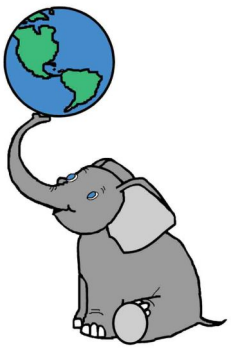
Bonnes pratiques et optimisation



Ministère de l'Écologie, du Développement
durable et de l'Énergie

Licence ETALAB

1er mars 2016
Version 1.0.0

Table des matières

Objectifs

Les objectifs du module sont de :

- Savoir installer correctement le serveur PostgreSQL/PostGIS
- Optimiser le paramétrage du serveur pour l'exploitation des données spatiales
- Connaître et appliquer les bonnes pratiques de sécurité (intrusion, intégrité des données)
- Savoir optimiser la gestion de la base de données spatiale
- Savoir optimiser les requêtes SQL

Introduction

Le temps d'apprentissage de ce module est estimé à 2 heures et 30 minutes.

Il ne comporte pas d'exercices mais fournit des informations importantes sur l'optimisation et la sécurité.

Installation du serveur PostgreSQL/PostGIS

Les informations qui suivent sont extraites du document PostgreSQL/PostGIS : bonnes pratiques et astuces (portail SIG)

http://www.portailsig.org/content/postgresql-postgis-bonnes-pratiques-et-astuces

et du support de formation réalisé par Christophe Masse et Didier Leclerc, CMSIG.

Fondamental

- Le paramétrage du serveur PostgreSQL est enregistré dans le fichier de configuration **postgresql.conf**
- Seul l'administrateur du serveur peut changer les paramètres du serveur et modifier le fichier postgresql.conf
- Le serveur PostgreSQL doit être relancé pour appliquer les modifications de configuration

1 - Choisir un port différent de celui par défaut (5432)

Cette mesure de sécurité basique est surtout conseillée dans le cadre d'une ouverture vers le web.

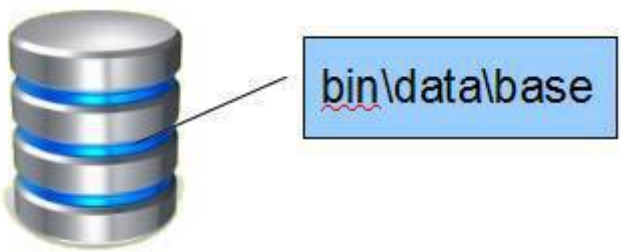
Méthode

Dans le fichier de configuration postgresql.conf, changer la valeur **port=5432**

2 - Créer un tablespace sur une autre partition que la partition système

Le tablespace est un espace de stockage qui va permettre de stocker les données d'une base de données (tables, index, tables systèmes...).

En le définissant sur une autre partition que la partition système, vous évitez d'encombrer celle-ci en l'attribuant aux nouvelles bases de données.



Attention : le modifier à posteriori pour une base ne sera pris en compte que pour les nouveaux éléments (tables, index...) de la base.

Pour changer le tablespace sur ceux-ci, il vous faut modifier le tablespace de chacun des éléments (table, index).

Exemple : Créer un tablespace sur une autre partition que la partition système :

- Sous LINUX : *CREATE TABLESPACE espace_rapide LOCATION '/mnt/sda1/postgresql/data';*
- Sous WINDOWS : *CREATE TABLESPACE espace_rapide LOCATION 'E:/sda1/postgresql/data';*

Remarque :

On peut aussi modifier le paramètre **default_tablespace** dans le fichier postgresql.conf

Ex : **default_tablespace = '/mnt/sda1/postgresql/data'**

3 - Mettre toutes les données dans un ou plusieurs autre(s) schéma(s) que le schéma par défaut "public"

Cette action très importante est à mettre en oeuvre dès la phase de conception de la base de données

Elle permet de :

- Séparer le coeur de PostGIS (tables systèmes PostGIS, fonctions...) des données elles-mêmes
- Organiser les données par la création de schémas par thématique métier , ou selon d'autres organisations (référentiel, données tabulaires/vectorielles etc)

- Simplifier potentiellement la gestion des droits d'utilisation par cette organisation en schémas multiples

Le principal avantage reste néanmoins de faciliter la mise à jour de PostGIS ainsi que la restauration et l'échange plus efficaces de données.

L'import d'une sauvegarde contenant les fonctionnalités PostGIS dans une autre installation peut en effet causer des conflits.

4 - Bien vérifier les "ouvertures" vers l'extérieur

En effet, il est important de limiter l'accès à PostgreSQL au strict nécessaire.

Paramètre	Fichier de configuration	Commentaires
listen_addresses	**postgresql.conf**	Permet de définir quelles IP sont "écoutées" par PostgreSQL (toutes, localhost). Par défaut : '*' (le serveur écoute tout)
listen_addresses	**pg_hba.conf**	Permet de gérer plus finement les accès en configurant les IP ou plages d'IPs autorisées en précisant les bases autorisées et les modes d'accès (sans mot de passe, mot de passe cryptés...).

Le chapitre suivant est consacré à l'optimisation des performances du serveur PostgreSQL pour l'exploitation des données spatiales.

II

Paramétrage du serveur PostgreSQL pour l'exploitation des données spatiales

Les informations qui suivent sont extraites du document "Paramétrer PostgreSQL pour le spatial" réalisé par Paul Ramsay

http://www.postgis.fr/chrome/site/docs/workshop-foss4g/doc/tuning.html

La documentation PostgreSQL est aussi à consulter pour plus de détails :

http://docs.postgresql.fr/9.4/runtime-config.html

Fondamental

- Le paramétrage du serveur PostgreSQL est enregistré dans 2 fichiers de configuration dont le fichier **postgresql.conf**
- Seul l'administrateur du serveur peut changer les paramètres du serveur et modifier le fichier postgresql.conf
- Le serveur PostgreSQL doit être relancé pour appliquer les modifications de configuration

Mode opératoire

PostgreSQL est une base de données capable de fonctionner dans des environnements ayant des ressources très limitées et partageant ces ressources avec un grand nombre d'autres applications.

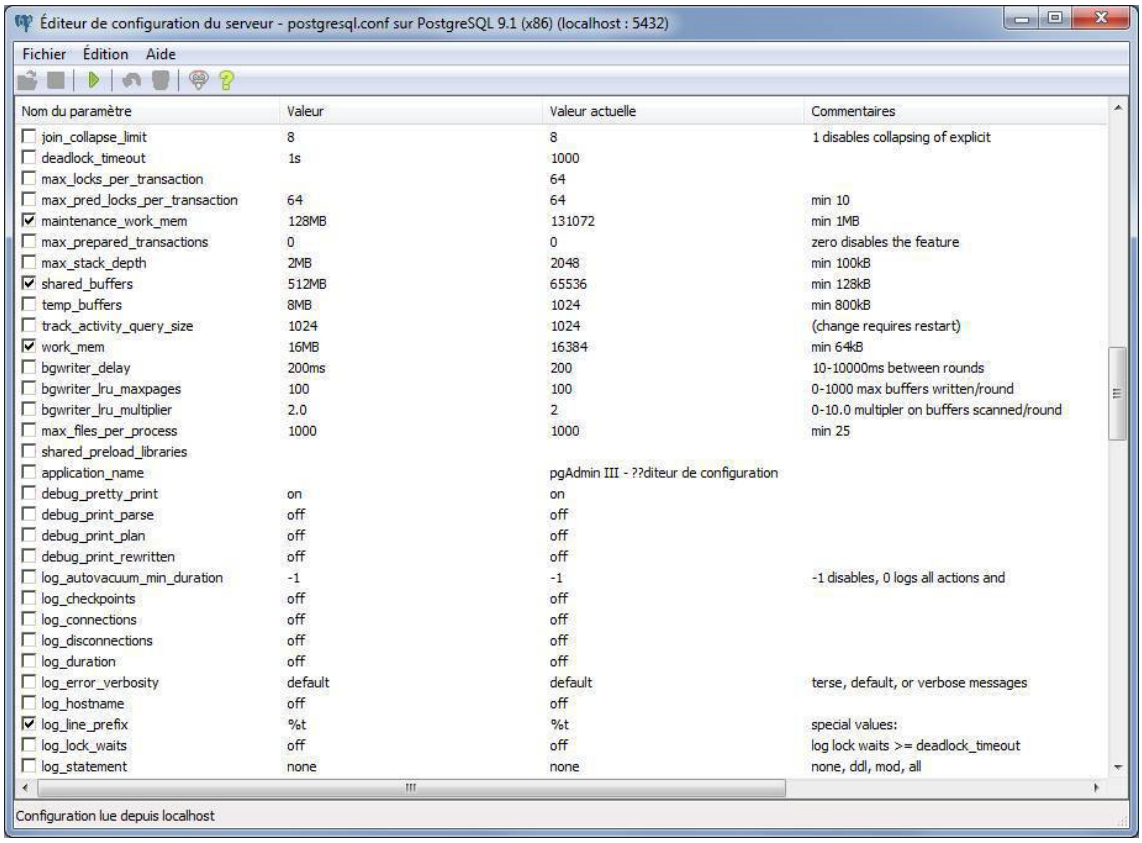
Afin d'assurer que PostgreSQL tournera convenablement dans ces environnements, la configuration par défaut est très peu consommatrice de ressources mais inadaptée pour des bases de données spatiales.

