatial_ref_sys.

ST_Transform(geometry, srid) : retourne une nouvelle géométrie après avoir re-projeté les données dans le système correspondant au SRID passé en paramètre.

CHAPITRE 17

Partie 16 : Exercices sur les projections

Voici un rappel de certaines fonctions que nous avons vu. Elles seront utiles pour les exercices !
- **sum(expression)** agrégation qui retourne la somme d'un ensemble de valeurs
- **ST_Length(linestring)** retourne la longueur d'une ligne
- **ST_SRID(geometry, srid)** retourne le SRID d'une géométrie
- **ST_Transform(geometry, srid)** reprojette des géométries dans un autre système de référence spatiale
- **ST_GeomFromText(text)** retourne un objet `geometry`
- **ST_AsText(geometry)** retourne un WKT (`texte`)
- **ST_AsGML(geometry)** retourne un GML (`texte`)

Rappelez-vous les ressources en ligne :
- http ://spatialreference.org
- http ://prj2epsg.org

Et les tables disponibles :
- `nyc_census_blocks`
  - name, popn_total, boroname, the_geom
- `nyc_streets`
  - name, type, the_geom
- `nyc_subway_stations`
  - name, the_geom
- `nyc_neighborhoods`
  - name, boroname, the_geom

17.1 Exercices

- **"Quelle est la longueur des rue de New York, mesurée en UTM 18 ?"**

```
SELECT Sum(ST_Length(the_geom))
  FROM nyc_streets;
```

```
10418904.7172
```

- **"Quelle est la définition du SRID 2831 ?"**

```
SELECT srtext FROM spatial_ref_sys
WHERE SRID = 2831;
```

Ou, via prj2epsg

```
PROJCS["NAD83(HARN) / New York Long Island",
GEOGCS["NAD83(HARN)",
  DATUM["NAD83 (High Accuracy Regional Network)",
    SPHEROID["GRS 1980", 6378137.0, 298.257222101, AUTHORITY["EPSG","7019"]],
    TOWGS84[-0.991, 1.9072, 0.5129, 0.0257899075194932, -0.00965009896027040 2, -0.0
    AUTHORITY["EPSG","6152"]],
  PRIMEM["Greenwich", 0.0, AUTHORITY["EPSG","8901"]],
  UNIT["degree", 0.017453292519943295],
  AXIS["Geodetic longitude", EAST],
  AXIS["Geodetic latitude", NORTH],
  AUTHORITY["EPSG","4152"]],
PROJECTION["Lambert Conic Conformal (2SP)", AUTHORITY["EPSG","9802"]],
PARAMETER["central_meridian", -74.0],
PARAMETER["latitude_of_origin", 40.166666666666664],
PARAMETER["standard_parallel_1", 41.03333333333333],
PARAMETER["false_easting", 300000.0],
PARAMETER["false_northing", 0.0],
PARAMETER["scale_factor", 1.0],
PARAMETER["standard_parallel_2", 40.666666666666664],
UNIT["m", 1.0],
AXIS["Easting", EAST],
AXIS["Northing", NORTH],
AUTHORITY["EPSG","2831"]]
```

– **"Quelle est la longueur des rues de New York, mesurée en utilisant le SRID 2831 ?"**

```
SELECT Sum(ST_Length(ST_Transform(the_geom,2831)))
  FROM nyc_streets;
```

```
10421993.706374
```

Note : La différence entre les mesure en UTM 18 et en Stateplane Long Island est de (10421993 - 10418904)/10418904, soit 0.02%. Calculé sur la sphéroïde en utilisant en *Partie 17 : Coordonnées géographiques*, le total des longueurs des routes est 10421999, ce qui est proche de la valeur dans l'autre système de projection (Stateplane Long Island). Ce dernier est précisément calibré pour une petite zone géographique (la ville de New York) alors que le système UTM 18 doit fournir un résultat raisonnable pour une zone régionale beaucoup plus large.

– **"Quelle est la représentation KML du point de la station de métris 'Broad St' ?"**

```
SELECT ST_AsKML(the_geom)
FROM nyc_subway_stations
WHERE name = 'Broad St';
```

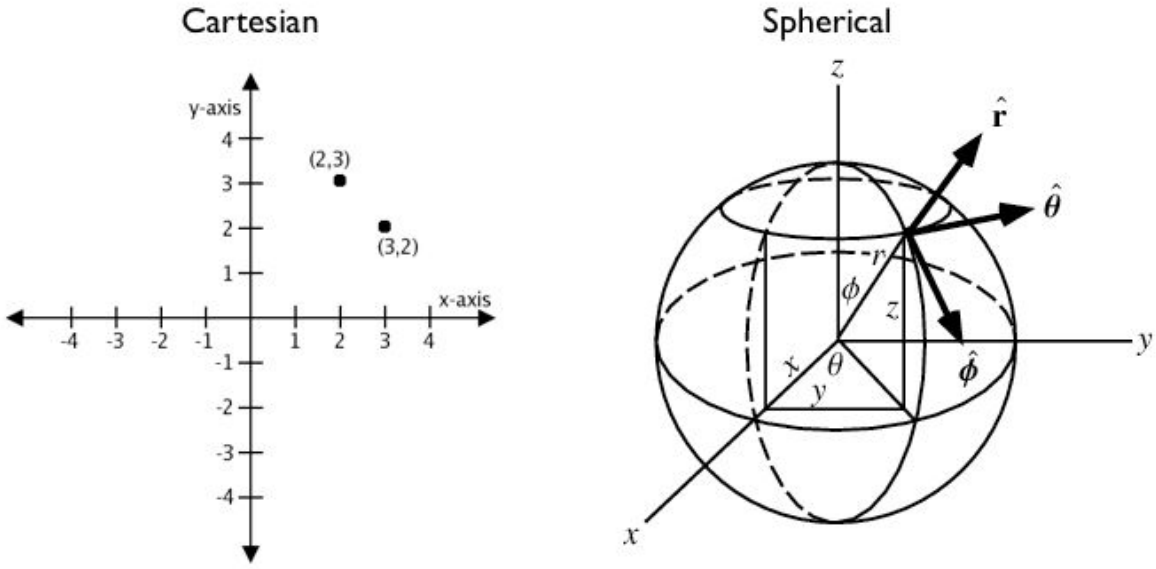
```
<Point><coordinates>-74.010671468873468,40.707104815584088</coordinates></Point>
```

Hé ! les coordonnées sont géographiques bien que nous n'ayons pas fait appel à la fonction **ST_Transform**, mais pourquoi ? Parce que le standard KML spécifie que toutes les coordonnées *doivent* être géographiques (en fait, dans le système EPSG :4326), donc la fonction **ST_AsKML** réalise la transformation automatiquement.

Partie 17 : Coordonnées géographiques

Il est très fréquent de manipuler des données à coordonnées “géographiques” ou de “longitude/latitude”.

Au contraire des coordonnées de type Mercator, UTM ou Stateplane, les coordonnées géographiques ne représentent pas une distance linéaire depuis une origine, tel que dans un plan. Elles décrivent la distance angulaire entre l’équateur et les pôles. Dans les sytèmes de coordonnées sphériques, un point est spécifié par son rayon (distance à l’origine), son angle de rotation par rapport au méridien plan, et son angle par rapport à l’axe pôlaire.



Vous pouvez continuer à utiliser des coordonnées géographiques comme des coordonnées cartésiennes approximatives pour vos analyses spatiales. Par contre les mesures de distances, d’aires et de longueurs seront erronées. Etant donné que les coordonnées sphériques mesurent des angles, l’unité est le degré. Par exemple, les résultats cartésien approximatifs de tests tels que ‘intersects’ et ‘contains’ peuvent s’avérer terriblement faux. Par ailleurs, plus une zone est située près du pôle ou de la ligne de date internationale, plus la distance entre les points est agrandie.

Voici par exemple les coordonnées des villes de Los Angeles et Paris.
– Los Angeles : `POINT(-118.4079 33.9434)`
– Paris : `POINT(2.3490 48.8533)`

La requête suivante calcule la distance entre Los Angeles et Paris en utilisant le système cartésien standard de PostGIS **ST_Distance(geometry, geometry)**. Notez que le SRID 4326 déclare un système de référence spatiale géographique.

```
SELECT ST_Distance(
  ST_GeometryFromText('POINT(-118.4079 33.9434)', 4326), -- Los Angeles (LAX)
  ST_GeometryFromText('POINT(2.5559 49.0083)', 4326)     -- Paris (CDG)
  );
```

```
121.898285970107
```

Aha ! 121 ! Mais, que veut dire cela ?

L'unité pour SRID 4326 est le degré. Donc la réponse signifie 121 degrés. Sur une sphère, la taille d'un degré "au carré" est assez variable. Elle devient plus petite au fur et à mesure que l'on s'éloigne de l'équateur. Pensez par exemple aux méridiens sur le globe qui se resserrent entre eux au niveau des pôles. Donc une distance de 121 degrés ne veut rien dire !

Pour calculer une distance ayant du sens, nous devons traiter les coordonnées géographiques non pas comme des coordonnées cartésiennes approximatives, mais plutôt comme de réelles coordonnées sphériques. Nous devons mesurer les distances entre les points comme de vrais chemins par dessus une sphère, comme une portion d'un grand cercle.

Depuis sa version 1.5, PostGIS fournit cette fonctionnalité avec le type `geography`.

Note : Différentes bases de données spatiales développent différentes approches pour manipuler les coordonnées géographiques.

- Oracle essaye de mettre à jour la différence de manière transparente en lançant des calculs lorsque le SRID est géographique.
- SQL Server utilise deux types spatiaux, "STGeometry" pour les coordonnées cartésiens et STGeography" pour les coordonnées géographqiues.
- Informix Spatial est une pure extension cartésienne d'Informix, alors qu'Informix Geodetic est une pure extension géographique.
- Comme SQL Server, PostGIS utilise deux types : "geometry" et "geography".

En utilisant le type `geography` plutot que `geometry`, essayons sà nouveau de mesurer la distance entre Los Angeles et Paris. Au lieu de la commande **ST_GeometryFromText(text)**, nous utiliserons cette fois **ST_GeographyFromText(text)**.

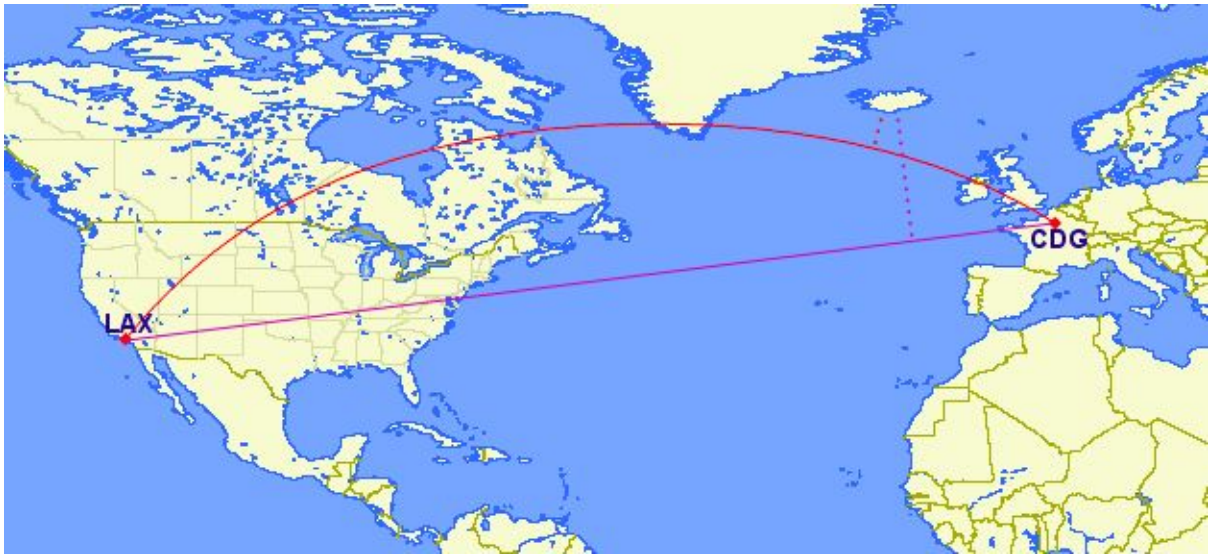
```
SELECT ST_Distance(
  ST_GeographyFromText('POINT(-118.4079 33.9434)'), -- Los Angeles (LAX)
  ST_GeographyFromText('POINT(2.5559 49.0083)')     -- Paris (CDG)
  );
```

```
9124665.26917268
```

Toutes les valeurs retournées étant en mètres, notre réponse est donc 9124 kilomètres.

Les versions plus anciennes de PostGIS supportaient uniquement des calculs sur sphère très basiques comme la fonction **ST_Distance_Spheroid(point, point, measurement)**. Celle-ci est très limitée et ne fonctionne uniquement sur des points. Elle ne supporte pas non plus l'indexation au niveau des pôles ou de la ligne de date internationale.

Le besoin du support des autres types de géométries se fit ressentir lorsqu'il s'agissait de répondre à des questions du type "A quelle distance la ligne de vol d'un avion Los Angeles/Paris passe-t-elle de l'Islande ?"



Répondre à cette question en travaillant avec un plan cartésien fournit une très mauvaise réponse en effet ! En utilisant la ligne rouge, nous obtenons une bien meilleure réponse. Si nous convertissons notre vol LAX-CDG en une ligne et que nous calculons la distance à un point en Islande, nous obtiendrons la réponse exacte, en mètres.

```
SELECT ST_Distance(
  ST_GeographyFromText('LINESTRING(-118.4079 33.9434, 2.5559 49.0083)'), -- LAX-CDG
  ST_GeographyFromText('POINT(-21.8628 64.1286)')                        -- Iceland
);
```

```
531773.757079116
```

Donc le point le plus proche de l'Islande pendant le vol LAX-CDG est de 532 kilomètres.

L'approche cartésienne pour manipuler les coordonnées géographiques perd tout son sens pour les objets situés au dessus de la ligne de date internationale. La route "sphérique" la plus courte entre Los-Angeles et Tokyo traverse l'océan Pacifique. La route "cartésienne" la plus courte traverse quant à elle les océans Atlantique et Indien.


```
SELECT ST_Distance(
  ST_GeometryFromText('Point(-118.4079 33.9434)'),  -- LAX
```

```
  ST_GeometryFromText('Point(139.733 35.567)'))      -- NRT (Tokyo/Narita)
    AS geometry_distance,
ST_Distance(
  ST_GeographyFromText('Point(-118.4079 33.9434)'), -- LAX
  ST_GeographyFromText('Point(139.733 35.567)'))     -- NRT (Tokyo/Narita)
    AS geography_distance;

 geometry_distance | geography_distance
-------------------+--------------------
  258.146005837336 |   8833954.76996256
```

18.1 Utiliser le type 'Geography'

Afin d'importer des données dans une table de type `geography`, les objets géographiques doivent d'abord être projetés dans le système EPSG :4326 (longitude/latitude), ensuite ils doivent être convertis en objets de type `geography`. La fonction **ST_Transform(geometry,srid)** convertit les coordonnées en `geography` et la fonction **Geography(geometry)** change le type ("cast") de géométrie à géographie.

```
CREATE TABLE nyc_subway_stations_geog AS
SELECT
  Geography(ST_Transform(the_geom,4326)) AS geog,
  name,
  routes
FROM nyc_subway_stations;
```

La construction d'une indexation spatiale sur une table stockant des objets de type `geography` est exactement identique à la méthode employée pour les géométries :

```
CREATE INDEX nyc_subway_stations_geog_gix
ON nyc_subway_stations_geog USING GIST (geog);
```

La différence est camouflée : l'indexation des objets de type `geography` gère correctement les requêtes qui recouvrent les pôles ou traversent les fuseaux horaires, alors que les géométries ne le supporteront pas.