Nous allons voir comment configurer le serveur PostgreSQL pour optimiser son fonctionnement pour l'exploitation des données spatiales en modifiant quelques paramètres importants dans le fichier postgresql.conf.

Le fichier postgresql.conf peut être édité et modifié par l'administrateur du serveur dans un éditeur de texte mais la façon la plus simple de configurer les paramètres est d'éditer le fichier

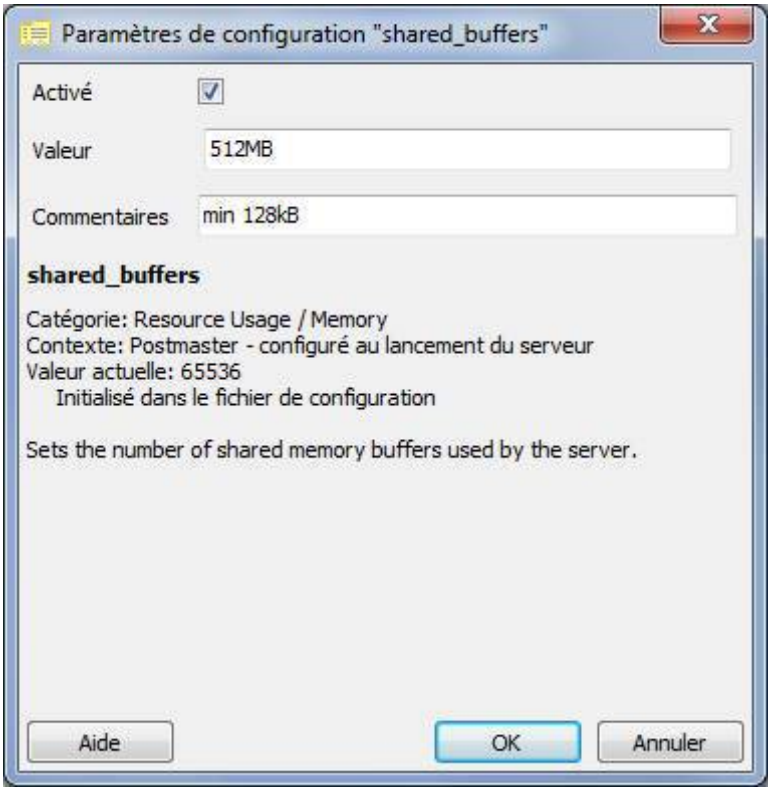
postgresql.conf au moyen de la console PgAdminIII qui comporte un éditeur de configuration du serveur :

Menu Outils → Configuration du serveur → postgresql.conf



Exemple d'édition du fichier postgresql.conf dans PgAdmin III

Un double clic sur une ligne du fichier permet d'éditer le paramètre et d'en modifier la valeur :



- La nouvelle valeur du paramètre est à inscrire dans la fenêtre Valeur (nombre et unités sans espace de séparation ex : 512MB)
- Cocher la case 'Activé' pour forcer la prise en compte de la nouvelle valeur
- Cliquer sur OK pour terminer

Quand tous les paramètres concernés ont été modifiés :

- Sauvegarder les modifications en enregistrant le fichier : Fichier -> Enregistrer
- Relancer le serveur : Fichier -> Rechargement du serveur

Paramètres à modifier

Important :

Les valeurs recommandées ci-dessous pour chaque paramètre de configuration constituent un bon point de départ pour améliorer sensiblement le fonctionnement du serveur pour l'exploitation des données spatiales, mais elles doivent être testées et ajustées pour chaque configuration matérielle du serveur informatique, notamment par rapport à la mémoire vive disponible.

Consommation des ressources : mémoire

shared_ buffers (integer)

Initialise la quantité de mémoire que le serveur de bases de données utilise comme mémoire partagée.

La valeur par défaut, en général 128kB, est insuffisante pour une base de données spatiales en production.

Si vous disposez d'un serveur dédié à la base de données, avec 1 Go de mémoire ou plus, une valeur de départ raisonnable pour ce paramètre est de 25% de la mémoire du système.

PostgreSQL profite aussi du cache du système d'exploitation, il est donc peu probable qu'une allocation de plus de 40% de la mémoire fonctionnera mieux qu'une valeur plus restreinte.

Valeur recommandée : 512MB (en fait 1/4 de la mémoire totale de la machine)

work_ mem (integer)

Indique la quantité de mémoire que les opérations de tri interne et les tables de hachage (relations données/empreintes ou signatures des données) peuvent utiliser avant de basculer sur des fichiers disque temporaires.

La valeur par défaut est de 1MB.

Pour une requête complexe, il peut y avoir plusieurs opérations de tri ou de hachage exécutées en parallèle ; chacune peut utiliser de la mémoire à hauteur de cette valeur avant de commencer à placer les données dans des fichiers temporaires sur le disque.

De plus, de nombreuses sessions peuvent exécuter de telles opérations simultanément. La mémoire totale utilisée peut, de ce fait, atteindre plusieurs fois la valeur de work_mem.

Il faut 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 opérations de classification, dont les clauses ORDER BY et DISTINCT, les jointures, les agrégations basées sur les hachages et l'exécution de requête imbriquées, pourront être réalisées sans avoir à passer par un stockage sur disque.

Valeur recommandée : 16MB (on peut mettre plus, par exemple 128MB pour une mémoire totale de 24 Go RAM)

maintenance_work_mem (integer)

Indique la quantité maximale de mémoire que peuvent utiliser les opérations de maintenance telles que VACUUM, CREATE INDEX et ALTER TABLE ADD FOREIGN KEY.

La valeur par défaut est de 16MB.

Puisque seule une de ces opérations peut être exécutée à la fois dans une session et que, dans le cadre d'un fonctionnement normal, peu d'opérations de ce genre sont exécutées concurrentiellement sur une même installation, il est possible d'initialiser cette variable à une valeur bien plus importante que work_mem.

Une grande valeur peut améliorer les performances des opérations VACUUM et de la restauration des sauvegardes.

Valeur recommandée : 128MB (256MB à 1GB pour une mémoire totale de 24 Go RAM)

Consommation des ressources : Write Ahead Log

Write-Ahead Logging (WAL) est une méthode conventionnelle pour s'assurer de l'intégrité des données.

Le concept central du WAL est d'effectuer les changements des fichiers de données (donc les tables et les index) uniquement après que ces changements ont été écrits de façon sûre dans un journal, appelé journal des transactions.

Il n'est pas nécessaire d'écrire les pages de données vers le disque à chaque validation de transaction car, dans l'éventualité d'une défaillance, on pourra récupérer la base de données en utilisant le journal des transactions.

wal_buffers (integer)

Définit la quantité de mémoire partagée utilisée pour les données des journaux de transactions qui n'ont pas encore été écrites sur disque.

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 suffisamment 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 volumineuses.

Il est donc recommandé d'augmenter la quantité de mémoire spécifiée dans ce paramètre.

Valeur recommandée : 1MB

checkpoint_ segments (integer)

Nombre maximum de journaux de transaction entre deux points de vérification automatique des WAL (chaque segment fait normalement 16 Mo).

Cette valeur définit le nombre maximum de segments des journaux (typiquement 16MB) qui doit être rempli 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 repérées 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 de récupérer l'état des données avant la défaillance.

Étant donné que les points de reprise nécessitent un repérage 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 pour les données spatiales.

Augmenter cette valeur limitera le nombre de points de reprise, mais impliquera un redémarrage plus lent en cas de défaillance.

La valeur par défaut est de 3 segments.

Dans les phases **d'import de référentiel volumineux** ou les opérations de remontée de dump conséquent, ce qui constitue la grosse part du travail a l'initialisation de la base le paramètre peut être porté à beaucoup plus (exemple 128*16MB soir 2 GB). Cela signifie que postgres n'exécute pas l'écriture physique tant qu'il n'a pas atteint 2GB de donnés à écrire. Ceci afin de ne pas faire travailler les disques pour des petites opérations en usage courant mais bien par block de tâche à exécuter. Par ailleurs, même si 2GB ne sont pas écrits, si le paramètre checkpoint_timeout est par défaut à 5min, postgresql écrira soit tous les 2GB soit toutes les 5 minutes, le premier de l'un des deux termes atteint.

Valeur recommandée : 10 (et jusqu'à 128 pour 24 Go de RAM totale, en particulier en cas d'imports volumineux).

Constantes de coût du planificateur

random_ page_ cost (floating point)

Initialise l'estimation faite par le planificateur du coût de récupération non-séquentielle d'une page disque.

Mesurée comme un multiple du coût de récupération d'une page séquentielle, sa valeur par défaut est 4.0

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 peut généralement être optimisée.

Réduire cette valeur par rapport au paramètre *seq_page_cost* incite le système à privilégier les parcours d'index.

L'augmenter donne l'impression de parcours d'index plus coûteux.

Valeur recommandée : 2.0

effective_cache_size (integer)

Initialise l'estimation faite par le planificateur de la taille réelle du cache disque disponible pour une requête.

Ce paramètre est lié à l'estimation du coût d'utilisation d'un index : une valeur importante favorise les parcours d'index, une valeur faible les parcours séquentiels.

Ce paramètre n'a pas d'influence sur la taille de la mémoire partagée allouée par PostgreSQL et ne réserve pas non plus le cache disque du noyau, il n'a qu'un rôle estimatif. Le système ne suppose pas non plus que les données reste dans le cache du disque entre des requêtes.

La valeur par défaut est de 128MB.

Valeur recommandée : 512MB (peut être porté à 2/3 de la RAM totale pour un serveur dédié).

Journalisation

Il est recommandé d'activer le journal des opérations.

Attention cependant car le fichier de log doit être analysé et nettoyé régulièrement.

logging_collector = on

On se reportera à la *documentation de PostgreSQL* pour la gestion détaillée.

il est possible d'activer un mécanisme de rotation des fichiers de log avec *log_truncate_on_rotation*.

De nombreux paramètres permettent de définir les événements à tracer. La mise au point est à affiner avec l'administrateur système.

Bonnes pratiques de sécurité

Les informations qui suivent sont extraites du document PostgreSQL/PostGIS : bonnes pratiques et astuces (portail SIG)

http://www.portailsig.org/content/postgresql-postgis-bonnes-pratiques-et-astuces

et du support de formation réalisé par Christophe Masse et Didier Leclerc, CMSIG

Sécurité contre l'intrusion

Bien gérer les utilisateurs (users) et les groupes (rôles) permet de bien sécuriser ses bases, tant contre l'intrusion que contre les erreurs de manipulation.

Les accès peuvent se gérer à plusieurs niveaux (base de données, schéma, table, tablespace, function...).

On peut autoriser à un utilisateur (à tous les niveaux) :

- La lecture (select) : droit de consulter les données
- L'écriture (insert) : droit de créer de nouvelles données
- La modification (update) : droit de modifier des données existantes
- La suppression (delete) : droit de supprimer des données
- La création de clés primaires / étrangères (references) : droit de créer des clés primaires et étrangères.
- La création de triggers (trigger) : droit de créer des triggers. Un trigger est une fonction qui se déclenche lors d'un événement prédéfini.

L'utilisation des rôles et des droits associés permet de gérer de manière très fine les accès et les utilisateurs, permettant ainsi d'assurer l'intégrité des données.

Typiquement, les données de référence seront en lecture seule (sauf pour le ou les administrateurs SIG) et les données métier seront lisibles uniquement pour les services concernés.

La définition des groupes et rôles est réalisée au niveau du serveur par l'administrateur.

La conception de ces rôles et droits d'accès est aussi importante que la conception de la base de

données elle-même.

Faire des sauvegardes (dumps) régulières voire mettre en place un système de réplication

Il est indispensable de mettre en place un système de sauvegarde des bases de données, organisé selon les besoins.

Le plus simple est de faire des dumps réguliers des bases de données.

Il est important de stocker ces dumps sur une autre machine que celle du serveur PostgreSQL pour garantir leur disponibilité en cas de problème.

(par un *point de montage* ou une solution de sauvegarde automatique type arkeia software).

En cas d'incident les données restaurées dateront du moment de la sauvegarde. Il est conseillé d'effectuer ce type de sauvegarde à des moments de faible charge du serveur.

Source documentaire : *http://docs.postgresql.fr/9.5/backup.html*

Il est aussi possible de mettre en place un système de réplication de la base qu'il soit synchrone (en quasi temps-réel) ou asynchrone.

On parle alors de serveurs maîtres et esclaves.

Cette fonctionnalité est disponible par défaut dans PostgreSQL mais il existe aussi des extensions pour faciliter la réplication, la synchronisation ou la répartition des charges, qui peuvent avoir certains avantages (pgCluster, pgpool, slony etc).

Optimisation de la gestion de la base de données

IV

Les informations qui suivent sont extraites du document PostgreSQL/PostGIS : bonnes pratiques et astuces (portail SIG)

http://www.portailsig.org/content/postgresql-postgis-bonnes-pratiques-et-astuces

du support de formation réalisé par Christophe Masse et Didier Leclerc, CMSIG

et de la documentation PostGIS : *http://www.postgis.fr/chrome/site/docs/workshop-foss4g/doc/index.html*

1) Vérifier la validité de la géométrie

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

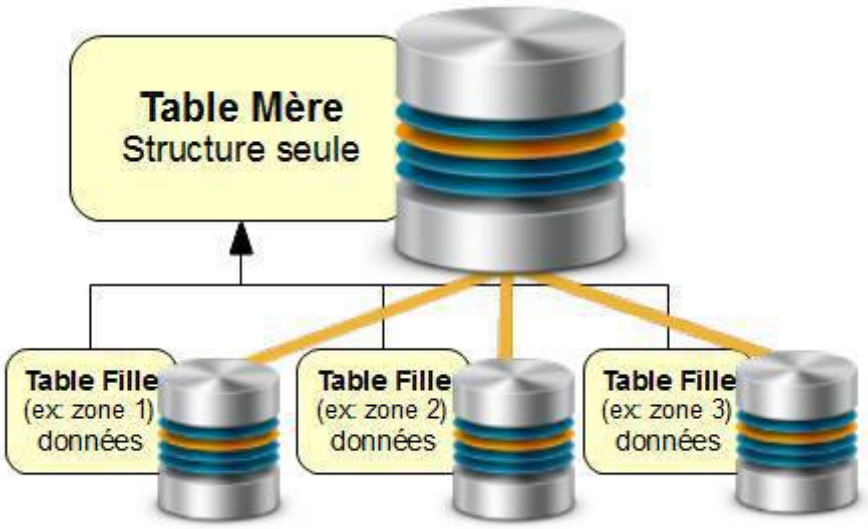
La notion de validité de la géométrie et les méthodes pour corriger les géométries non valides sont exposées dans le *module 4 Compléments SQL - p.23*

2) Répartir les tables volumineuses sur plusieurs tables

Le *partitionnement* consiste à diviser une grande table en plusieurs tables plus petites.