Il n'y a qu'un petit nombre de fonctions disponibles pour le type `geography` :
- **ST_AsText(geography)** retourne la représentation `textuelle`
- **ST_GeographyFromText(text)** retourne un objet de type `geography`
- **ST_AsBinary(geography)** retourne la représentation binaire `bytea`
- **ST_GeogFromWKB(bytea)** retourne un objet de type `geography`
- **ST_AsSVG(geography)** retourne `text`
- **ST_AsGML(geography)** retourne `text`
- **ST_AsKML(geography)** retourne `text`
- **ST_AsGeoJson(geography)** retourne `text`
- **ST_Distance(geography, geography)** retourne `double`
- **ST_DWithin(geography, geography, float8)** retourne `boolean`
- **ST_Area(geography)** retourne `double`
- **ST_Length(geography)** retourne `double`
- **ST_Covers(geography, geography)** retourne `boolean`
- **ST_CoveredBy(geography, geography)** retourne `boolean`
- **ST_Intersects(geography, geography)** retourne `boolean`

– **ST_Buffer(geography, float8)** retourne `geography` [1]
– **ST_Intersection(geography, geography)** retourne `geography` [1]

18.2 Création d'une table stockant des géographies

Le code SQL permettant la création d'une nouvelle table avec une colonne de type `geography` ressemble à la création d'une table stockant des géométries. Cependant, les objets de type `geography` permettent de spécifier directement le type d'objet géographique à la création de la table. Par exemple :

```
CREATE TABLE airports (
  code VARCHAR(3),
  geog GEOGRAPHY(Point)
);

INSERT INTO airports VALUES ('LAX', 'POINT(-118.4079 33.9434)');
INSERT INTO airports VALUES ('CDG', 'POINT(2.5559 49.0083)');
INSERT INTO airports VALUES ('REK', 'POINT(-21.8628 64.1286)');
```

Lors de la définition le type `GEOGRAPHY(Point)` spécifie que nos aéroports sont des points. Le nouveau champ géographie n'est pas référencé dans la table `geometry_columns`. Le stockage des métadonnées relatives aux données de type `geography` s'effectue dans une vue appelée `geography_columns` qui est maintenue à jour automatiquement sans avoir besoin d'utiliser des fonctions comme `geography_columns`.

```
SELECT * FROM geography_columns;

          f_table_name          | f_geography_column | srid |   type
--------------------------------+--------------------+------+----------
 nyc_subway_stations_geography | geog               |    0 | Geometry
 airports                       | geog               | 4326 | Point
```

Note : La possibilité de définir les types et le SRID lors de la création de la table (requête `CREATE`), et la mise à jour automatique des métadonnées `geometry_columns` sont des fonctionalités qui seront adaptées pour le type géométrie pour la version 2.0 de PostGIS.

18.3 Conversion de type

Bien que les fonctions de base qui s'appliquent au type `geography` puissent être utilisées dans un grand nombre de cas d'utilisation, il est parfois nécessaire d'accéder aux autres fonctions qui ne supportent que le type géométrie. Heureusement, il est possible de convertir des objets de type géométrie en des objets de types géographie et inversement.

1. Les fonctions buffer et intersection sont actuellement construites sur le principe de conversion de type en géométries, et ne sont pas actuellement capable de gérer des coordonnées sphériques. Il en résulte qu'elles peuvent ne pas parvenir à retourner un résultat correcte pour des objets ayant une grande étendue qui ne peut être représenté correctement avec une représentation planaire.

Par exemple, la fonction **ST_Buffer(geography,distance)** transforme les objets géographiques dans la "meilleure" projection, crée la zone tampon, puis les transforme à nouveau en des géographies. S'il n'y a pas de "meilleure" projection (l'objet est trop vaste), l'opération peut ne pas réussir à retourner une valeur correcte ou retourner un tampon mal formé.

La syntaxe habituelle de PostgreSQL pour les conversion de type consiste à ajouter à la valeur la chaîne suivante `::typename`. Donc, `2::text` convertit la valeur numérique deux en une chaîne de caractères '2'. La commande : `'POINT(0 0)'::geometry` convertira la représentation textuelle d'un point en une point géométrique.

La fonction **ST_X(point)** supporte seulement le type géométrique. Comment lire la coordonnée X d'une de nos géographie ?

```
SELECT code, ST_X(geog::geometry) AS longitude FROM airports;
```

```
 code | longitude
------+-----------
 LAX  | -118.4079
 CDG  |    2.5559
 REK  |  -21.8628
```

En ajoutant la chaîne `::geometry` à notre valeur géographique, nous la convertissons en une géographie ayant le SRID : 4326. À partir de maintenant, nous pouvons utiliser autant de fonctions s'appliquant aux géométries que nous le souhaitons. Mais, souvenez-vous - maintenant que nos objets sont des géométries, leur coordonnées seront interprétées comme des coordonnées cartésiennes, non pas sphériques.

18.4 Pourquoi (ne pas) utiliser les géographies

Les géographies ont des coordonnées universellement acceptées - chacun peut comprendre que représente la latitude et la longitude, mais peu de personne comprennent ce que les coordonnées UTM signifient. Pourquoi ne pas tout le temps utiliser des géographies ?

- Premièrement, comme indiqué précédemment, il n'y a que quelques fonctions qui supportent ce type de données. Vous risquez de perdre beaucoup de temps à contourner les problèmes liés à la non-disponibilité de certaines fonctions.
- Deuxièmement, les calculs sur une sphère sont plus consomateurs en ressource que les mêmes calculs dans un système cartésien. Par exemple, la formule de calcul de distance (Pythagore) entraîne un seul appel à la fonction racine carré (sqrt()). La formule de calcul de distance sphérique (Haversine) utilise deux appels à la fonction racine carré, et un appel à arctan(), quatre appels à sin() et deux à cos(). Les fonctions trigonométriques sont très coûteuses, et les calculs sphériques les utilisent massivement.

Quel conclusion en tirer ?

Si vos données sont géographiquement compactes (contenu à l'intérieur d'un état, d'un pays ou d'une ville), utilisez le type `geometry` avec une projection cartésienne qui est pertinente pour votre localisation. Consultez le site http ://spatialreference.org et tapez le nom de votre région pour visualiser la liste des systèmes de projection applicables dans votre cas.

Si, d'un autre coté, vous avez besoin de calculer des distances qui sont géographiquement éparses (recouvrant la plupart du monde), utilisez le type `geography`. La complexité de l'application évitée en travaillant avec des objets de type `geography` dépassera les problèmes de performances. La conversion de type en géométrie permettra de dépasser les limites des fonctionnalités proposées pour ce type.

18.5 Liste des fonctions

ST_Distance(geometry, geometry) : Pour le type géométrie, renvoie la distance cartésienne, pour les géographies la distance sphérique en mètres.

ST_GeographyFromText(text) : Retourne la valeur géographique à partir d'une représentation en WKT ou EWKT.

ST_Transform(geometry, srid) : Retourne une nouvelle géométrie avec ses coordonnées reprojetées dans le système de référence spatial référencé par le SRID fourni.

ST_X(point) : Retourne la coordonnée X d'un point, ou NULL si non disponible. La valeur passée doit être un point.

CHAPITRE 19

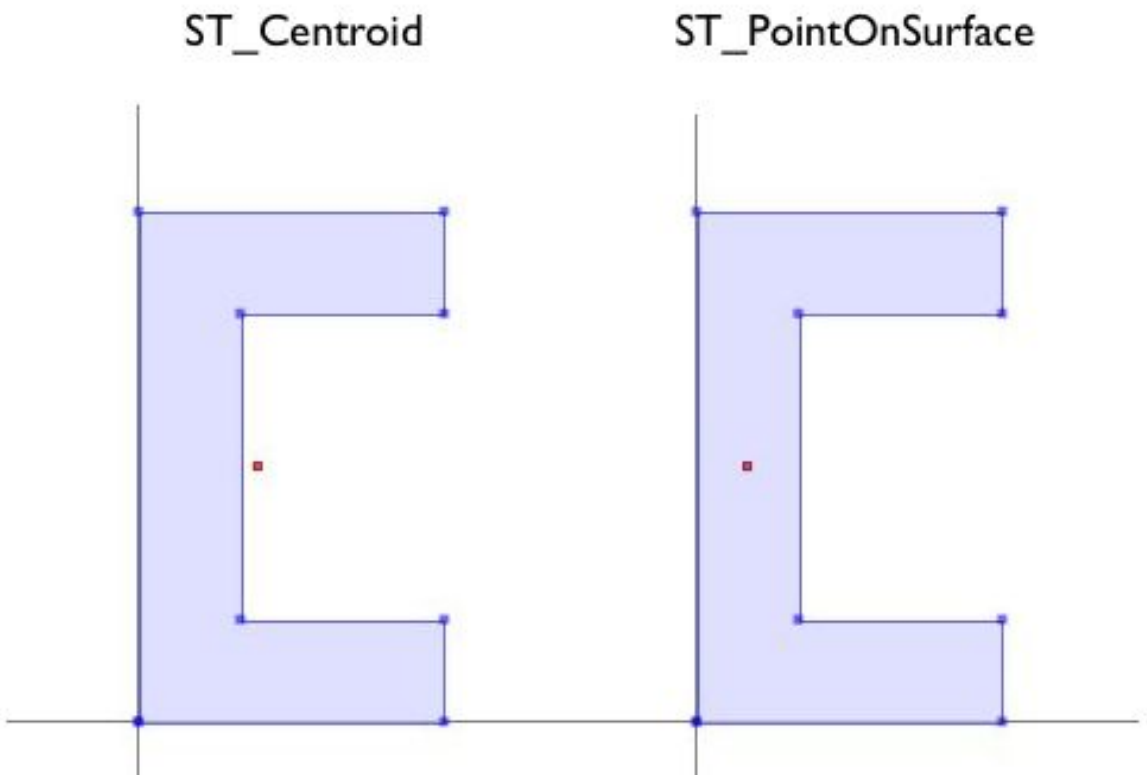
Partie 18 : Fonctions de construction de géométries

Toute les fonctions que nous avons vu jusqu'à présent traitent les géométries "comme elles sont" et retournent :
- une analyse des objets (**ST_Length(geometry)**, **ST_Area(geometry)**),
- une sérialisation des objets (**ST_AsText(geometry)**, **ST_AsGML(geometry)**),
- une partie de l'objet (**ST_RingN(geometry,n)**) ou
- un résultat vrai/faux (**ST_Contains(geometry,geometry)**, **ST_Intersects(geometry,geometry)**).

Les "fonctions de construction de géométries" prennent des géométries en entrée et retourne de nouvelles formes.

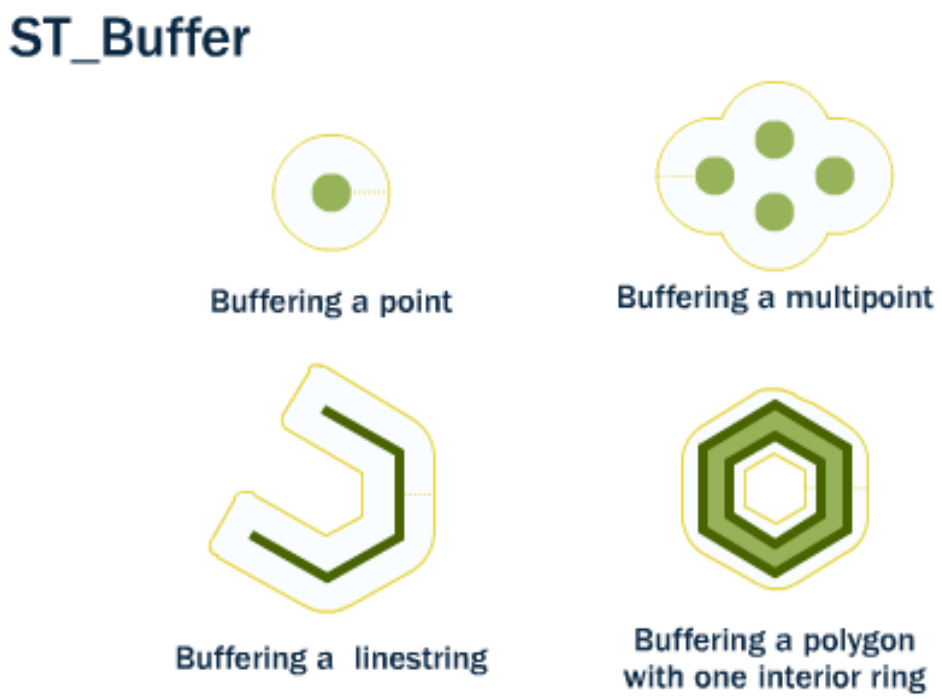
19.1 ST_Centroid / ST_PointOnSurface

Un besoin commun lors de la création de requêtes spatiales est de remplacer une entité polygonale par un point représentant cette entité. Cela est utile pour les jointures spatiales (comme indiqué ici : *Polygones/Jointures de polygones*) car utiliser **ST_Intersects(geometry,geometry)** avec deux polygones impliquera un double comptage : un polygone pour le contour externe intersectera dans les deux sens ; le remplacer par un point le forcera à être dans un seul sens, pas les deux.
- **ST_Centroid(geometry)** retourne le point qui est approximativement au centre de la masse de la géométrie passée en paramètre. C'est un calcul simple et rapide, mais parfois non profitable, car le point retourné peut se trouver à l'extérieur de l'entité elle-même. Si l'entité fournie est convexe (imaginez la lettre 'C') le centroïde renvoyé pourrait ne pas être à l'intérieur du polygone.
- **ST_PointOnSurface(geometry)** retourne un point qui est obligatoirement dans l'entité passée en paramètre. Cette fonction coûte plus cher en ressource que le calcul du centroïde.



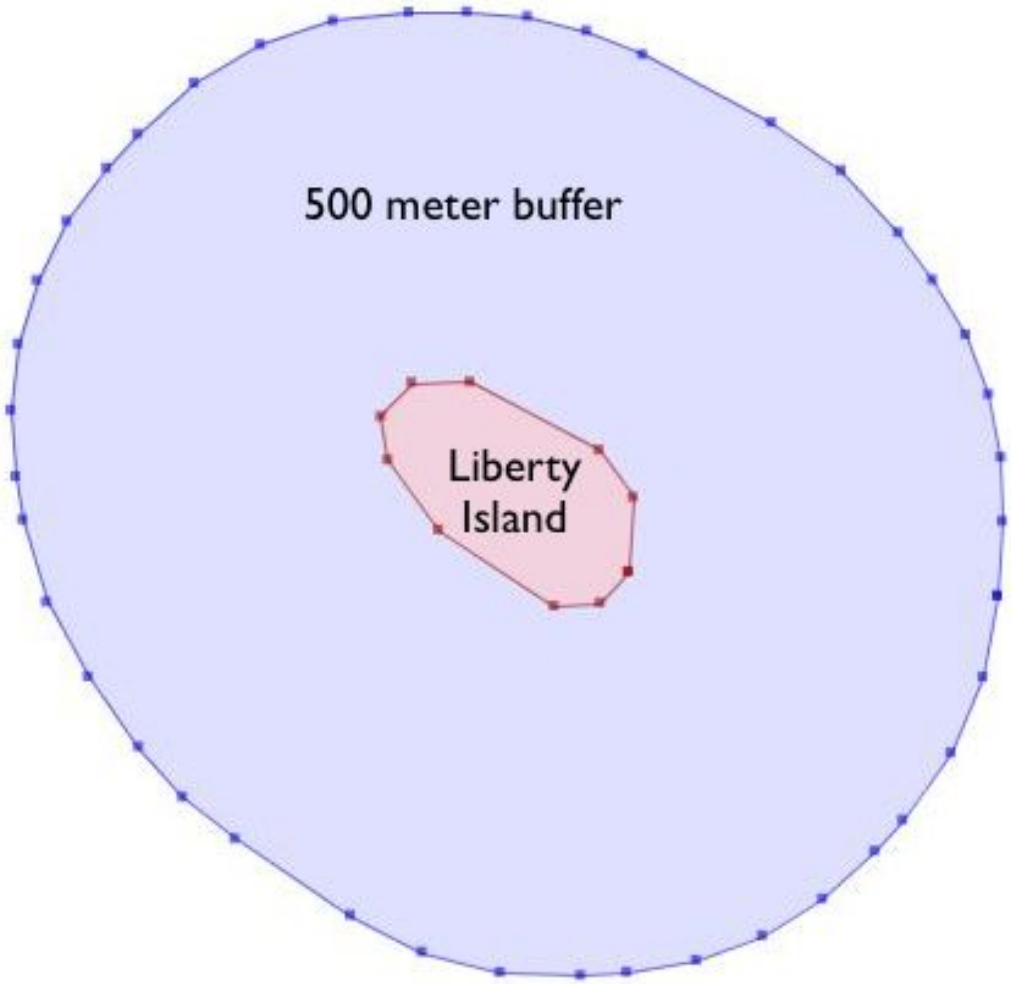
19.2 ST_Buffer

L'opération de zone tampon est souvent disponible dans les outils SIG, il est aussi disponible dans PostGIS. La fonction **ST_Buffer(geometry,distance)** prend en paramètre une géométrie et une distance et retourne une zone tampon dont le contour est à une distance donnée de la géométrie d'origine.

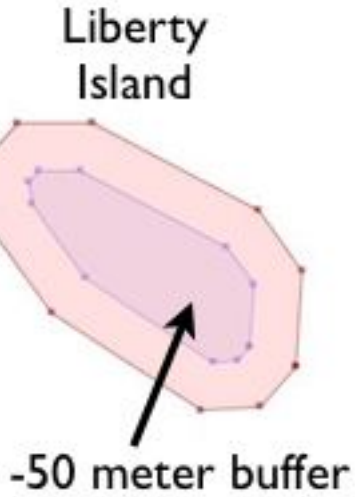


Par exemple, si les services des parcs américains souhaitaient renforcer la zone du traffic maritime autour de l'île 'Liberty', ils pourraient construire une zone tampon de 500 mètres autour de l'île. L'île de 'Liberty' est représentée par un seul bloc dans notre table `nyc_census_blocks`, nous pouvons donc facilement réaliser ce calcul.

```
-- Création d'une nouvelle table avec une zone tampon de 500 m autour de 'Liberty Islanc
CREATE TABLE libery_island_zone AS
SELECT ST_Buffer(the_geom,500) AS the_geom
FROM nyc_census_blocks
WHERE blkid = '360610001009000';

-- Mise à jour de la table geometry_columns
SELECT Populate_Geometry_Columns();
```



La fonction **ST_Buffer** permet aussi d'utiliser des valeurs négatives pour le paramètre distance et construit un polygone inclus dans celui passé en paramètre. Pour les points et les lignes vous obtiendrez simplement un résultat vide.