Le partitionnement peut offrir plusieurs avantages :

- Les performances des requêtes peuvent être améliorées de façon spectaculaire dans certaines situations (meilleure efficience des index).
- Le partitionnement devrait être appliqué dès que la taille de la table est supérieure à la mémoire physique du serveur de base de données



Pour plus de précisions, consulter la page : *http://docs.postgresql.fr/9.3/ddl-partitioning.html*

3) Bien gérer les contraintes (clés primaires et étrangères)

Le module 2 Rappels et concepts de base détaille les notions de *clé primaire et étrangère* - *p.22* .

4) Utiliser les index (spatiaux ou non) pour accélérer les requêtes

L'utilisation d'index est une façon habituelle d'améliorer les performances d'une base de données.

Un index permet au serveur de bases de données de retrouver une ligne spécifique bien plus rapidement qu'en parcourant séquentiellement la table.

Les index peuvent aussi bénéficier aux commandes UPDATE et DELETE comportant des conditions de recherche.

De plus, les index peuvent être utilisés dans les jointures. Ainsi, un index défini sur une colonne qui fait partie d'une condition de jointure peut aussi accélérer significativement les requêtes avec jointures.

Mais les index ajoutent aussi une surcharge au système de base de données dans son ensemble, si bien qu'ils doivent être utilisés avec discernement.

Après la création d'un index, le système doit le maintenir synchronisé avec la table. Cela rend plus lourdes les opérations de manipulation de données.

C'est pourquoi les index qui sont peu, voire jamais, utilisés doivent être supprimés.

Attention : la présence d'index ralentit très fortement l'insertion donc il ne faut pas en mettre lorsque cela n'est pas nécessaire.

Source documentaire : *http://docs.postgresql.fr/9.3/indexes.html*

5) Autres actions de maintenance de la base

- Effectuer régulièrement des VACUUM et ANALYSE (auto-vacuum par exemple)
- Effectuer une réindexation régulière
- Utiliser la commande CLUSTER pour les tables géographiques

Ces commandes sont expliquées dans le *module 2 Administration - p.38*

6) Mise à jour de la table système geometry_ columns

Pour les tables et les vues géographiques (PostGIS $<$ 2.0) , maintenir la table système geometry_columns à jour :

Manuellement ou grâce à la fonction Populate_Geometry_Columns().

Cette mise à jour est indispensable pour l'utilisation des couches PostGis dans des logiciels SIG clients.

Remarque

Depuis la version 2.0 de PostGIS, geometry_columns devient une vue présentant la même structure que les tables Geometry_Columns des versions précédentes, mais construite automatiquement depuis les catalogues de la base de données.

Optimisation des requêtes SQL

V

Les conseils qui suivent sont extraits du document PostgreSQL/PostGIS : bonnes pratiques et astuces (portail SIG)

http://www.portailsig.org/content/postgresql-postgis-bonnes-pratiques-et-astuces

et du support de formation réalisé par Christophe Masse et Didier Leclerc, CMSIG

Mettre les conditions (where) les plus "rapides" en premier

L'analyseur de requêtes se charge lui-même d'adopter la stratégie d'exécution de la requête théoriquement la plus rapide.

Il détermine donc l'ordre d'exécution des conditions.

Dans certains cas où l'analyseur n'a pas assez d'éléments pour choisir, mettre en premier la condition la plus "rapide" ou la plus discriminante permet d'optimiser les requêtes.

Utiliser les sous-requêtes

Dans certains cas, l'utilisation de sous-requêtes est plus efficace que les multiples conditions et jointures.

Il faut donc penser à tester l'utilisation de sous-requêtes, d'autant plus qu'elles permettent de décomposer un problème complexe en plusieurs problèmes plus simples

Utilisation des fonctions spatiales préfixées ST_ dans les requêtes spatiales

Les fonctions spatiales de PostGIS (préfixées ST_ pour respecter le standard SQL/MM) comprennent pour la plupart l'appel intégré de l'opérateur spatial &&.

Cet opérateur utilise les index spatiaux pour effectuer un pré-tri résultant de l'intersection des rectangles englobants des objets (Bounding Box) ce qui accélère considérablement l'exécution des requêtes spatiales.

Le tableau ci-dessous précise pour chaque fonction spatiale l'utilisation ou non de l'opérateur spatial && :

Fonction spatiale	**Opérateur && intégré**
ST_Contains	Oui
ST_ContainsProperly	Oui
ST_Covers	Oui
ST_CoveredBy	Oui
ST_Crosses	Oui
ST_Disjoint	Non
ST_Equals	Oui
ST_Intersects	Oui
ST_Overlaps	Oui
ST_Relate	Non
ST_Touches	Oui
ST_Within	Oui

Utilisation de la commande EXPLAIN

PostgreSQL réalise un plan de requête pour chaque requête qu'il reçoit.

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

La commande EXPLAIN permet d'obtenir des informations sur le déroulement de la requête pour évaluer son efficacité.

Cette commande affiche le plan d'exécution que l'optimiseur de PostgreSQL engendre pour l'instruction fournie.

L'utilisation de la commande EXPLAIN a été détaillée dans le *module 5 Aller plus loin - p.32*.

Contenus annexes

> Clef primaire et clefs étrangères

Une clef primaire sert à identifier une ligne d'une table de façon unique. Dans un SGBDR la clef primaire est identique à une contrainte d'unicité et une contrainte NOT NULL, composée de une ou plusieurs colonnes.

Exemple en SQL :

```
CREATE TABLE exemple (a integer, b integer, c integer, PRIMARY KEY(a,b)) ;
```

Une clé primaire indique qu'une colonne ou un groupe de colonnes (dans l'exemple ci-dessus a et b) peut être utilisé(e) comme identifiant unique des lignes de la table. L'ajout d'une clé primaire créera automatiquement un index unique sur la colonne ou le groupe de colonnes utilisé dans la clé primaire. Une table a, au plus, une clé primaire.

Le choix d'une clé primaire est une étape importante dans la conception d'une table. Il peut être nécessaire d'ajouter une colonne qui soit auto-incrémentée.

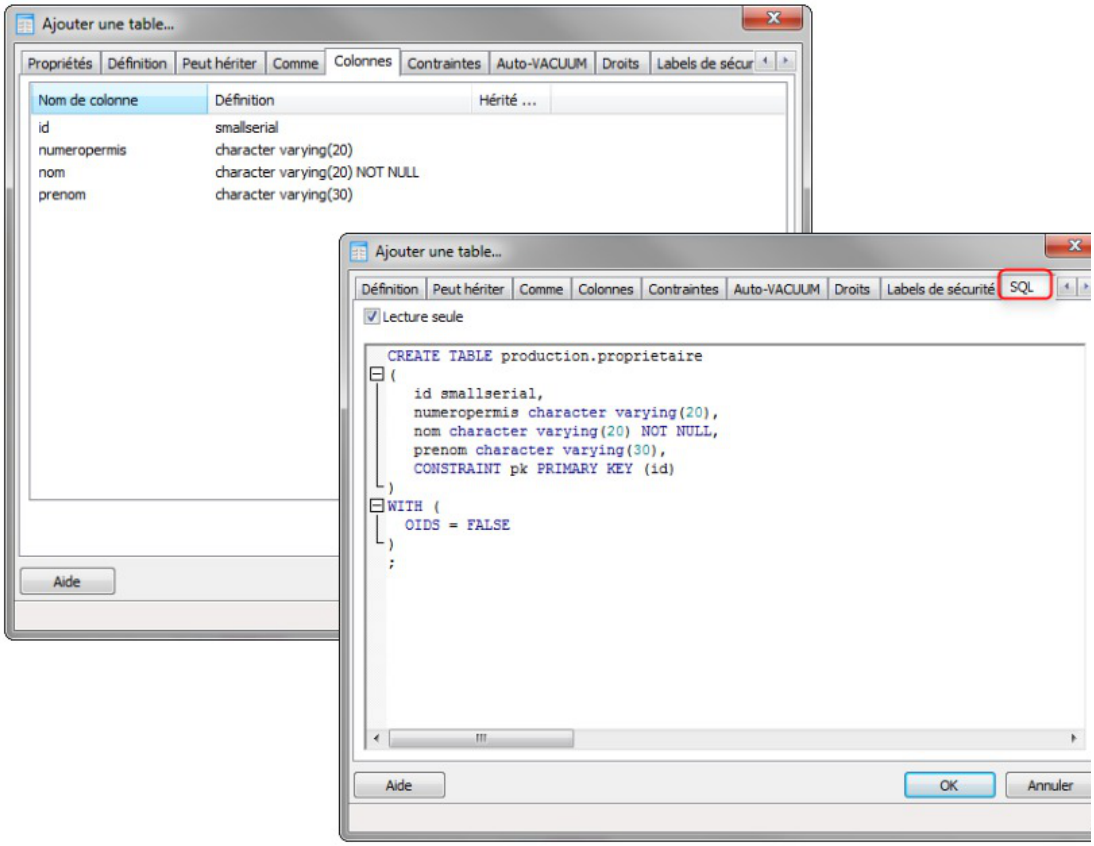
Si on considère pour les besoins de l'exemple, que l'on souhaite pouvoir gérer un propriétaire d'un véhicule même s'il ne possède pas de permis, alors le numéro de permis ne peut être la clef primaire. On va donc définir une nouvelle colonne *id* qui sera incrémentée automatiquement.

(On pourra consulter en complément *cet extrait* de la formation de Stéphane Crozat sur les clefs artificielles et les clefs signifiantes)

Le SQL pour créer la table s'écrira :

```
CREATE TABLE proprietaire (
id SMALLINT AUTO_INCREMENT,
numeropermis VARCHAR(20),
nom VARCHAR(30) NOT NULL,
prenom VARCHAR(30),
PRIMARY KEY (id)
)
```

Nous verrons qu'il est possible de réaliser les requêtes SQL à l'aide des assistants sous PgAdminIII et vérifier la syntaxe SQL dans le dernier onglet de l'assistant :



on notera que *smallserial* et *smallint AUTO_INCREMENT* sont équivalent, de même que *VARCHAR* et *character varying*.

Complément : Les séquences et type sérié

Sous PostgreSQL on peut utiliser les *séquences* pour générer une clef primaire. Voir également *les types sériés*.

Une contrainte de clé étrangère stipule que les valeurs d'une colonne (ou d'un groupe de colonnes) doivent correspondre aux valeurs qui apparaissent dans les lignes d'une autre table. On dit que cela maintient l'**intégrité référentielle** entre les deux tables.

Si nous reprenons notre exemple Propriétaire / véhicule.

Si on considère qu'un véhicule doit toujours avoir un propriétaire préalablement existant, on peut déclarer :

```
CREATE TABLE vehicule (
numeroserie VARCHAR(20) PRIMARY KEY,
type VARCHAR(20),
marque VARCHAR(20),
id_proprietaire SMALLINT REFERENCE proprietaire(id)
)
```

Lors de la saisie d'un nouveau véhicule, le système vérifiera que l'id saisie dans id_proprietaire existe bien dans la table proprietaire.

Une table peut contenir plusieurs contraintes de clé étrangère. Les relations n-m entre tables sont implantées ainsi (voir par exemple l'extrait de la formation de Stéphane Crozat à partir d'*ici*).

voir également *la documentation de PostgreSQL*.

> Correction de géométrie

Notion de validité de la géométrie des entités

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

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

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

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

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

Les deux dernières règles font partie de la catégorie arbitraire.

PostGIS est conforme au standard OGC OpenGIS Specifications.

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

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

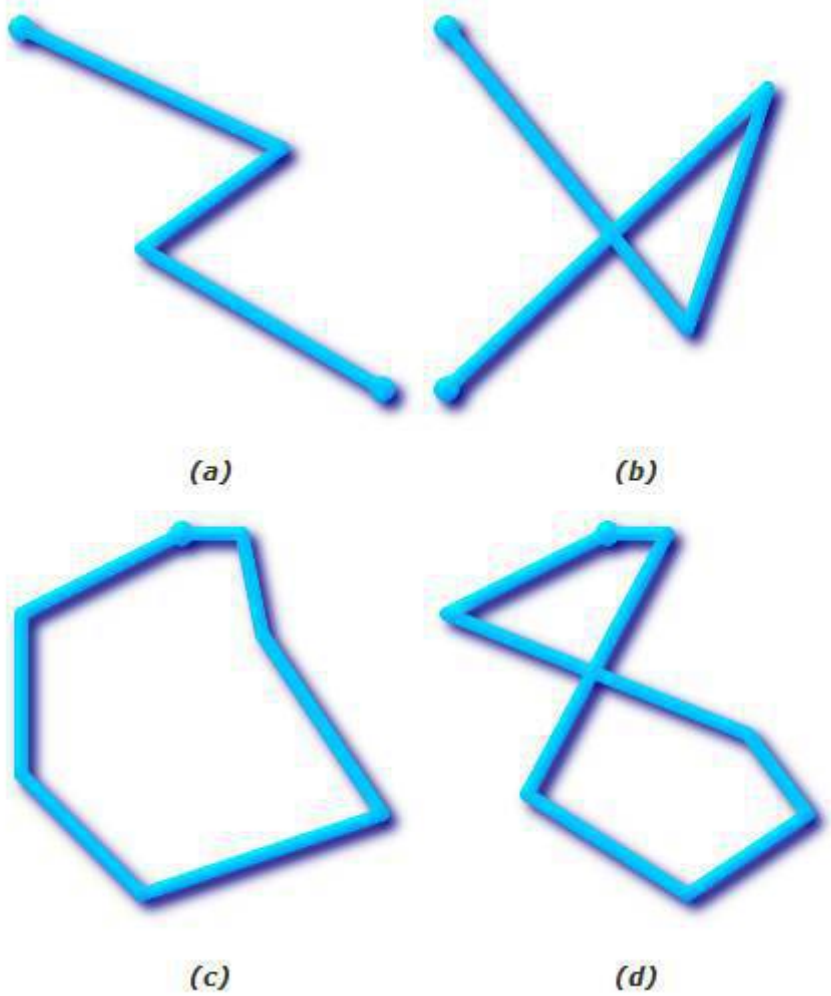
Selon les spécifications de l'OGC, une **géométrie simple** est une géométrie qui ne comporte pas de points géométriques anormaux, comme des auto-intersections ou des auto-tangences, ce qui concerne essentiellement les points, les multi-points, les polylignes et les multi-polylignes.