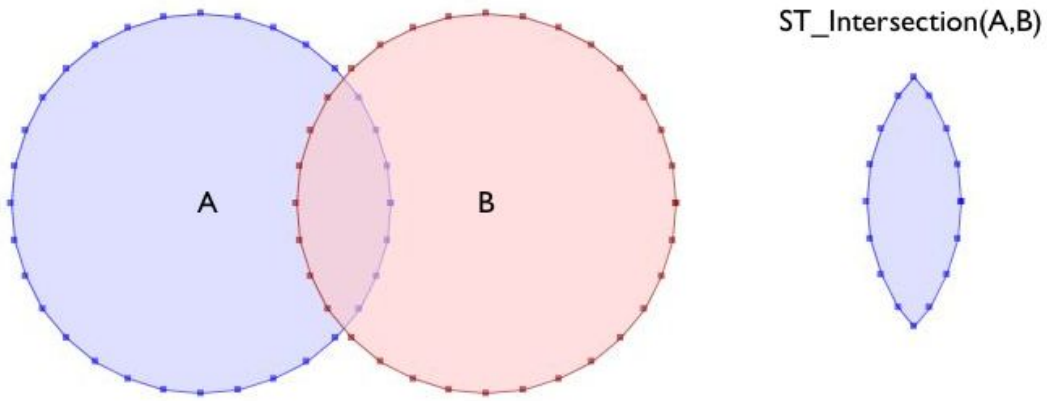


19.3 ST_Intersection

Une autre opération classique présente dans les SIG - le chevauchement - crée une nouvelle entité en calculant la zone correspondant à l'intersection de deux polygones superposés. Le résultat à la propriété de permettre de reconstruire les entités de base à l'aide de ce résultat.

La fonction **ST_Intersection(geometry A, geometry B)** retourne la zone géographique (ou une ligne, ou un point) que les deux géométries ont en commun. Si les géométries sont disjointes, la fonction retourne une géométrie vide.

```
-- Quelle est l'aire que ces deux cercles ont en commun ?
-- Utilisons la fonction ST_Buffer pour créer ces cercles !

SELECT ST_AsText(ST_Intersection(
  ST_Buffer('POINT(0 0)', 2),
  ST_Buffer('POINT(3 0)', 2)
));
```

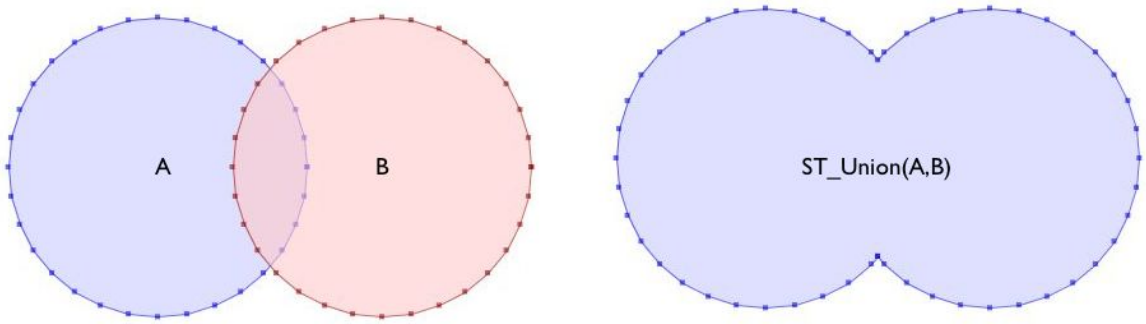


19.4 ST_Union

Dans l'exemple précédent, nous intersections des géométries, créant une nouvelle géométrie unique à partir de deux entités. La commande **ST_Union** fait l'inverse, elle prend en paramètre des géométries et supprime les parties communes. Il y a deux versions possibles de la fonction **ST_Union** :

– **ST_Union(geometry, geometry)** : une version avec deux paramètres qui prend les géométries et retourne l'union des deux. Par exemple, nos deux cercles ressemblent à ce qui suit si nous utilisons l'opération union plutôt que l'intersection.

```
-- Quelle est l'aire totale de ces deux cercles ?
-- Utilisons ST_Buffer pour créer les cercles !

SELECT ST_AsText(ST_Union(
  ST_Buffer('POINT(0 0)', 2),
  ST_Buffer('POINT(3 0)', 2)
));
```



– **ST_Union([geometry])** : une version agrégée qui prendre un ensemble de géométries et retourne une géométrie contenant l'ensemble des géométries rassemblées. La fonction agrégée ST_Union peut être utilisé grâce au SQL `GROUP BY` pour créer un ensemble rassemblant des sous-ensembles de géométries basiques. Cela est très puissant.

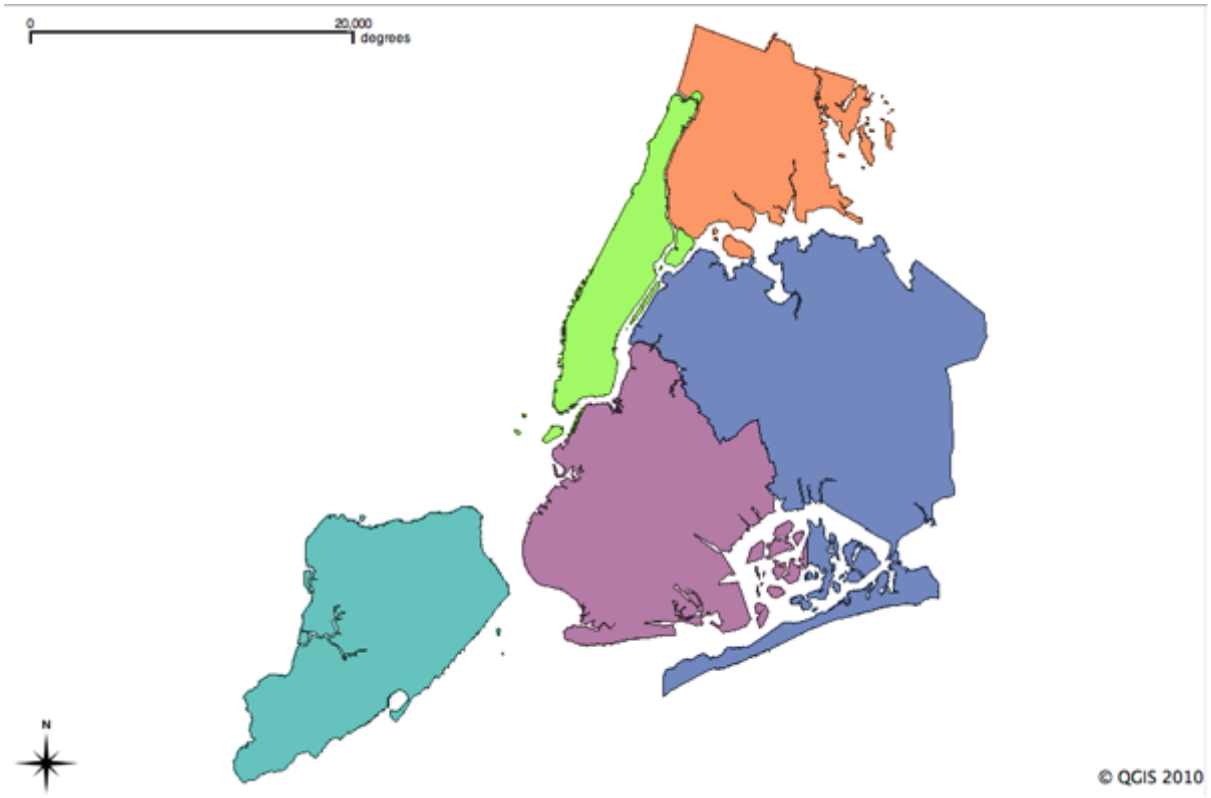
Comme exemple pour la fonction d'agrégation **ST_Union**, considérons notre table `nyc_census_blocks`. Les géographie du recensement sont construites de manière à ce qu'on puisse créer d'autres géographies à partir des premières. ainsi, nous pouvons créer une carte des secteurs de recensement en fusionnant les blocs que forme chaque secteur (comme nous le ferons après dans *la création des tables secteurs*). Ou, nous pouvons créer une carte du comté en fusionnant les blocs qui relèvent de chaque comté.

Pour effectuer la fusion, notez que la clé unique `blkid` incorpore des informations sur les géographies de niveau supérieur. Voici les parties de la clé pour Liberty Island que nous avons utilisé précédemment.

```
360610001009000 = 36 061 00100 9000

36     = State of New York
061    = New York County (Manhattan)
000100 = Census Tract
9      = Census Block Group
000    = Census Block
```

Ainsi, nous pouvons créer une carte du comté en fusionnant toutes les géométries qui partagent les 5 premiers chiffres de `blkid`.

```
-- Création d'une table nyc_census_counties en regroupant les blocs
CREATE TABLE nyc_census_counties AS
SELECT
  ST_Union(the_geom) AS the_geom,
  SubStr(blkid,1,5) AS countyid
FROM nyc_census_blocks
GROUP BY countyid;

-- Mise à jour de la table geometry_columns
SELECT Populate_Geometry_Columns();
```



Un test de surface peut confirmer que notre opération d'union n'a pas perdu de géométries. Tout d'abord, nous calculons la surface de chacun des blocs de recensement et faisons la somme de ces surfaces en les groupant par l'identifiant de recensement des comtés.

```
SELECT SubStr(blkid,1,5) AS countyid, Sum(ST_Area(the_geom)) AS area
FROM nyc_census_blocks
GROUP BY countyid;
```

```
 countyid |        area
----------+------------------
 36005    | 109807439.720947
 36047    | 184906575.839355
 36061    | 58973521.6225586
 36081    | 283764734.207275
 36085    | 149806077.958252
```

Ensuite nous calculons l'aire de chaque zone de nos nouveaux polygones de région de la table count :

```
SELECT countyid, ST_Area(the_geom) AS area
FROM nyc_census_counties;
```

```
 countyid |        area
----------+------------------
 36005    | 109807439.720947
 36047    | 184906575.839355
 36061    | 58973521.6225586
 36081    | 283764734.207275
 36085    | 149806077.958252
```

La même réponse ! Nous avons construit avec succès une table des régions de NYC à partir de nos données initiales.

19.5 Liste des fonctions

ST_AsText(text) : retourne la représentation Well-Known Text (WKT) de la géométrie/géographie sans métadonnée SRID.

ST_Buffer(geometry, distance) : Pour les géométries : retourne une géométrie qui représente tous les points dont la distance depuis cette géométrie est inférieure ou égale à la distance utilisée. Les calculs se font dans le système de référence spatial de cette géométrie. Pour les géographies : utilise une fonction de transformation planaire pour effectuer le calcul.

ST_Intersection(geometry A, geometry B) : retourne une géométrie qui représente la portion commune des géométries A et B. L'implémentation du type géographie fait une transformation vers une géométrie pour faire l'intersection puis reprojette le résultat en WGS84.

ST_Union() : Renvoie un objet géométrique qui représente l'ensemble d'union des objets géométriques désignés.

substring(string [from int] [for int]) : Fonction de chaîne PostgreSQL pour extraire une sous-chaîne de caractères.

sum(expression) : Fonction d'agrégation PostgreSQL qui retourne la somme des valeurs d'une colonne dans un ensemble d'enregistrements.

CHAPITRE 20

Partie 19 : Plus de jointures spatiales

Dans la partie précédente nous avons vu les fonctions **ST_Centroid(geometry)** et **ST_Union(geometry)** ainsi que quelques exemples simples. Dans cette partie nous réaliserons des choses plus élaborées.

20.1 Création de la table de traçage des recensements

Dans le répertoire `\data\` des travaux pratiques, il y a un fichier qui contient des données attributaires, mais pas de géométries, ce fichier est nommé `nyc_census_sociodata.sql`. La table contient des données sociaux-économiques intéressantes à propos de New York : revenus financiers, éducation .... Il y a juste un problème, les données sont rassemblées en "trace de recensement" et nous n'avons pas de données spatiales associées !

Dans cette partie nous allons
- Charger la table `nyc_census_sociodata.sql`
- Créer une table spatiale pour les traces de recensement
- Joindre les données attributaires à nos données spatiales
- Réaliser certaines analyses sur nos nouvelles données

20.1.1 Chargement du fichier nyc_census_sociodata.sql

1. Ouvrez la fenêtre de requêtage SQL depuis PgAdmin
2. Sélectionnez **File->Open** depuis le menu et naviguez jusqu'au fichier `nyc_census_sociodata.sql`
3. Cliquez sur le bouton "Run Query"
4. Si vous cliquez sur le bouton "Refresh" depuis PgAdmin, la liste des tables devrait contenir votre nouvelle table `nyc_census_sociodata`

20.1.2 Création de la table traces de recensement

Comme nous l'avons fait dans la partie précédente, nous pouvons construire des géométries de niveau suppérieur en utilisant nos blocs de base en utilisant une partie de la clef `blkid`. Afin de calculer les traces de recensement, nous avons besoin de regrouper les blocs en uitlisant les 11 premiers caractères de la colonne `blkid`.

```
360610001009000 = 36 061 00100 9000

36     = State of New York
061    = New York County (Manhattan)
000100 = Census Tract
9      = Census Block Group
000    = Census Block
```

Création de la nouvelle table en utilisant la fonction d'agrégation **ST_Union** :

```
-- Création de la table
CREATE TABLE nyc_census_tract_geoms AS
SELECT
  ST_Union(the_geom) AS the_geom,
  SubStr(blkid,1,11) AS tractid
FROM nyc_census_blocks
GROUP BY tractid;

-- Indexation du champ tractid
CREATE INDEX nyc_census_tract_geoms_tractid_idx ON nyc_census_tract_geoms (tractid);

-- Mise à jour de la table geometry_columns
SELECT Populate_Geometry_Columns();
```

20.1.3 Regrouper les données attributaires et spatiales

L'objectif est ici de regrouper les données spatiales que nous avons créé avec les données attributaires que nous avions chargé initialement.

```
-- Création de la table
CREATE TABLE nyc_census_tracts AS
SELECT
  g.the_geom,
  a.*
FROM nyc_census_tract_geoms g
JOIN nyc_census_sociodata a
ON g.tractid = a.tractid;

-- Indexation des géométries
CREATE INDEX nyc_census_tract_gidx ON nyc_census_tracts USING GIST (the_geom);

-- Mise à jour de la table geometry_columns
SELECT Populate_Geometry_Columns();
```

20.1.4 Répondre à une question intéressante

Répondre à une question intéressante ! "Lister les 10 meilleurs quartiers ordonnés par la proportion de personnes ayant acquis un diplôme".

```
SELECT
  Round(100.0 * Sum(t.edu_graduate_dipl) / Sum(t.edu_total), 1) AS graduate_pct,
  n.name, n.boroname
FROM nyc_neighborhoods n
JOIN nyc_census_tracts t
```

```
ON ST_Intersects(n.the_geom, t.the_geom)
WHERE t.edu_total > 0
GROUP BY n.name, n.boroname
ORDER BY graduate_pct DESC
LIMIT 10;
```

Nous sommons les statistiques qui nous intéressent, nous les divisons ensuite à la fin. Afin d'éviter l'erreur de non-division par zéro, nous ne prenons pas en compte les quartiers qui n'ont aucune personne ayant obtenu un diplôme.

```
 graduate_pct |       name        | boroname
--------------+-------------------+-----------
         40.4 | Carnegie Hill     | Manhattan
         40.2 | Flatbush          | Brooklyn
         34.8 | Battery Park      | Manhattan
         33.9 | North Sutton Area | Manhattan
         33.4 | Upper West Side   | Manhattan
         33.3 | Upper East Side   | Manhattan
         32.0 | Tribeca           | Manhattan
         31.8 | Greenwich Village | Manhattan
         29.8 | West Village      | Manhattan
         29.7 | Central Park      | Manhattan
```

20.2 Polygones/Jointures de polygones

Dans notre requête intéressante (dans *Répondre à une question intéressante*) nous avons utilisé la fonction **ST_Intersects(geometry_a, geometry_b)** pour déterminer quelle entité polygonale à inclure dans chaque groupe de quartier. Ce qui nous conduit à la question : que ce passe-t-il si une entité tombe entre deux quartiers ? Il intersectera chacun d'entre eux et ainsi sera inclut dans **chacun** des résultats.

Pour éviter ce cas de double comptage il existe trois méthodes :
- La méthode simple consiste a s'assurer que chaque entité ne se retrouve que dans **un** seul groupe géographique (en utilisant **ST_Centroid(geometry)**)
- La méthode complexe consiste à disviser les parties qui se croisent en utilisant les bordures (en utilisant **ST_Intersection(geometry,geometry)**)

Voici un exemple d'utilisation de la méthode simple pour éviter le double comptage dans notre requête précédente :

```
SELECT
  Round(100.0 * Sum(t.edu_graduate_dipl) / Sum(t.edu_total), 1) AS graduate_pct,
  n.name, n.boroname
FROM nyc_neighborhoods n
JOIN nyc_census_tracts t
ON ST_Contains(n.the_geom, ST_Centroid(t.the_geom))
WHERE t.edu_total > 0
GROUP BY n.name, n.boroname
ORDER BY graduate_pct DESC
LIMIT 10;
```

Remarquez que la requête prend plus de temps à s'exécuter, puisque la fonction **ST_Centroid** doit être effectuée pour chaque entité.

```
 graduate_pct |       name        | boroname
--------------+-------------------+-----------
         49.2 | Carnegie Hill     | Manhattan
         39.5 | Battery Park      | Manhattan
         34.3 | Upper East Side   | Manhattan
         33.6 | Upper West Side   | Manhattan
         32.5 | Greenwich Village | Manhattan
         32.2 | Tribeca           | Manhattan
         31.3 | North Sutton Area | Manhattan
         30.8 | West Village      | Manhattan
         30.1 | Downtown          | Brooklyn
         28.4 | Cobble Hill       | Brooklyn
```

Éviter le double comptage change le résultat !

20.3 Jointures utilisant un large rayon de distance

Une requête qu'il est "sympa" de demander est : "Comment les temps de permutation des gens proches (dans un rayon de 500 mètres) des stations de métro diffèrent de ceux qui en vivent loin ? "

Néanmoins, la question rencontre les mêmes problèmes de double comptage : plusieurs personnes seront dans un rayon de 500 mètres de plusieurs stations de métro différentes. Comparons la population de New York :

```
SELECT Sum(popn_total)
FROM nyc_census_blocks;
```

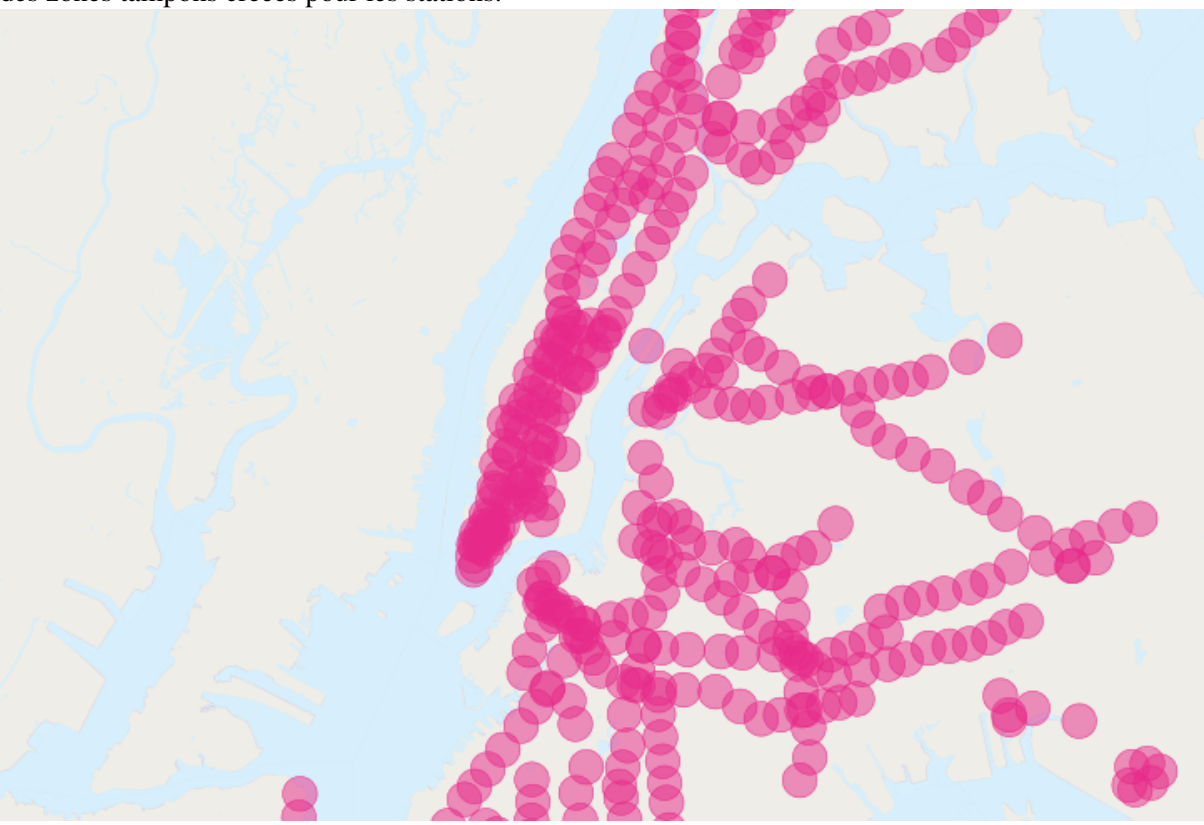
```
8008278
```

Avec la population des gens de New York dans un rayon de 500 mètres d'une station de métro :

```
SELECT Sum(popn_total)
FROM nyc_census_blocks census
JOIN nyc_subway_stations subway
ON ST_DWithin(census.the_geom, subway.the_geom, 500);
```

```
10556898
```

Il y a plus de personnes proches du métro qu'il y a de personnes ! Clairement, notre requête SQL simple rencontre un gros problème de double comptage. Vous pouvez voir le problème en regardant l'image des zones tampons créées pour les stations.



La solution est de s'assurer que nous avons seulement des blocs distincts avant de les regrouper. Nous pouvons réaliser cela en cassant notre requête en sous-requêtes qui récupèrent les blocs distincts, les regroupent pour ensuite retourner notre réponse :

```
SELECT Sum(popn_total)
FROM (
  SELECT DISTINCT ON (blkid) popn_total
  FROM nyc_census_blocks census
  JOIN nyc_subway_stations subway
  ON ST_DWithin(census.the_geom, subway.the_geom, 500)
) AS distinct_blocks;
```

```
4953599
```

C'est mieux ! Donc un peu plus de 50 % de la population de New York vit à proximité (500m, environ 5 à 7 minutes de marche) du métro.

CHAPITRE 21

Partie 20 : Validité

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

21.1 Qu'est-ce que la validité ?

La validité est surtout importante pour les polygones, qui définissent des aires et requièrent une bonne structuration. Les lignes sont vraiment simples et ne peuvent pas être invalides ainsi que les points.

Certaines des règles de validation des polygones semble évidentes, et d'autre semblent arbitraires (et le sont vraiment).
- Les contours des polygones doivent être fermés.
- Les contours qui définissent des trous doivent être inclus dans la zone définie par le contour extérieur.
- Les contours ne doivent pas s'intersecter (ils ne doivent ni se croiser ni se toucher).
- Les contours ne doivent pas toucher les autres contours, sauf en un point unique.

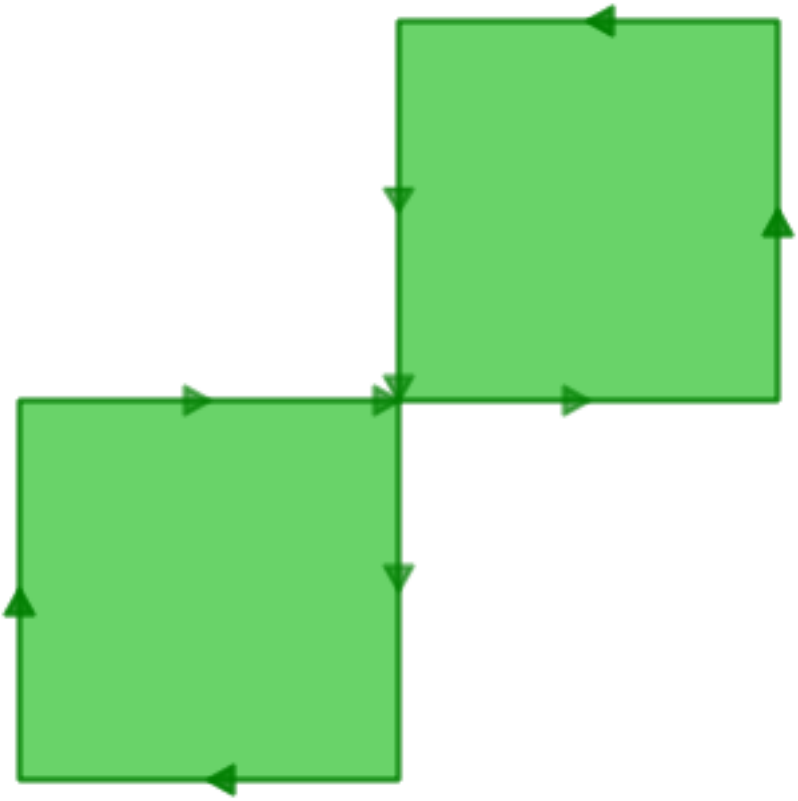
Les deux dernières règles font partie de la catégorie arbitraire. Il y a d'autres moyens de définir des polygones qui sont consistants mais les règles ci-dessus sont celles utilisées dans le standard *OGC SFSQL* que respecte PostGIS.

La raison pour laquelle ces règles sont importantes est que les algorithmes de calcul dépendent de cette structuration consistante des arguments. Il est possible de construire des algorithmes qui n'utilisent pas cette structuration, mais ces fonctions tendent à être très lentes, étant donné que la première étape consiste à "analyser et construire des structures à l'intérieur des données".

Voici un exemple de pourquoi cette structuration est importante. Ce polygone n'est pas valide :

```
POLYGON((0 0, 0 1, 2 1, 2 2, 1 2, 1 0, 0 0));
```

Vous pouvez comprendre ce qui n'est pas valide en regardant cette figure :



Le contour externe est exactement en forme en 8 avec une intersection au milieu. Notez que la fonction de rendu graphique est tout de même capable d'en afficher l'intérieur, donc visuellement cela ressemble bien à une "aire" : deux unités carré, donc une aire couplant ces deux unités.

Essayons maintenant de voir ce que pense la base de données de notre polygone :

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

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

Que ce passe-t-il ici ? L'algorithme qui calcule l'aire suppose que les contours ne s'intersectent pas. Un contour normal devra toujours avoir une aire qui est bornée (l'intérieur) dans un sens de la ligne du contour (peu importe quelle sens, juste *un* sens). Néanmoins, dans notre figure en 8, le contour externe est à droite de la ligne pour un lobe et à gauche pour l'autre. Cela entraine que les aires qui sont calculées pour chaque lobe annulent la précédente (l'une vaut 1 et l'autre -1) donc le résultat est une "aire de zéro".

21.2 Détecter la validité

Dans l'exemple précédent nous avions un polygone que nous **savions** non-valide. Comment déterminer les géométries non valides dans une tables d'un million d'enregistrements ? Avec la fonction **ST_IsValid(geometry)** utilisée avec notre polygone précédent, nous obtenons rapidement la réponse :

```
SELECT ST_IsValid(ST_GeometryFromText('POLYGON((0 0, 0 1, 1 1, 2 1, 2 2, 1 2, 1 1, 1 0,
```

```
f
```

Maintenant nous savons que l'entité est non-valide mais nous ne savons pas pourquoi. Nous pouvons utiliser la fonction **ST_IsValidReason(geometry)** pour trouver la cause de non validité :

```
SELECT ST_IsValidReason(ST_GeometryFromText('POLYGON((0 0, 0 1, 1 1, 2 1, 2 2, 1 2, 1 1,
```

```
Self-intersection[1 1]
```

Vous remarquerez qu'en plus de la raison (intersection) la localisation de la non validité (coordonnée (1 1)) est aussi renvoyée.

Nous pouvons aussi utiiliser la fonction **ST_IsValid(geometry)** pour tester nos tables :

```
-- Trouver tous les polygones non valides et leur problème
SELECT name, boroname, ST_IsValidReason(the_geom)
FROM nyc_neighborhoods
WHERE NOT ST_IsValid(the_geom);
```

```
          name            |   boroname    |                      st_isvalidreason
--------------------------+---------------+---------------------------------------------
 Howard Beach             | Queens        | Self-intersection[597264.083368305 4499924.54
 Corona                   | Queens        | Self-intersection[595483.058764138 4513817.95
 Steinway                 | Queens        | Self-intersection[593545.572199759 4514735.20
 Red Hook                 | Brooklyn      | Self-intersection[584306.820375986 4502360.51
```

21.3 Réparer les invalides

Commençons par la mauvaise nouvelle : il n'y a aucune garantie de pouvoir corriger une géométrie non valide. Dans le pire des scénarios, vous pouvez utiliser la fonction **ST_IsValid(geometry)** pour identifier les entités non valides, les déplacer dans une autre table, exporter cette table et les réparer à l'aide d'un outil extérieur.

Voici un exemple de requête SQL qui déplace les géométries non valides hors de la table principale dans une table à part pour les exporter vers un programme de réparation.

```
-- Table à part des géométries non-valides
CREATE TABLE nyc_neighborhoods_invalid AS
SELECT * FROM nyc_neighborhoods
WHERE NOT ST_IsValid(the_geom);

-- Suppression de la table principale
DELETE FROM nyc_neighborhoods
WHERE NOT ST_IsValid(the_geom);
```

Un bon outil pour réparer visuellement des géométries non valide est OpenJump (http ://openjump.org) qui contient un outils de validation depuis le menu **Tools->QA->Validate Selected Layers**.

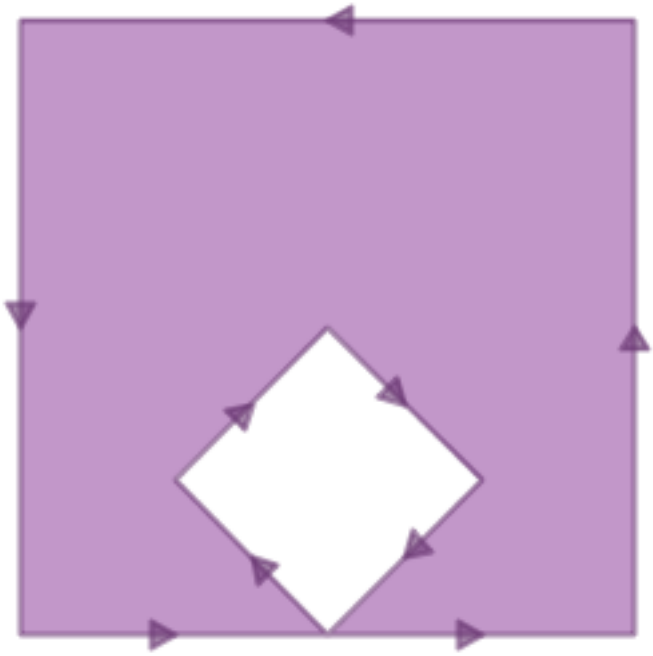
Maintenant, la bonne nouvelle : un grand nombre de non-validités **peut être résolu dans la base de données** en utilisant la fonction : **ST_Buffer**.

Le coup du Buffer tire avantage de la manière dont les buffers sont construits : une géométrie bufferisée est une nouvelle géométrie, construite en déplaçant les lignes de la géométrie d'origine. Si vous déplacez

les lignes originales par *rien* (zero) alors la nouvelle géométrie aura une structure identique à l'originale, mais puisqu'elle utilise les règles topologiques de l'*OGC*, elle sera valide.