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

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

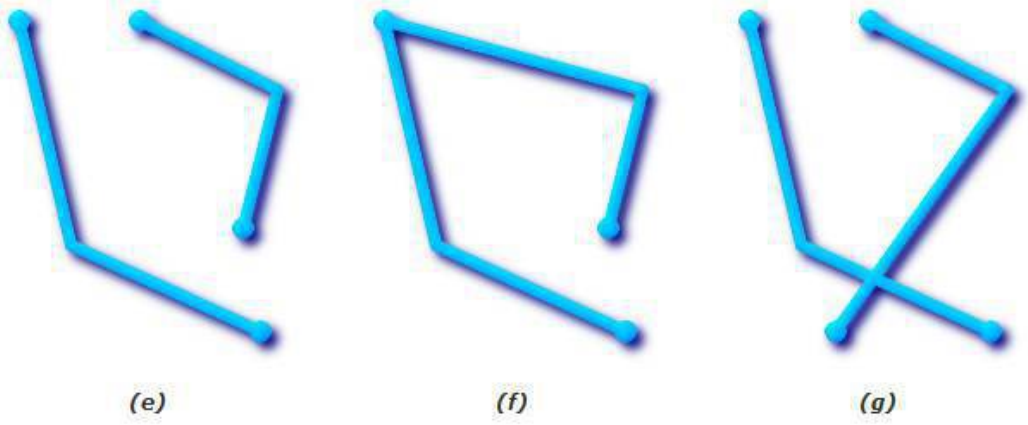
Un objet multi-points est simple si tous les points le composant sont distincts.

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



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

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



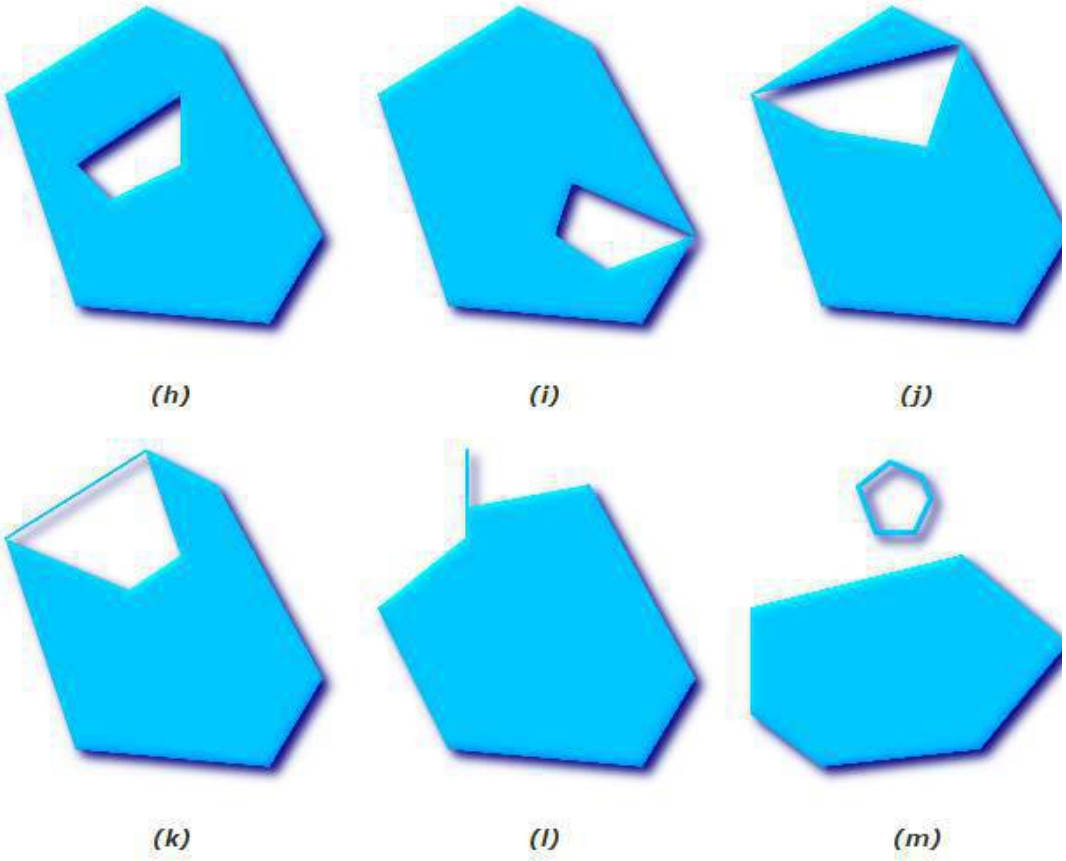
(e) et (f) sont des multipolylignes simples, pas (g)

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

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

Un polygone ne doit pas comporter de lignes interrompues (les limites doivent être continues) ou de point de rebroussement (pic).

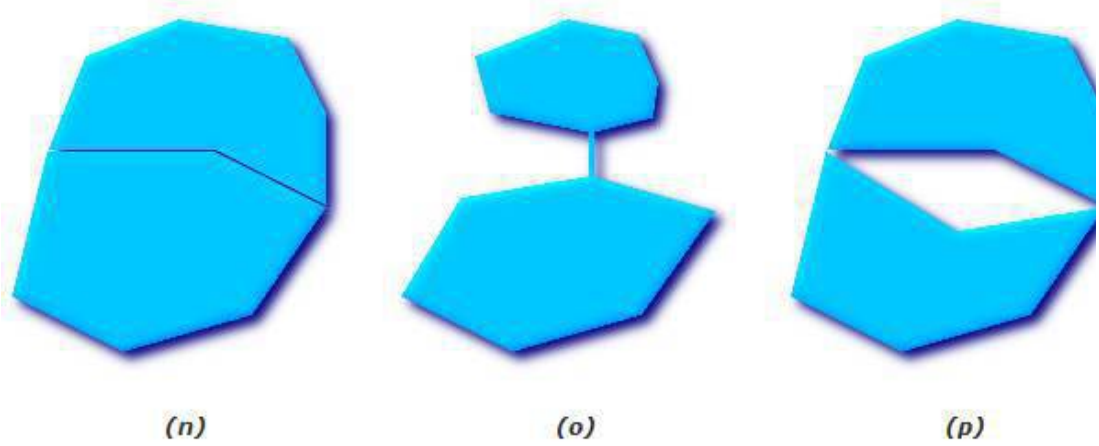
Les anneaux intérieurs doivent être entièrement contenus dans la limite extérieure du polygone.



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

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

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



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

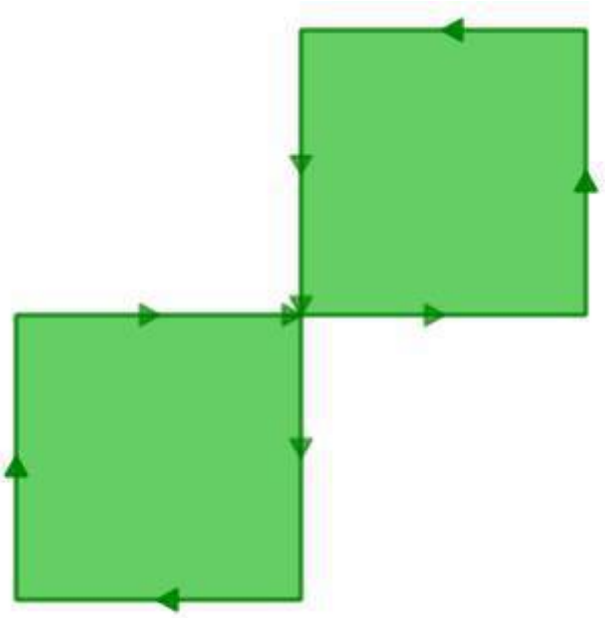
Par défaut, PostGIS n'applique pas le test de validité géométrique lors de l'import d'entités géométriques, parce que le test de validité géométrique consomme beaucoup de temps processeur pour les géométries complexes, notamment les polygones.

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

Exemple

Exemple :

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



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

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

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

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

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

Que ce passe-t-il ici ?

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

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

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

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

- Détecter la validité

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

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

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

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

résultat : f (false)

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

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

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

résultat : Self-intersection[1 1]

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

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

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

Détecter et corriger les erreurs géométriques par la pratique

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

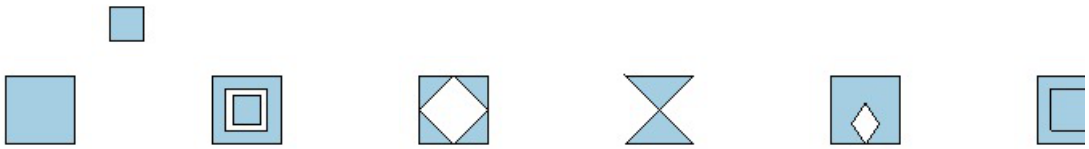
Pour créer une table de géométries invalides dans le schéma travail:

exécuter le script SQL ci-dessous :

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

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

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

INSERT INTO travail.invalidgeometry (type, geom) VALUES ('Dis. Interior', ST_Multi(ST_GeomFromText('POLYGON((465060 6700000, 465070 6700000,465070 6700010, 465060 6700010, 465060 6700000), (465065 6700000, 465070 6700005, 465065 6700010, 465060 6700005, 465065 6700000))', 2154)));

INSERT INTO travail.invalidgeometry (type, geom) VALUES ('Self Intersect.', ST_multi(ST_GeomFromText('POLYGON((465090 6700000, 465100 6700010, 465090 6700010, 465100 6700000, 465090 6700000))',2154)));

INSERT INTO travail.invalidgeometry (type, geom) VALUES ('Ring Self Intersect.', ST_multi(ST_GeomFromText('POLYGON((465125 6700000, 465130 6700000, 465130 6700010, 465120 6700010, 465120 6700000, 465125 6700000, 465123 6700003, 465125 6700006, 465127 6700003, 465125 6700000))',2154)));
```

```
INSERT INTO travail.invalidgeometry (type, geom) VALUES ('Nested
Shells', ST_multi(ST_GeomFromText('MULTIPOLYGON(((465150 6700000,
465160 6700000, 465160 6700010, 465150 6700010, 465150
6700000)),(( 465152 6700002, 465158 6700002, 465158 6700008,
465152 6700008, 465152 6700002)))',2154)));
```

Visualiser la table dans QGIS :



Chacun de ces objets est invalide.

Vérifions-le avec la requête suivante :

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

qui nous renvoi :

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

Méthode : Correction avec ST_MakeValid()

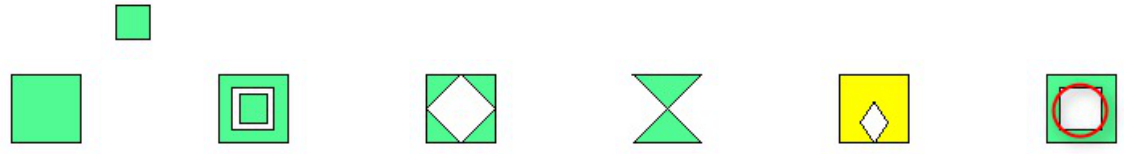
Executer le script SQL :

```
CREATE TABLE travail.makevalidgeometry AS
( S E L E C T i d , t y p e ,
ST_MULTI(ST_MakeValid(geom))::geometry(MULTIPOLYGON, 2154) AS geom
FROM travail.invalidgeometry WHERE NOT St_IsValid(geom));
```

qui renvoi :

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

Charger la nouvelle table dans QGIS :



Seul le dernier élément est différent :

constater que la requête :

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

Ne trouve plus d'erreur.

Exécuter :

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

qui renvoi :

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

"MULTIPOLYGON(((465030 6700000,465030 6700010,465040 6700010,465040 6700000,465030 6700000),(465032 6700002,465038 6700002,465038 6700008,465032 6700008,465032 6700002)),((465033 6700003,465033 6700007,465037 6700007,465037 6700003,465033 6700003)))"

"MULTIPOLYGON(((465065 6700000,465060 6700000,465060 6700005,465065 6700000)),((465060 6700005,465060 6700010,465065 6700010,465060 6700005)),((465065 6700010,465070 6700010,465070 6700005,465065 6700010)),((465070 6700005,465070 6700000,465065 6700000,4650 (...)"

"MULTIPOLYGON(((465090 6700000,465095 6700005,465100 6700000,465090 6700000)),((465095 6700005,465090 6700010,465100 6700010,465095 6700005)))"

"MULTIPOLYGON(((465125 6700000,465120 6700000,465120 6700010,465130 6700010,465130 6700000,465125 6700000),(465125 6700000,465127 6700003,465125 6700006,465123 6700003,465125 6700000)))"

"MULTIPOLYGON(((465150 6700000,465150 6700010,465160 6700010,465160 6700000,465150 6700000),(465152 6700002,465158 6700002,465158 6700008,465152 6700008,465152 6700002)))"
```

On peut comparer avec la table de départ.

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

renvoi :

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

"MULTIPOLYGON(((465030 6700000,465040 6700000,465040 6700010,465030 6700010,465030 6700000),(465032 6700002,465032 6700008,465038 6700008,465038 6700002,465032 6700002),(465033 6700003,465033 6700007,465037 6700007,465037 6700003,465033 6700003)))"

"MULTIPOLYGON(((465090 6700000,465100 6700010,465090 6700010,465100 6700000,465090 6700000)))"

"MULTIPOLYGON(((465125 6700000,465130 6700000,465130 6700010,465120 6700010,465120 6700000,465125 6700000,465123 6700003,465125 6700006,465127 6700003,465125 6700000)))"

"MULTIPOLYGON(((465150 6700000,465160 6700000,465160 6700010,465150 6700010,465150 6700000)),((465152 6700002,465158 6700002,465158 6700008,465152 6700008,465152 6700002)))"