Par exemple, voici un cas classique de non-validité - le "polygone de la banane" - un seul contour que crée une zone mais se touche, laissant un "trou" qui n'en est pas un.

```
POLYGON((0 0, 2 0, 1 1, 2 2, 3 1, 2 0, 4 0, 4 4, 0 4, 0 0))
```



En créant un buffer de zero sur le polygone retourne un polygone *OGC* valide, le contour externe et un contour interne qui touche l'autre en un seul point.

```
SELECT ST_AsText(
         ST_Buffer(
           ST_GeometryFromText('POLYGON((0 0, 2 0, 1 1, 2 2, 3 1, 2 0, 4 0, 4 4, 0 4, 0
           0.0
         )
       );
```

```
POLYGON((0 0,0 4,4 4,4 0,2 0,0 0),(2 0,3 1,2 2,1 1,2 0))
```

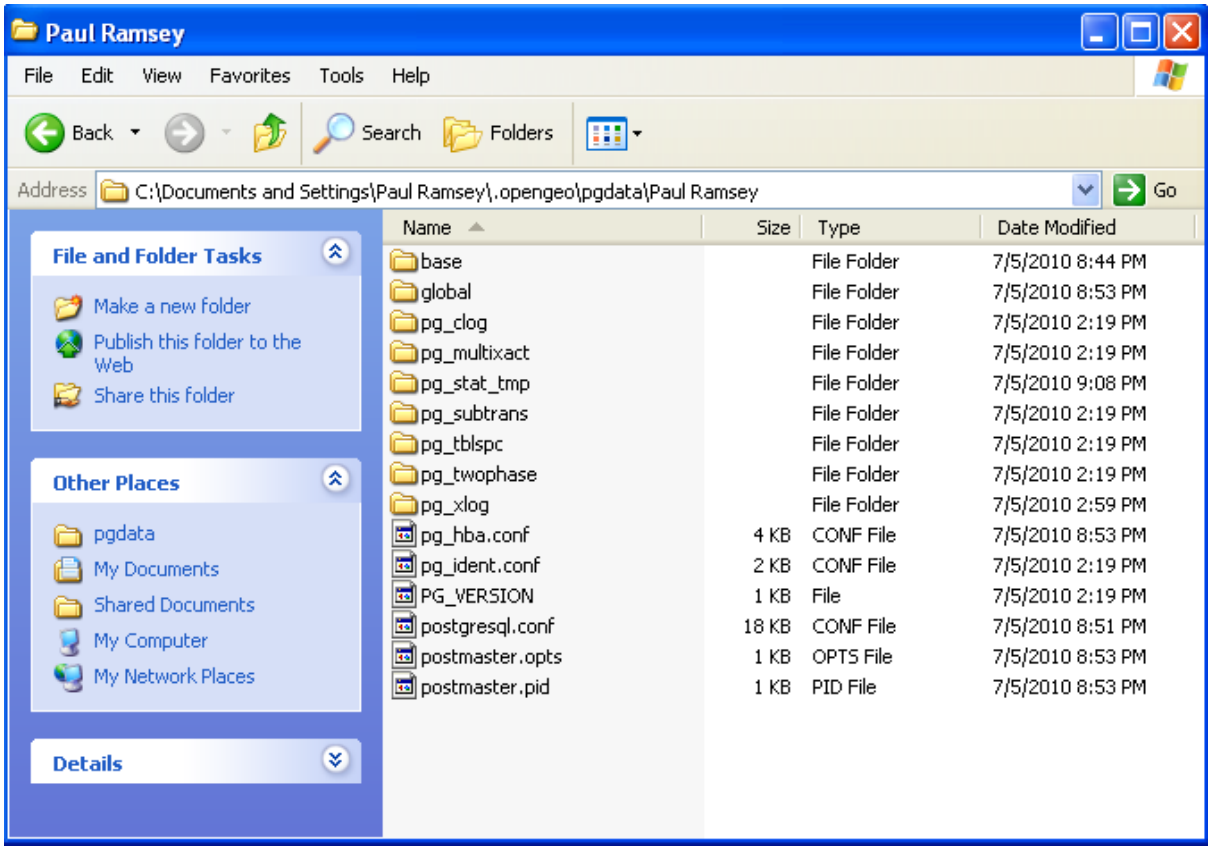
Note : Le "polygone banane" (ou "coquillage inversé") est un cas où le modèle topologique de l'*OGC* et de ESRI diffèrent. Le modèle ESRI considère que les contours qui se touchent sont non valides et préfère la forme de banane pour ce cas de figure. Le modèle de l'OGC est l'inverse.

CHAPITRE 22

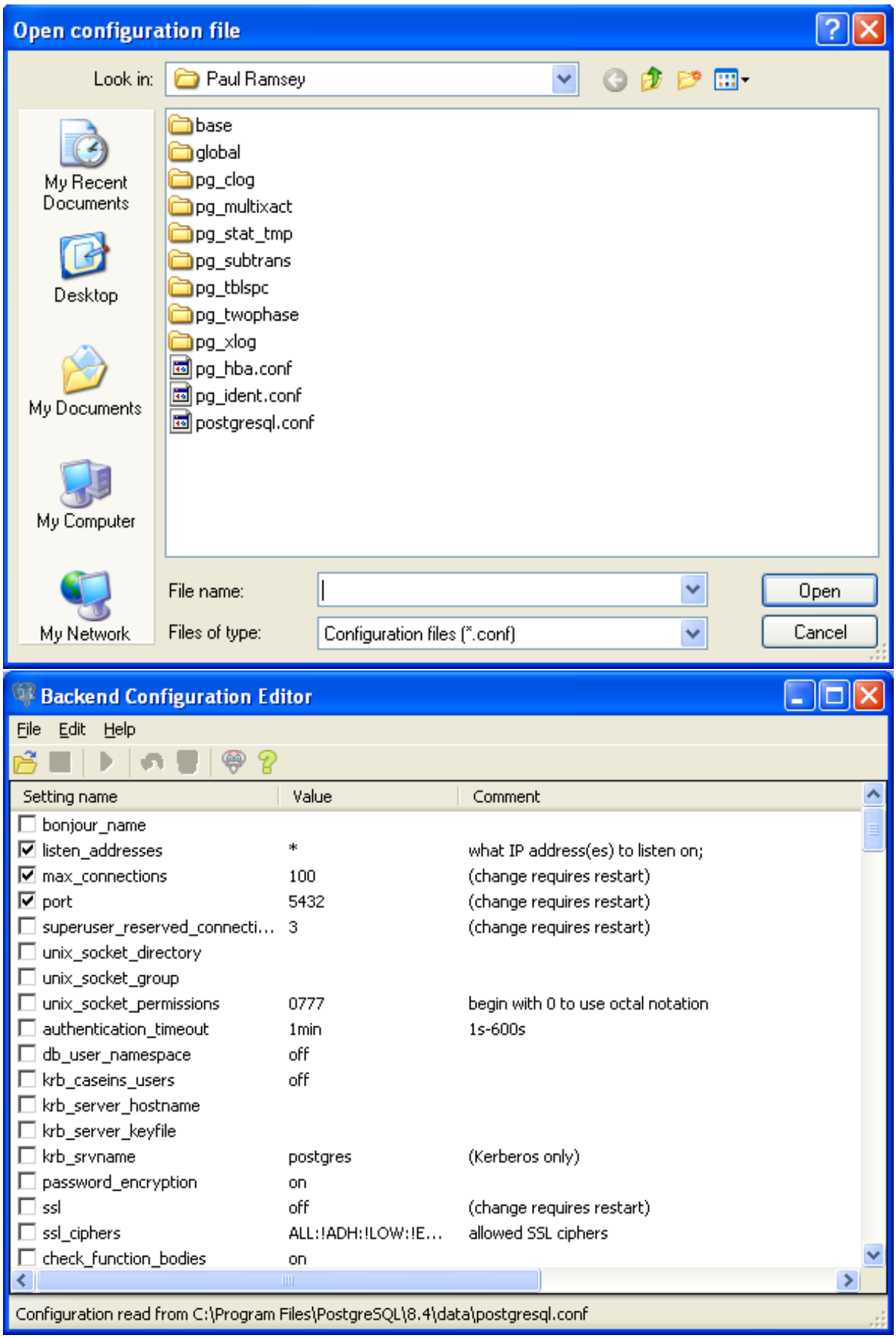
Partie 21 : Paramétrer PostgreSQL pour le spatial

PostgreSQL est une base de données très versatile, capable de tourner dans des environnements ayant des ressources très limitées et partageant ces ressources avec un grand nombre d'autres applications. Afin d'assurer qu'elle tournera convenablement dans ces environnements, la configuration par défaut est très peu consommatrice de ressources mais terriblement inadaptée pour des bases de données hautes-performances en production. Ajoutez à cela le fait que les bases de données spatiales ont différents types d'utilisation, et que les données sont généralement plus grandes que les autres types de données, vous en arriverez à la conclusion que les paramètres par défaut ne sont pas appropriés pour notre utilisation.

Tous ces paramètres de configuration peuvent être édités dans le fichier de configuration de la base de données : `C:\Documents and Settings\%USER\.opengeo\pgdata\%USER`. Le contenu du fichier est du texte et il peut donc être ouvert avec l'outil d'édition de fichiers de votre choix (Notepad par exemple). Les modifications apportées à ce fichier ne seront effectives que lors du redémarrage du serveur.



Une façon simple d'éditer ce fichier de configuration est d'utiliser l'outil nommé : "Backend Configuration Editor". Depuis pgAdmin, allez dans *File > Open postgresql.conf...*. Il vous sera demandé le chemin du fichier, naviguez dans votre arborescence jusqu'au fichier `C:\Documents and Settings\%USER\.opengeo\pgdata\%USER`.



Cette partie décrit certains des paramètres de configuration qui doivent être modifiés pour la mise ne

place d'une base de données spatiale en production. Pour chaque partie, trouvez le bon paramètre dans la liste et double cliquez dessus pour l'éditer. Changez le champ *Value* par la valeur que nous recommandons, assurez-vous que le champ est bien activé puis cliquez sur **OK**.

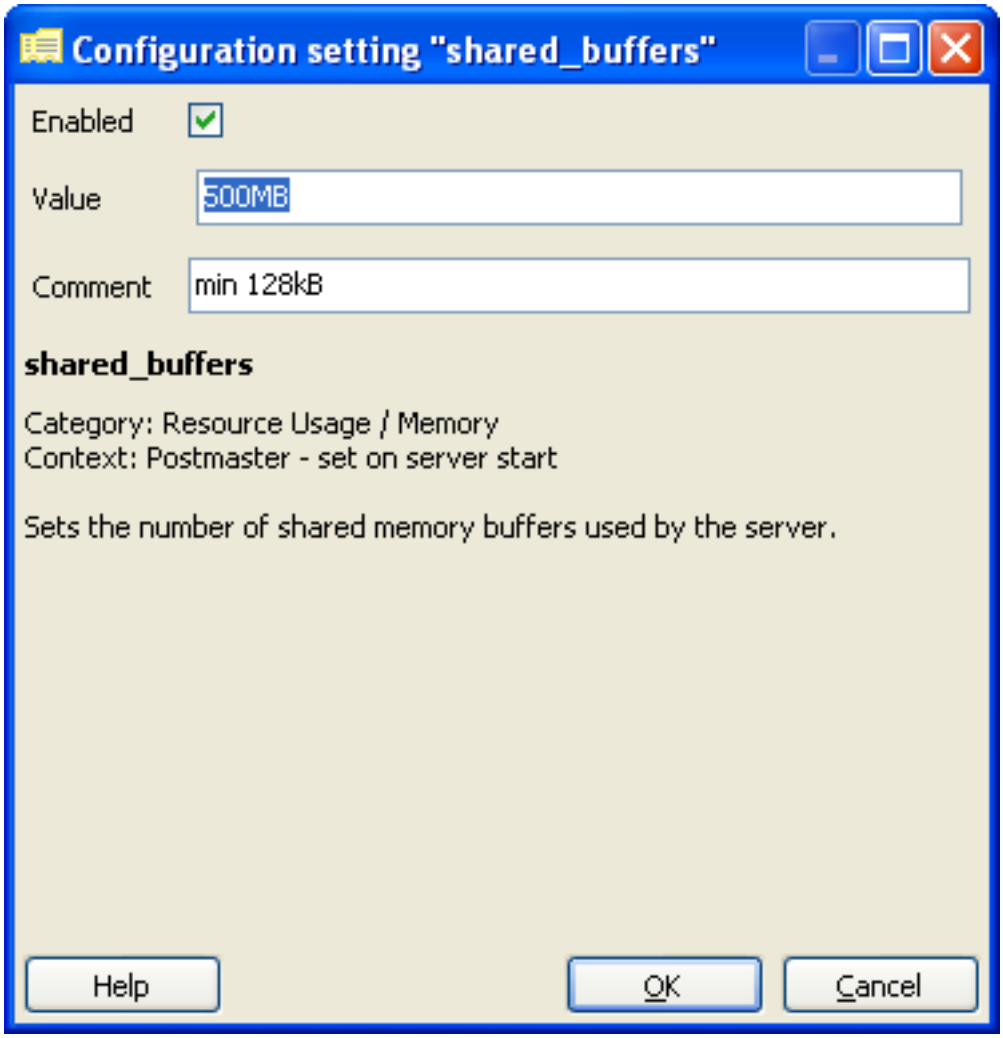
Note : Ces valeurs sont seulement celles que nous recommandons, chaque environnement diffèrera et tester les différents paramétrages est toujours nécessaire pour s'assurer d'utiliser la configuration optimale. Mais dans cette partie nous vous fournissons un bon point de départ.

22.1 shared_buffers

Alloue la quantité de mémoire que le serveur de bases de données utilise pour ses segments de mémoires partagées. Cela est partagé par tous les processus serveur, comme son nom l'indique. La valeur par défaut est affligeante et inadaptée pour une base de données en production.

Valeur par défaut : typiquement 32MB

Valeur recommandée : 75% de la mémoire de la base de données (500MB)



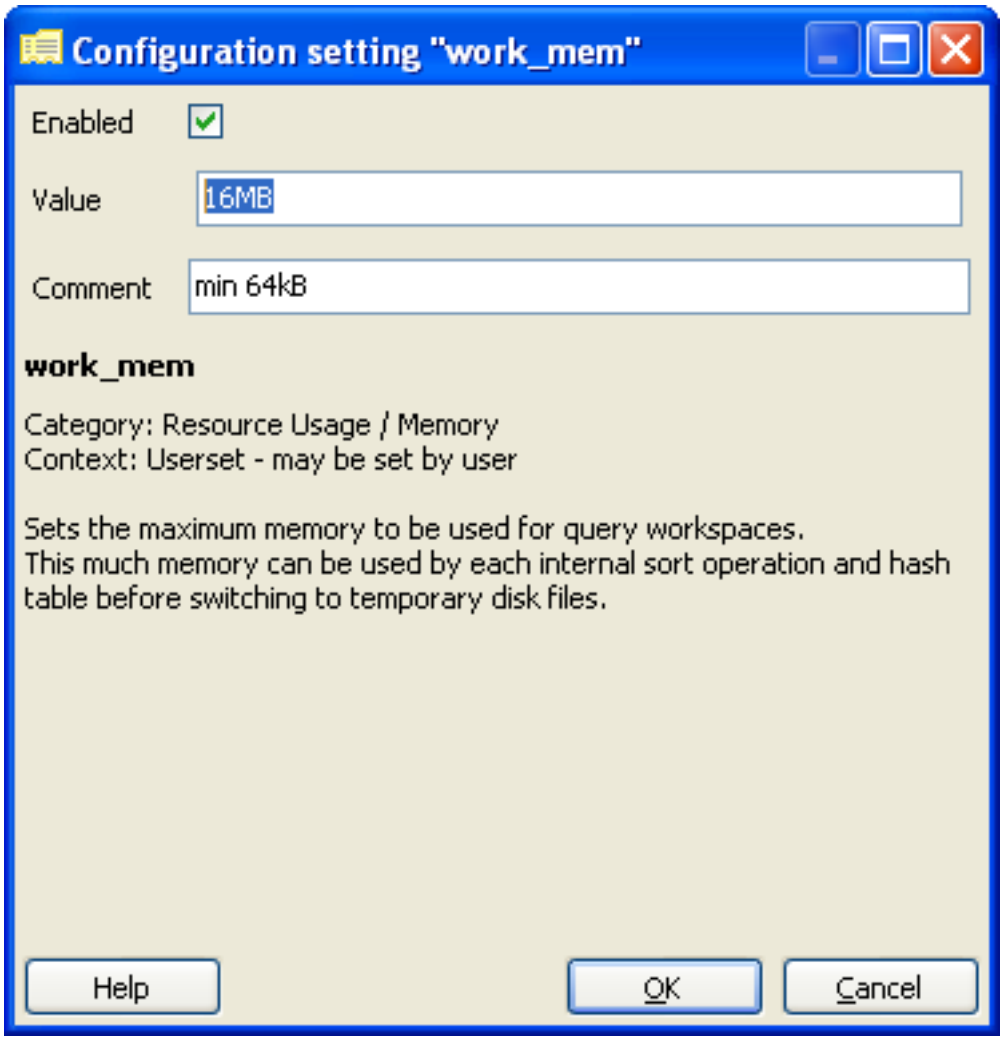
22.2 work_mem

Définit la quantité de mémoire que les opération internes d'ordonnancement et les tables de hachages peuvent consommer avec le serveur sur le disque. Cette valeur définit la mémoire disponible pour chaque opération complexe, les requêtes complexes peuvent avoir plusieurs ordres ou opération de hachage tournant en parallèle, et chaque client connecté peut exécuter une requête.

Vous devez donc considérer combien de connexions et quelle complexité est attendue dans les requêtes avant d'augmenter cette valeur. Le bénéfice acquis par l'augmentation de cette valeur est que la plupart des opération de classification, dont les clause ORDER BY et DISTINCT, les jointures, les agrégation basées sur les hachages et l'exécution de requête imbriquées, pourront être réalisées sans avoir à passer par un stockage sur disque.

Valeur par défaut : 1MB

Valeur recommandée : 16MB



22.3 maintenance_work_mem

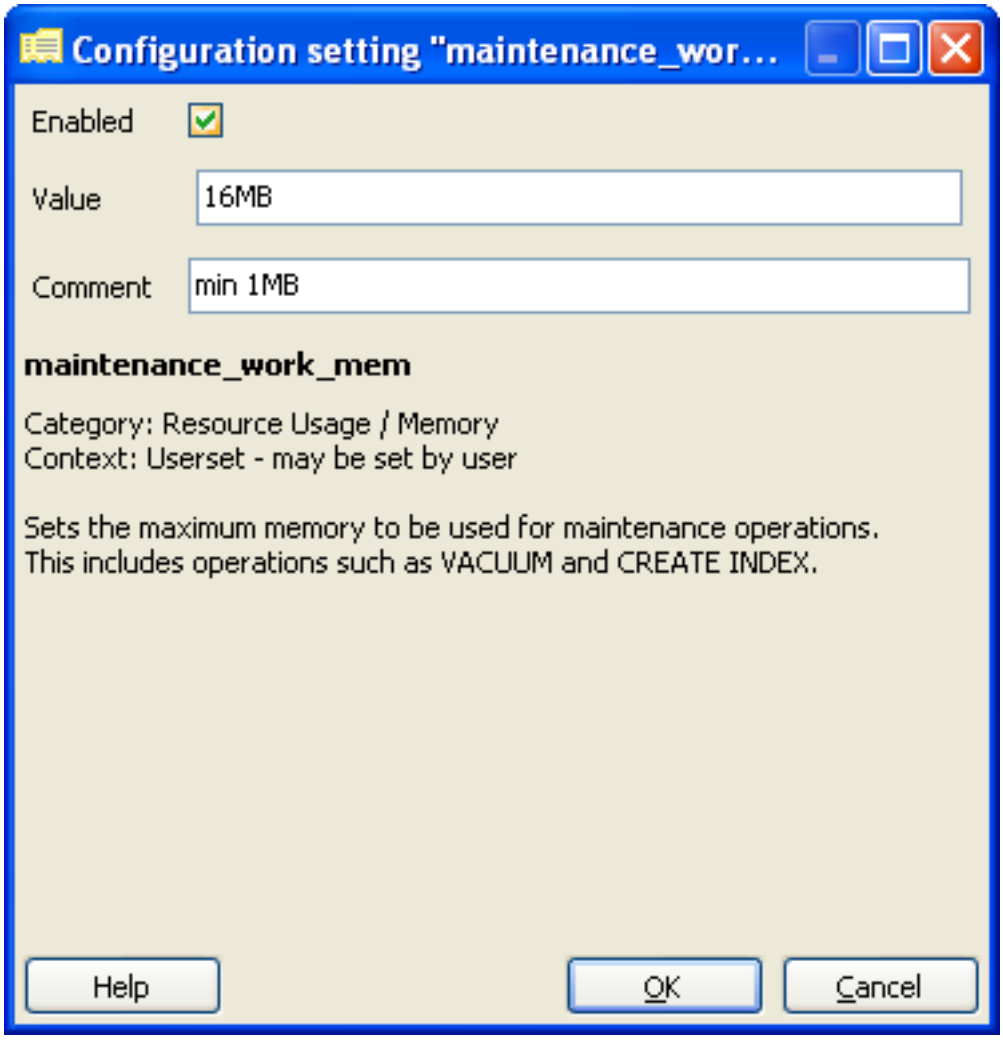
Définit la quantité de mémoire utilisée pour les opération de maintenance, dont le nettoyage (VACUUM), les index et la création de clefs étrangères. Comme ces opération sont couramment utilisées, la valeur par défaut devrait être acceptable. Ce paramètre peut être augmenté dynamiquement à l'exécution depuis une

connexion au serveur avant l'exécution d'un grand nombre d'appels à **CREATE INDEX** ou **VACUUM** comme le montre la commande suivante.

```
SET maintenance_work_mem TO '128MB';
VACUUM ANALYZE;
SET maintenance_work_mem TO '16MB';
```

Valeur par défaut : 16MB

Valeur recommandée : 128MB



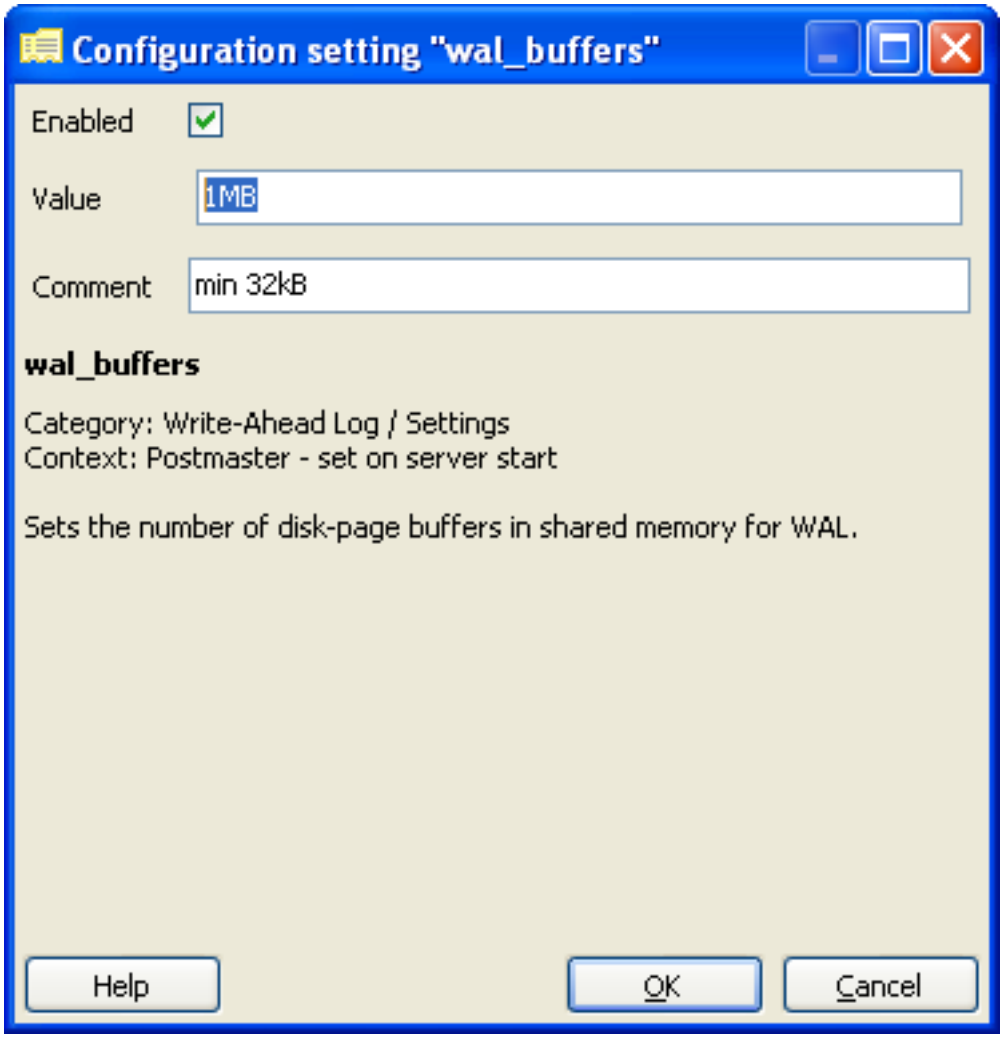
22.4 wal_buffers

Définit la quantité de mémoire utilisée pour l'écriture des données dans le journal respectant la règle du defer (WAL). Elle indique que les informations pour annuler les effets d'une opération sur un objet doivent être écrites dans le journal en mémoire stable avant que l'objet modifié ne migre sur le disque. Cette règle permet d'assurer l'intégrité des données lors d'une reprise après défaillance. En effet, il suffira de lire le journal pour retrouver l'état de la base lors de son arrêt brutal.

La taille de ce tampon nécessite simplement d'être suffisament grand pour stocker les données WAL pour une seule transaction. Alors que la valeur par défaut est généralement suffisante, les données spatiales tendent à être plus larges. Il est donc recommandé d'augmenter la taille spécifiée dans ce paramètre.

Valeur par défaut : 64kB

Valeur recommandée : 1MB



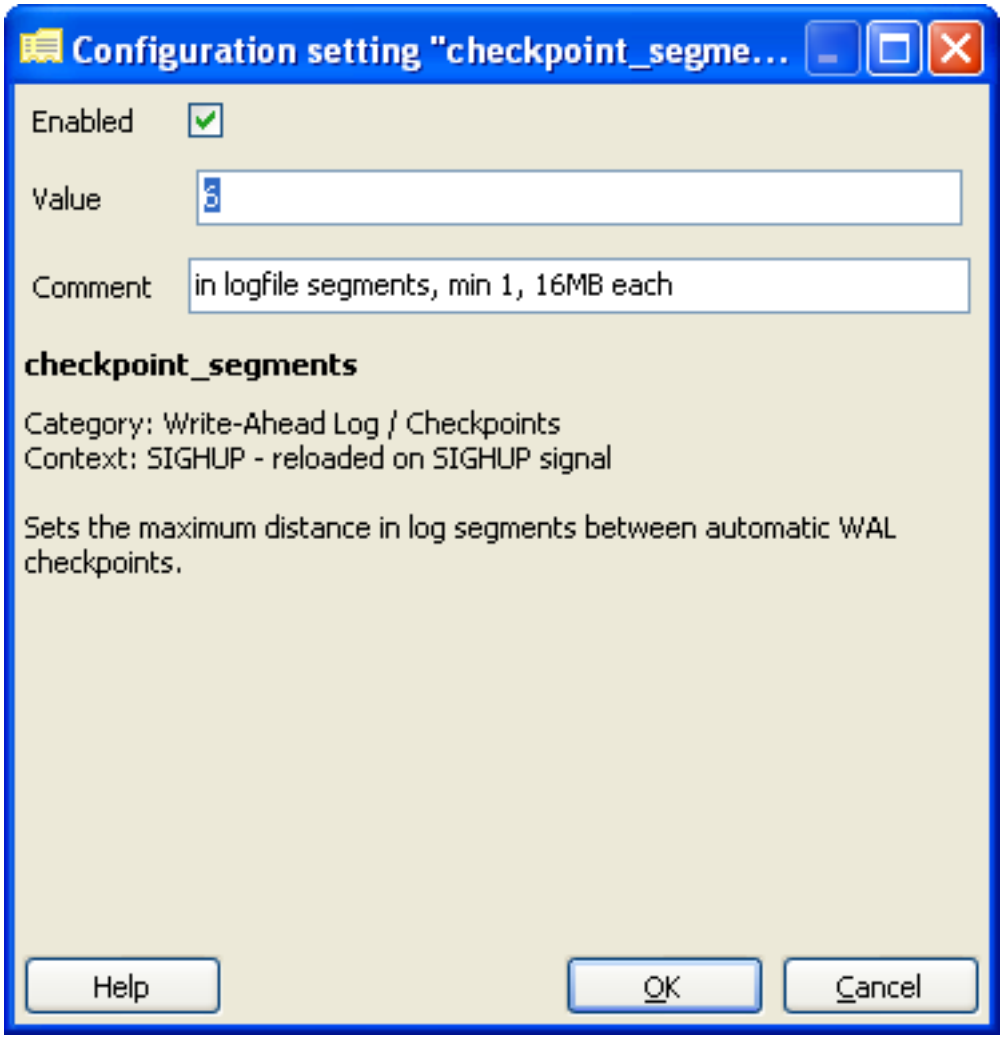
22.5 checkpoint_segments

Cette valeur définit le nombre maximum de segments des journaux (typiquement 16MB) qui doit être remplit entre chaque point de reprise WAL. Un point de reprise WAL est une partie d'une séquence de transactions pour lequel on garantit que les fichiers de données ont été mis à jour avec toutes les requêtes précédant ce point. À ce moment-là toutes les pages sont punaisées sur le disque et les points de reprise sont écrits dans le fichier de journal. Cela permet au processus de reprise après défaillance de trouver les derniers points de reprise et applique toute les lignes suivantes pour récupérer l'état des données avant la défaillance.

Étant donné que les points de reprise nécessitent un punaisage de toutes le pages ayant été modifiées sur le disque, cela va créer une charge d'entrées/sorties significative. Le même argument que précédemment s'applique ici, les données spatiales sont assez grandes pour contrebalancer l'optimisation de données non spatiales. Augmenter cette valeur limitera le nombre de points de reprise, mais impliquera un redémarrage plus lent en cas de défaillance.

Valeur par défaut : 3

Valeur recommandée : 6

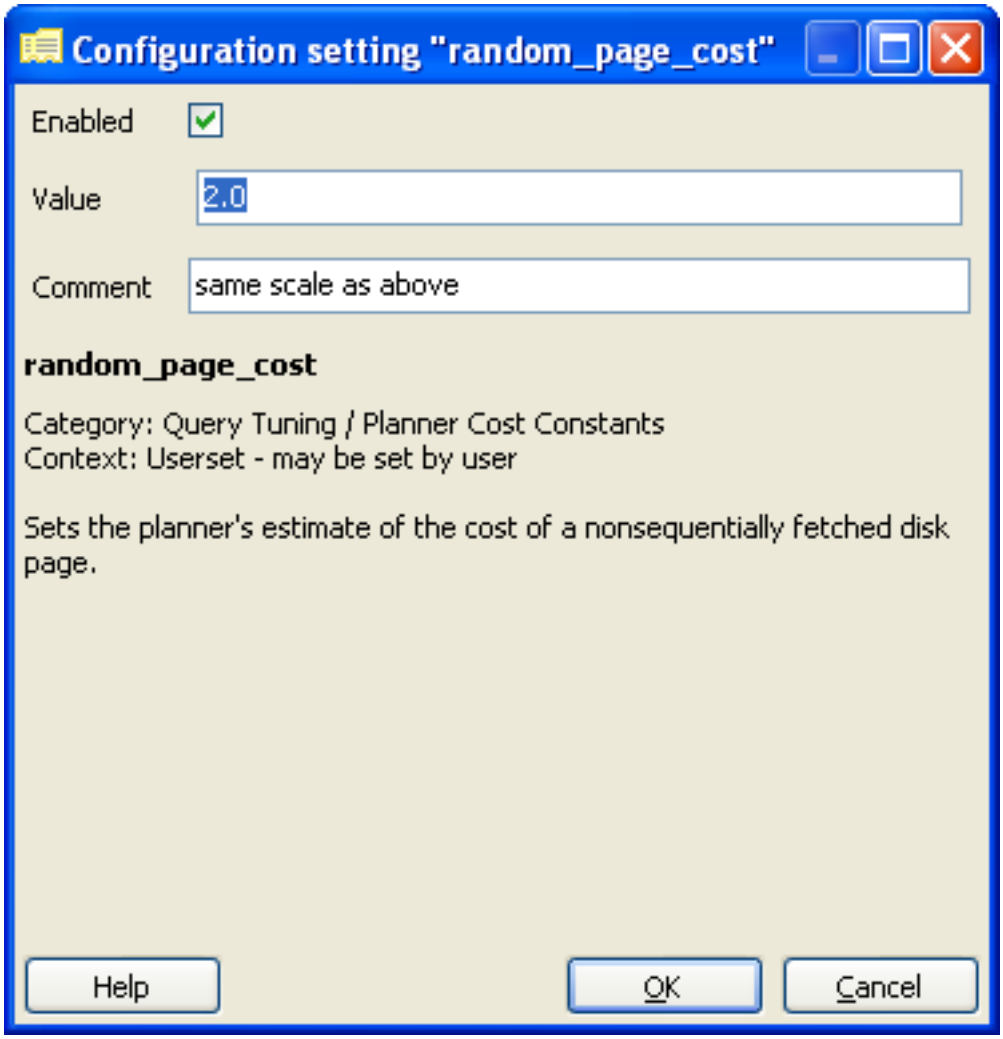


22.6 random_page_cost

Cette valeur sans unité représente le coût d'accès aléatoire à une page du disque. Cette valeur est relative aux autres paramètres de coût notamment l'accès séquentiel aux pages, et le coût des opérations processeur. Bien qu'il n'y ait pas de valeur magique ici, la valeur par défaut est généralement trop faible. Cette valeur peut être affectée dynamiquement par session en utilisant la commande `SET random_page_cost TO 2.0`.

Valeur par défaut : 4.0

Valeur recommandée : 2.0

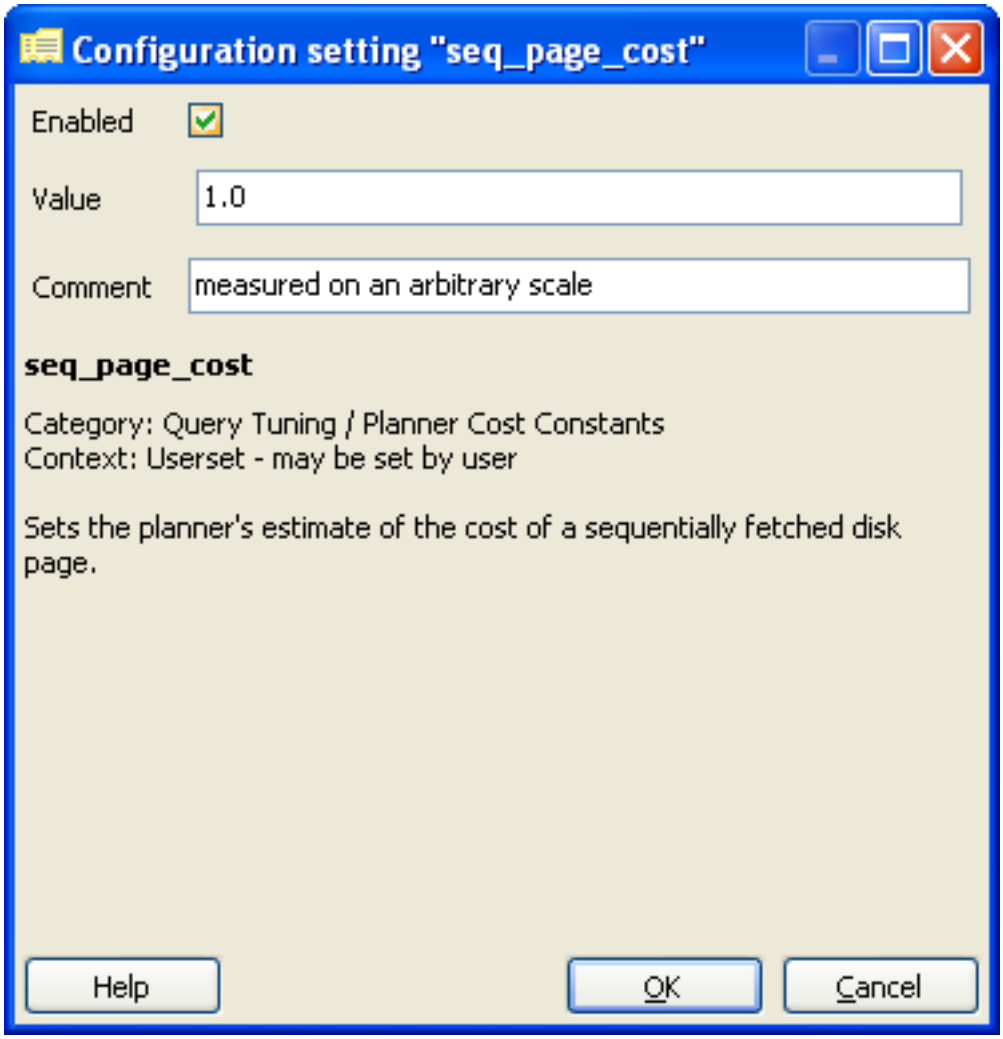


22.7 seq_page_cost

C'est une paramètre qui contrôle le coût des accès séquentiels aux pages. Il n'est généralement pas nécessaire de modifier cette valeur mais la différence entre cette valeur et la valeur `random_page_cost` affecte drastiquement le choix fait par le planificateur de requêtes. Cette valeur peut aussi être affectée depuis une session.

Valeur par défaut : 1.0

Valeur recommandée : 1.0



22.8 Recharger la configuration

Après avoir réalisé les changements mentionnés dans cette partie sauvez-les puis rechargez la configuration.

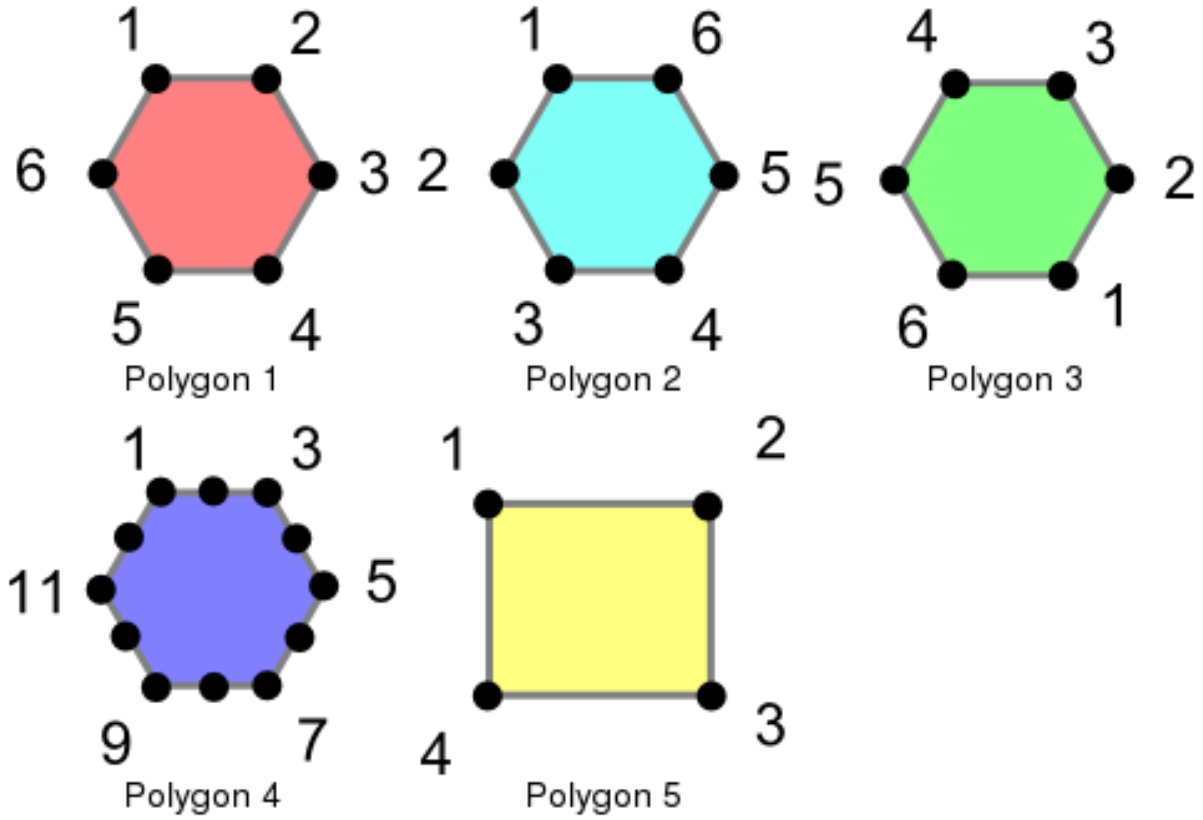
- Ceci se fait en cliquant avec le bouton droit sur le nom du serveur (`PostgreSQL 8.4 on localhost:54321`) depuis pgAdmin, selectionnez *Disconnect*.
- Cliquez sur le bouton *Shutdown* depuis le Dashboard OpenGeo, puis cliquez sur *Start*.
- Pour finir reconnectez-vous au serveur depuis pgAdmin (cliquez avec le bouton droit sur le serveur puis sélectionnez *Connect*).

CHAPITRE 23

Partie 22 : Égalité

23.1 Égalité

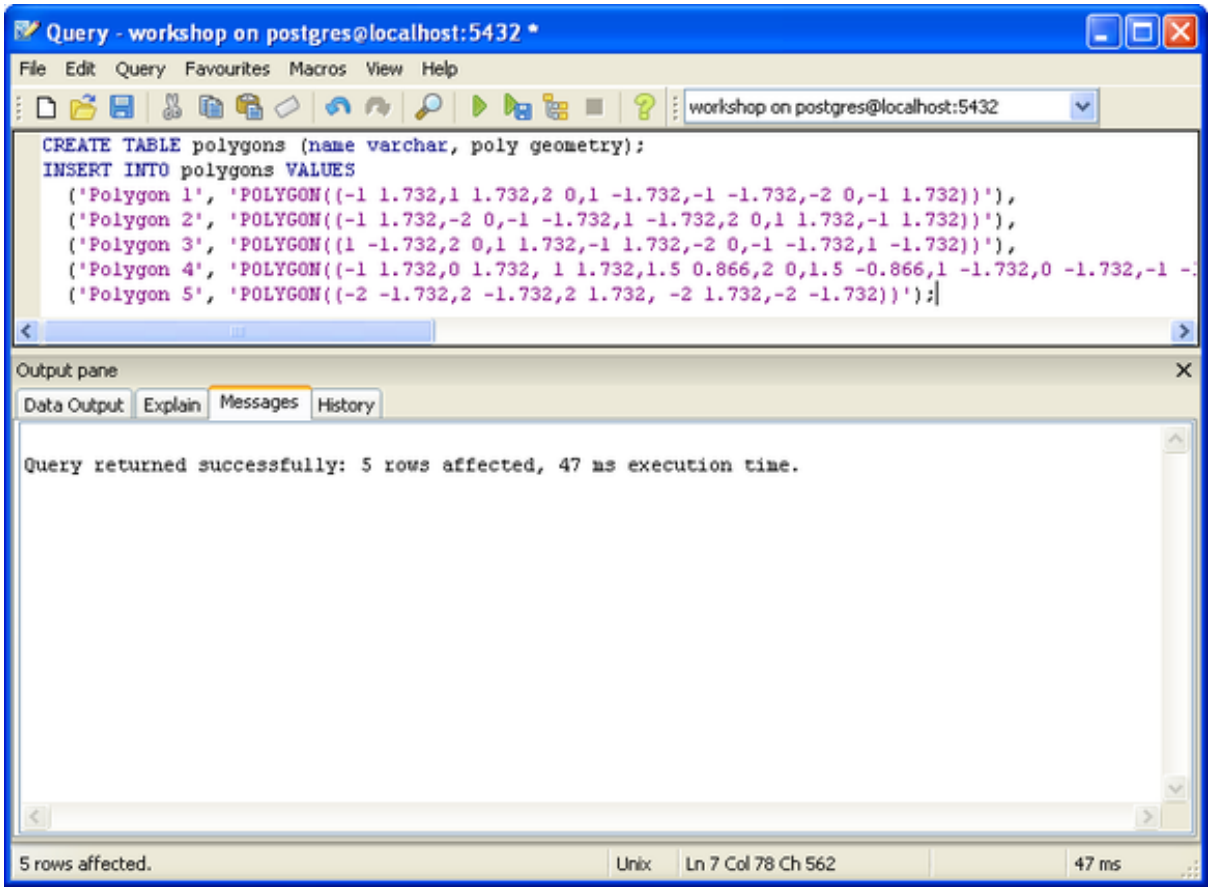
Être en mesure de déterminer si deux geométries sont égales peut être compliqué. PostGIS met à votre disposition différentes fonctions permettant de juger de l'égalité à différents niveaux, bien que pour des raison de simplicité nous nous contenterons ici de la définition fournie plus bas. Pour illustrer ces fonctions, nous utiliserons les polygones suivants.



Ces polygones sont charger à l'aide des commandes suivantes.

```
CREATE TABLE polygons (name varchar, poly geometry);

INSERT INTO polygons VALUES
```

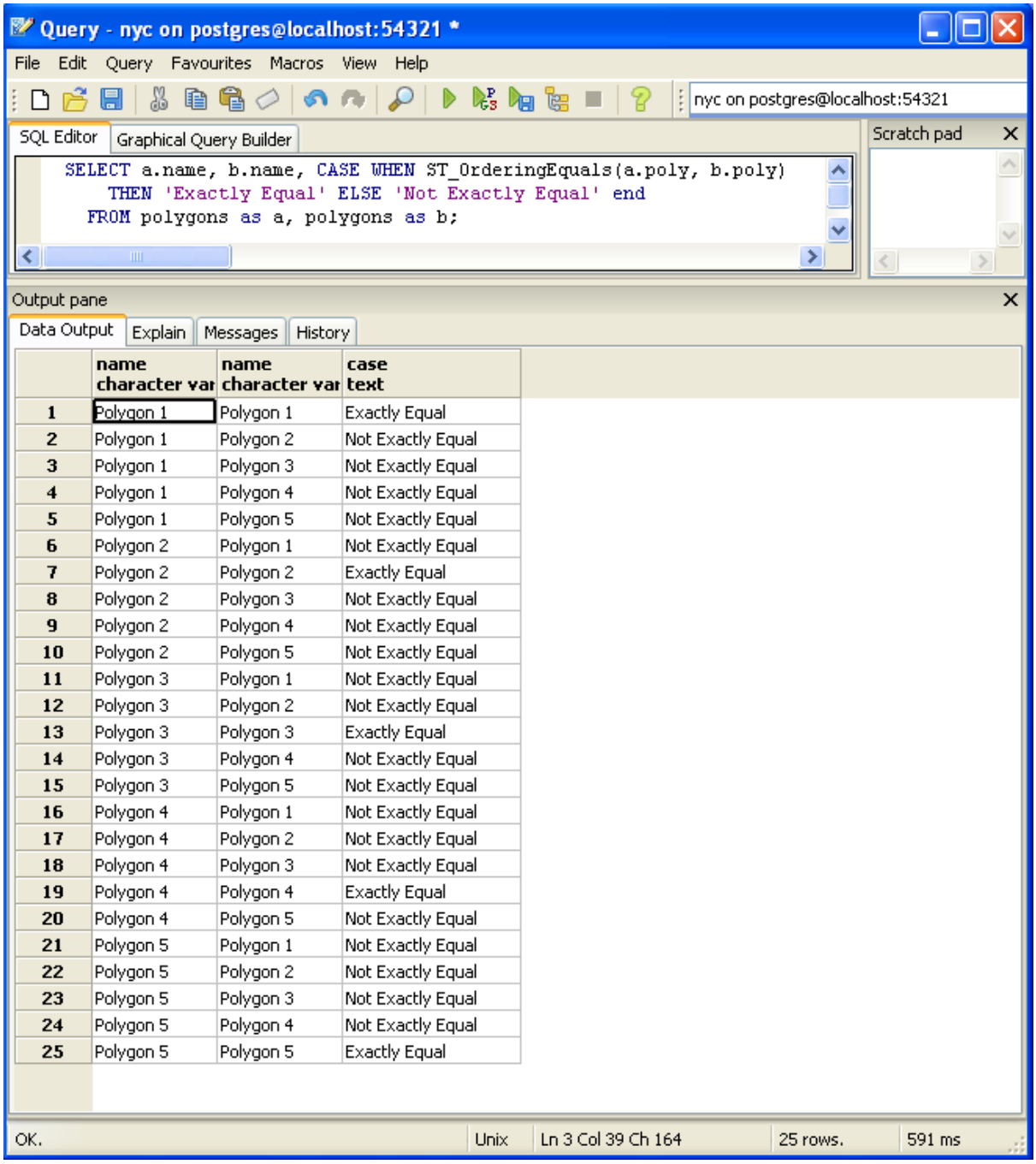
```
('Polygon 1', 'POLYGON((-1 1.732,1 1.732,2 0,1 -1.732,
    -1 -1.732,-2 0,-1 1.732))'),
('Polygon 2', 'POLYGON((-1 1.732,-2 0,-1 -1.732,1 -1.732,
    2 0,1 1.732,-1 1.732))'),
('Polygon 3', 'POLYGON((1 -1.732,2 0,1 1.732,-1 1.732,
    -2 0,-1 -1.732,1 -1.732))'),
('Polygon 4', 'POLYGON((-1 1.732,0 1.732, 1 1.732,1.5 0.866,
    2 0,1.5 -0.866,1 -1.732,0 -1.732,-1 -1.732,-1.5 -0.866,
    -2 0,-1.5 0.866,-1 1.732))'),
('Polygon 5', 'POLYGON((-2 -1.732,2 -1.732,2 1.732,
    -2 1.732,-2 -1.732))');

SELECT Populate_Geometry_Columns();
```



23.1.1 Exactement égaux

L'égalité exacte est déterminée en comparant deux géométries, sommets par sommets, dans l'ordre, pour s'assurer que chacun est à une position identique. Les exemples suivant montrent comment cette méthode peut être limitée dans son efficacité.

```
SELECT a.name, b.name, CASE WHEN ST_OrderingEquals(a.poly, b.poly)
    THEN 'Exactly Equal' ELSE 'Not Exactly Equal' end
  FROM polygons as a, polygons as b;
```



	name character var	name character var	case text
1	Polygon 1	Polygon 1	Exactly Equal
2	Polygon 1	Polygon 2	Not Exactly Equal
3	Polygon 1	Polygon 3	Not Exactly Equal
4	Polygon 1	Polygon 4	Not Exactly Equal
5	Polygon 1	Polygon 5	Not Exactly Equal
6	Polygon 2	Polygon 1	Not Exactly Equal
7	Polygon 2	Polygon 2	Exactly Equal
8	Polygon 2	Polygon 3	Not Exactly Equal
9	Polygon 2	Polygon 4	Not Exactly Equal
10	Polygon 2	Polygon 5	Not Exactly Equal
11	Polygon 3	Polygon 1	Not Exactly Equal
12	Polygon 3	Polygon 2	Not Exactly Equal
13	Polygon 3	Polygon 3	Exactly Equal
14	Polygon 3	Polygon 4	Not Exactly Equal
15	Polygon 3	Polygon 5	Not Exactly Equal
16	Polygon 4	Polygon 1	Not Exactly Equal
17	Polygon 4	Polygon 2	Not Exactly Equal
18	Polygon 4	Polygon 3	Not Exactly Equal
19	Polygon 4	Polygon 4	Exactly Equal
20	Polygon 4	Polygon 5	Not Exactly Equal
21	Polygon 5	Polygon 1	Not Exactly Equal
22	Polygon 5	Polygon 2	Not Exactly Equal
23	Polygon 5	Polygon 3	Not Exactly Equal
24	Polygon 5	Polygon 4	Not Exactly Equal
25	Polygon 5	Polygon 5	Exactly Equal

Dans cette exemple, les polygones sont seulement égaux à eux-même, mais jamais avec un des autres polygones (dans notre exemple les polygones de 1 à 3). Dans le cas des polygones 1, 2 et 3, les sommets sont à des positions identiques mais sont définis dans un ordre différent. Le polygone 4 a des sommets en double causant la non-égalité avec le polygone 1.

23.1.2 Spatialement égaux

Comme nous l'avons précédemment montré, l'égalité exacte ne prend pas en compte la nature spatiale des géométries. Il y a une fonction, nommée **ST_Equals**, permettant de tester l'égalité spatiale ou l'équivalence des géométries.

```
SELECT a.name, b.name, CASE WHEN ST_Equals(a.poly, b.poly)
    THEN 'Spatially Equal' ELSE 'Not Equal' end
  FROM polygons as a, polygons as b;
```

Query - nyc on postgres@localhost:54321 *

File Edit Query Favourites Macros View Help

nyc on postgres@localhost:54321

SQL Editor | Graphical Query Builder | Scratch pad

```
SELECT a.name, b.name, CASE WHEN ST_Equals(a.poly, b.poly)
    THEN 'Spatially Equal' ELSE 'Not Equal' end
  FROM polygons as a, polygons as b;
```

Output pane

Data Output | Explain | Messages | History

	name character var	name character var	case text
1	Polygon 1	Polygon 1	Spatially Equal
2	Polygon 1	Polygon 2	Spatially Equal
3	Polygon 1	Polygon 3	Spatially Equal
4	Polygon 1	Polygon 4	Spatially Equal
5	Polygon 1	Polygon 5	Not Equal
6	Polygon 2	Polygon 1	Spatially Equal
7	Polygon 2	Polygon 2	Spatially Equal
8	Polygon 2	Polygon 3	Spatially Equal
9	Polygon 2	Polygon 4	Spatially Equal
10	Polygon 2	Polygon 5	Not Equal
11	Polygon 3	Polygon 1	Spatially Equal
12	Polygon 3	Polygon 2	Spatially Equal
13	Polygon 3	Polygon 3	Spatially Equal
14	Polygon 3	Polygon 4	Spatially Equal
15	Polygon 3	Polygon 5	Not Equal
16	Polygon 4	Polygon 1	Spatially Equal
17	Polygon 4	Polygon 2	Spatially Equal
18	Polygon 4	Polygon 3	Spatially Equal
19	Polygon 4	Polygon 4	Spatially Equal
20	Polygon 4	Polygon 5	Not Equal
21	Polygon 5	Polygon 1	Not Equal
22	Polygon 5	Polygon 2	Not Equal
23	Polygon 5	Polygon 3	Not Equal
24	Polygon 5	Polygon 4	Not Equal
25	Polygon 5	Polygon 5	Spatially Equal

OK. | Unix | Ln 4 Col 1 Ch 152 | 25 rows. | 230 ms

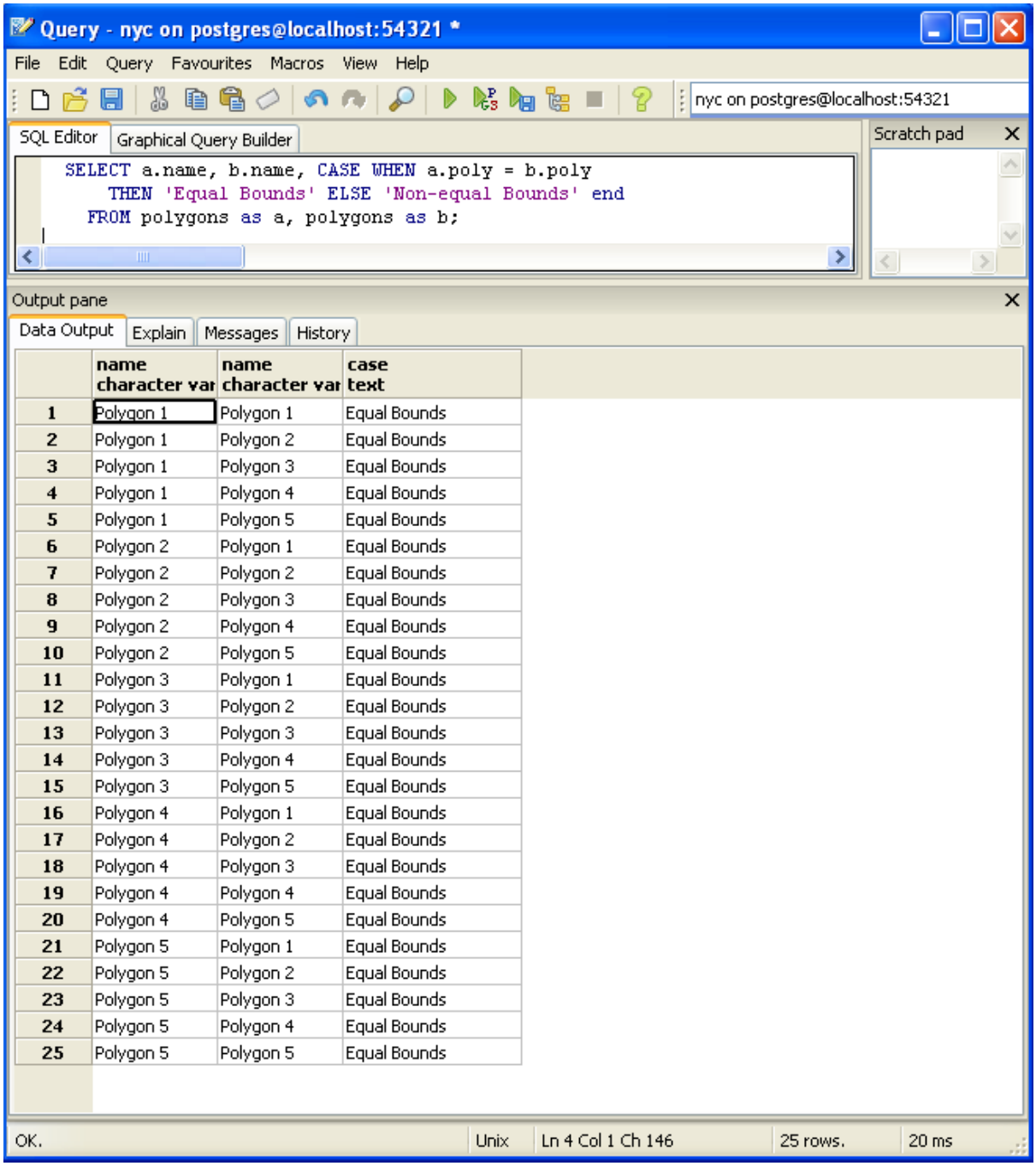
Ces résultats sont plus proches de notre compréhension intuitive de l'égalité. Les polygones de 1 à 4 sont considérés comme égaux, puisque qu'ils recouvrent la même zone. Notez que ni la direction des polygones n'est considérée, ni le point de départ pour la définition du polygone, ni le nombre de points. Ce qui importe c'est que la zone géographique représentée soit la même.

23.1.3 Égalité des étendues

L'égalité exacte nécessite, dans le pire des cas, de comparer chacun des sommets d'une géométrie pour déterminer l'égalité. Ceci peut être très lent, et s'avérer innaproprié pour comparer un grand nombre de

géométries. Pour permettre de rendre plus rapide ces comparaison, l'opération d'égalité des étendue est fournit : **=**. Cet opérateur utilise uniquement les étendues (cadre limite rectangulaire), assurant que les géométries occupent le même espace dans un repère cartésien en deux dimensions, mais ne représente pas nécessairement le même espace.

```
SELECT a.name, b.name, CASE WHEN a.poly = b.poly
    THEN 'Equal Bounds' ELSE 'Non-equal Bounds' end
  FROM polygons as a, polygons as b;
```



	name character var	name character var	case text
1	Polygon 1	Polygon 1	Equal Bounds
2	Polygon 1	Polygon 2	Equal Bounds
3	Polygon 1	Polygon 3	Equal Bounds
4	Polygon 1	Polygon 4	Equal Bounds
5	Polygon 1	Polygon 5	Equal Bounds
6	Polygon 2	Polygon 1	Equal Bounds
7	Polygon 2	Polygon 2	Equal Bounds
8	Polygon 2	Polygon 3	Equal Bounds
9	Polygon 2	Polygon 4	Equal Bounds
10	Polygon 2	Polygon 5	Equal Bounds
11	Polygon 3	Polygon 1	Equal Bounds
12	Polygon 3	Polygon 2	Equal Bounds
13	Polygon 3	Polygon 3	Equal Bounds
14	Polygon 3	Polygon 4	Equal Bounds
15	Polygon 3	Polygon 5	Equal Bounds
16	Polygon 4	Polygon 1	Equal Bounds
17	Polygon 4	Polygon 2	Equal Bounds
18	Polygon 4	Polygon 3	Equal Bounds
19	Polygon 4	Polygon 4	Equal Bounds
20	Polygon 4	Polygon 5	Equal Bounds
21	Polygon 5	Polygon 1	Equal Bounds
22	Polygon 5	Polygon 2	Equal Bounds
23	Polygon 5	Polygon 3	Equal Bounds
24	Polygon 5	Polygon 4	Equal Bounds
25	Polygon 5	Polygon 5	Equal Bounds

Comme vous pouvez le constater, toutes les géométries égales ont aussi une étendue égale. Malheureusement, le polygone 5 est aussi retourné comme étant égal avec ce test, puisqu'il partage la même étendue que les autres géométries. Mais alors, pourquoi est-ce utile ? Bien que cela soit traité en détail plus tard, la réponse courte est que cela permet l'utilisation d'indexations spatiales qui peuvent réduire drastiquement les ensembles de géométries à comparer en utilisant des filtres utilisant cette égalité d'étendue.

CHAPITRE 24

Annexes A : Fonctions PostGIS

24.1 Constructeurs

ST_MakePoint(Longitude, Latitude) Retourne un nouveau point. Note : ordre des coordonnées (longitude puis latitude).

ST_GeomFromText(WellKnownText, srid) Retourne une nouvelle géométrie à partir d'un représentation au format WKT et un SRID.

ST_SetSRID(geometry, srid) Met à jour le SRID d'une géométrie. Retourne la même géométrie. Cela ne modifie pas les coordonnées de la géométrie, cela met simplement à jour le SRID. Cette fonction est utile pour reconditionner les géométries sans SRID.

ST_Expand(geometry, Radius) Retourne une nouvelle géométrie qui est une extension de l'étendue de la géométrie passée en argument. Cette fonction est utile pour créer des enveloppes pour des recherches utilisant les indexations.

24.2 Sorties

ST_AsText(geometry) Retourne une géométrie au format WKT.

ST_AsGML(geometry) Retourne la géométrie au format standard OGC *GML*.

ST_AsGeoJSON(geometry) Retourne une géométrie au format "standard" GeoJSON.

24.3 Mesures

ST_Area(geometry) Retourne l'aire d'une géométrie dans l'unité du système de référence spatiale.

ST_Length(geometry) Retourne la longueur de la géométrie dans l'unité du système de référence spatiale.

ST_Perimeter(geometry) Retourne le périmètre de la géométrie dans l'unité du système de référence spatiale.

ST_NumPoints(linestring) Retourne le nombre de sommets dans une ligne.

ST_NumRings(polygon) Retourne le nombre de contours dans un polygone.

ST_NumGeometries(geometry) Retourne le nombre de géométries dans une collection de géométries.

24.4 Relations

ST_Distance(geometry, geometry) Retourne la distance entre deux géométries dans l'unité du système de référence spatiale.

ST_DWithin(geometry, geometry, radius) Retourne TRUE si les géométries sont distantes d'un rayon de l'autre, sinon FALSE.

ST_Intersects(geometry, geometry) Retourne TRUE si les géométries sont disjointes, sinon FALSE.

ST_Contains(geometry, geometry) Retourne TRUE si la première géométrie est totalement contenue dans la seconde, sinon FALSE.

ST_Crosses(geometry, geometry) Retourne TRUE si une ligne ou les contours d'un polygone croisent une ligne ou un contour de polygone, sinon FALSE.

CHAPITRE 25

Annexes B : Glossaire

CRS Un "système de référence spatiale". La combinaison d'un système de coordonnéee géographiques et un système de projection.

GDAL Geospatial Data Abstraction Library, prononcé "GéDAL", une bibliothèque open source permettant d'accéder aux données rasters supportant un grand nombre de formats, utilisé largement à la fois dans les applications open source et propriétaires.

GeoJSON "Javascript Object Notation", un format texte qui est très rapide et qui permet de représenter des objets JavaScript. En spatial, la spécification étendue GeoJSON est couramment utilisée.

SIG Système d'Information Géographique capture, stocke, analyse, gère, et présente les données qui sont reliées à la zone géographique.

GML Geography Markup Language. Le GML est un format standard XML *OGC* pour représenter les données géographiques.

JSON "Javascript Object Notation", un format texte qui est très rapide et permet de stocker des objets JavaScript. Au niveau spatial, la spécification étendue GeoJSON est couramment utilisé.

JSTL "JavaServer Page Template Library", est une bibliothèque pour *JSP* qui encapsule plusieurs fonctionnalités de bases gérées en JSP (requête de bases de données, itération, conditionnel) dans un syntaxe tierce.

JSP "JavaServer Pages" est un système de script pour les serveur d'applications Java qui permet de mixer du code XML et du code Java.

KML "Keyhole Markup Language", le format XML utilisé par Google Earth. Google Earth. Il fût à l'origine développé par la société "Keyhole", ce qui explique sa présence (maintenant obscure) dans le nom du format.

OGC Open Geospatial Consortium http ://opengeospatial.org/ (OGC) est une organisation qui développe des spécifications pour les services spatiaux.

OSGeo Open Source Geospatial Foundation http ://osgeo.org (OSGeo) est une association à but non lucratif dédiée à la promotion et au support des logiciels cartographiques open source.

SFSQL La spécification Simple Features for SQL (SFSQL) de l'*OGC* définit les types et les fonctions qui doivent être disponibles dans une base de données spatiale.

SLD Les spécifications Styled Layer Descriptor (SLD) de l'*OGC* définissent un format permettant de décrire la manière d'afficher des données vectorielles.

SRID "Spatial reference ID" est un identifiant unique assigné à un système de coordonnées géographiques particulier. La table PostGIS **spatial_ref_sys** contient une large collection de valeurs de SRID connus.

SQL "Structured query language" est un standard permettant de requêter les bases de données relationnelles. Référence http ://en.wikipedia.org/wiki/SQL.

SQL/MM SQL Multimedia ; spécification contenant différentes sections sur les types étendues. Elle inclue une section substantielle sur les types spatiaux.

SVG "Scalable vector graphics" est une famille de spécifications basé sur le format XML pour décrire des objet graphiques en 2 dimensions, aussi bien statiques que dynamiques (par exemple interactif ou animé). Réference : http ://en.wikipedia.org/wiki/Scalable_Vector_Graphics.

WFS La spécification Web Feature Service (WFS) de l'*OGC* définit une interface pour lire et écrire des données géographiques à travers internet.

WMS La spécification Web Map Service (WMS) de l'*OGC* définit une interface pour requêter une carte à travers internet.

WKB "Well-known binary". Fait référence à la représentation binaire des géométries comme décrit dans les spécifications Simple Features for SQL (*SFSQL*).

WKT "Well-known text". Fait référence à la représentation textuelle de géométries, avec des chaînes commençant par "POINT", "LINESTRING", "POLYGON", etc. Il peut aussi faire référence à la représentation textuelle d'un *CRS*, avec une chaîne commençant par "PROJCS", "GEOGCS", etc. Les représentations au format Well-known text sont des standards de l'*OGC*, mais n'ont pas leur propres documents de spécifications. La première description du WKT (pour les géométries et pour les CRS) apparaissent dans les spécifications *SFSQL* 1.0.

CHAPITRE 26

Annexes C : License

Index

C

CRS, 143

G

GDAL, 143
GeoJSON, 143
GML, 143

J

JSON, 143
JSP, 143
JSTL, 143

K

KML, 143

O

OGC, 143
OSGeo, 143

S

SFSQL, 143
SIG, 143
SLD, 143
SQL, 143
SQL/MM, 144
SRID, 143
SVG, 144

W

WFS, 144
WKB, 144
WKT, 144
WMS, 144