"MULTIPOLYGON(((465060 6700000,465070 6700000,465070 6700010,465060 6700010,465060 6700000),(465065 6700000,465070 6700005,465065 6700010,465060 6700005,465065 6700000)))"
```

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

'Nested Holes' (trous imbriqués) : est devenu un multipolygone composé d'un polygone avec

trou et d'un deuxième polygone qui est au centre (le plus petit).

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

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

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

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

Complément

il existe l'algorithme **Fix invalid polygons (STMakevalid)** dans Processing (menu traitement) pour réaliser cette opération.

ST_Multi() permet de convertir en MULTIPOLYGON si la géométrie est de type POLYGON.

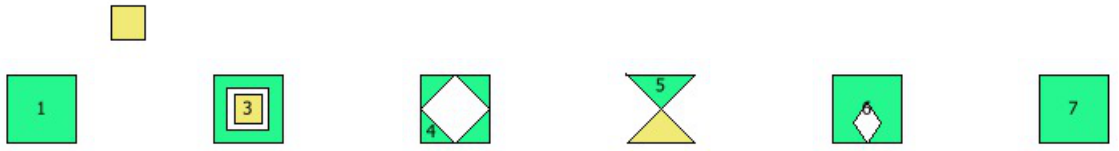
Méthode : Correction par ST_Buffer(geom,0)

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

Executer le script suivant :

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

Charger cette couche dans QGIS :



Constater que les parties de polygones en jaune ont disparu et que le polygone 7 a été corrigé différemment.

Cette méthode n'est donc pas recommandée, sauf cas particulier (ex : cas 7). En effet on perd des parties de polygones sans maîtrise.

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

Complément

Il existe l'algorithme **Fix invalid polygons (ST_Buffer)** dans Processing (menu traitement) pour réaliser cette opération. (il faut charger le plugin **PostGIS Geoprocessing tools** qui offre par ailleurs d'autres fonctions intéressantes pour les traitement des couches sous PostGIS et qui permet le cas échéant de masquer la syntaxe SQL des commandes effectuant les-dits traitements aux utilisateurs).

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

Il existe d'autres solutions pour experts, comme l'utilisation de *fonctions de GRASS* (v.in.ogr et v.clean) . A l'avenir il est possible que le plugin *Geometry Cleaning* de sourcepole soit intégré à QGIS.

> L'optimisation des requêtes avec EXPLAIN

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

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

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

Vous pouvez utiliser la commande ***EXPLAIN*** pour voir quel plan de requête le planificateur crée pour une requête particulière.

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

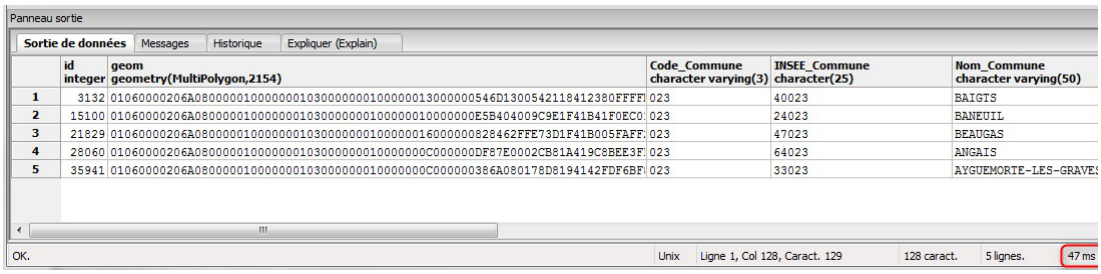
Le cours de Stéphane CROZAT constitue également une bonne introduction non orienté sur la géomatique.

Exemple d'optimisation avec création d'index

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

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

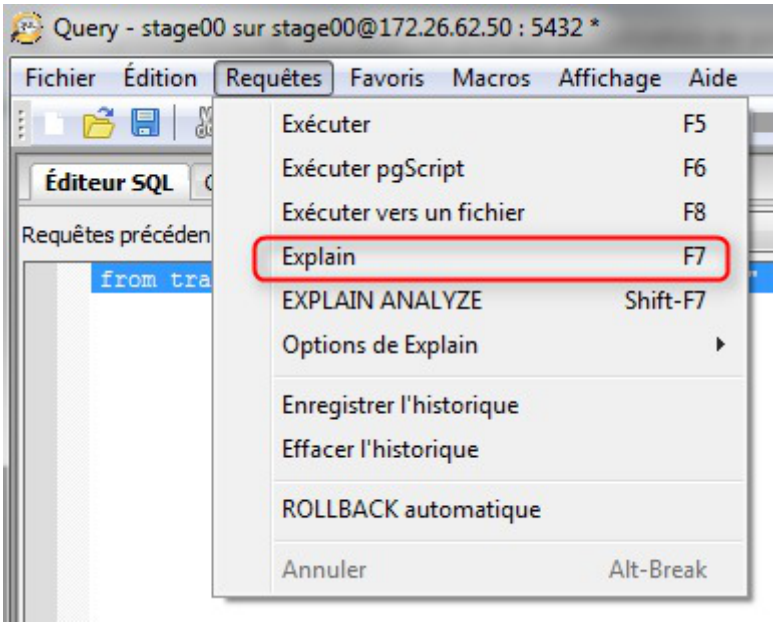
noter le temps qui apparaît en bas à droite du panneau de sortie :



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

Dans notre cas le temps d'exécution moyen est d'environ 45 ms.

Utilisez maintenant la commande EXPLAIN du menu Requête :



vous devez voir apparaître dans l'onglet 'expliquer'

et dans l'onglet 'sortie de données' :

« "Seq Scan on "FR_communes" (cost=0.00..2808.01 rows=5 width=520)" »

« " Filter: ((("Code_Commune")::text = '023'::text) AND (("Statut")::text = 'Commune simple'::text) AND (("Nom_Région")::text = 'AQUITAINE'::text))" »

Ceci indique que le moteur SQL de PostgreSQL exécute un balayage séquentiel (Seq scan) de la table "FR_communes".

Le coût (cost) donne deux chiffres :

le premier est le coût estimé de lancement, ici 0.

le deuxième est le coût total estimé du scan, ici 2808.01 (les coûts sont estimés en *unité arbitraire*)

rows indique que 5 lignes sont retournées.

et with la longueur estimées en octets des lignes.

EXPLAIN founi des données estimées.

On peut exécuter EXPLAIN ANALYZE qui lui va exécuter la requête et fournir les coûts mesurés. Par exemple dans notre cas :

« "Seq Scan on "FR_communes" (cost=0.00..2808.01 rows=5 width=520) (actual time=19.574..667.932 rows=5 loops=1)" »

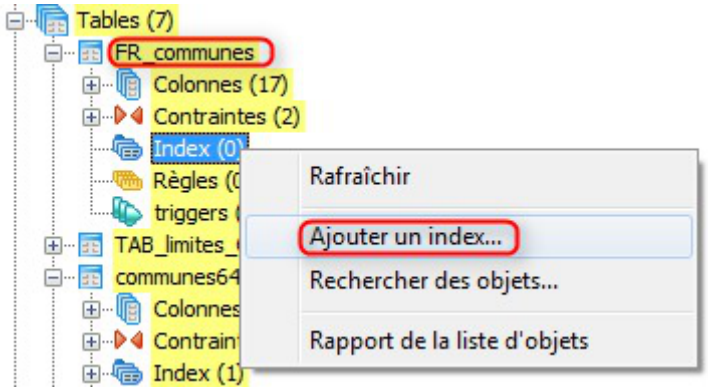
« " Filter: ((("Code_Commune")::text = '023'::text) AND (("Statut")::text = 'Commune simple'::text) AND (("Nom_Région")::text = 'AQUITAINE'::text))" »

« " Rows Removed by Filter: 36224" »

« "Total runtime: 668.013 ms" »

ici le coût mesuré est égal au coût estimé de 2808.01.

Ajoutons maintenant un index sur le champs "Statut"

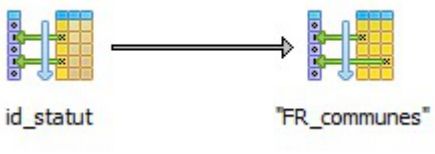


puis, réexécuter la requête SQL

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

Pour nous 31 ms.

Relancer le EXPLAIN



Constater que la sortie de données du EXPLAIN est maintenant :

« "Bitmap Heap Scan on "FR_communes" (cost=4.94..299.15 rows=5 width=520)" »

« " Recheck Cond: (("Code_Commune")::text = '023'::text)" »

« " Filter: ((("Statut")::text = 'Commune simple'::text) AND (("Nom_Région")::text = 'AQUITAINE'::text))" »

« " -> Bitmap Index Scan on id_statut (cost=0.00..4.94 rows=87 width=0)" »

« " Index Cond: (("Code_Commune")::text = '023'::text)" »

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

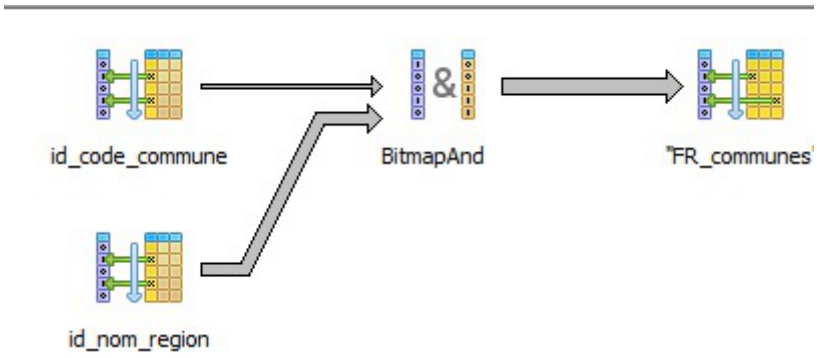
Le coût total est de 299.15 et donc très inférieur au 2808.01 sans l'index.

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

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

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

relancer la requête et le EXPLAIN :



la sortie est maintenant :

« "Bitmap Heap Scan on "FR_communes" (cost=58.77..81.92 rows=5 width=520)" »

« " Recheck Cond: ((("Code_Commune")::text = '023'::text) AND (("Nom_Région")::text = 'AQUITAINE'::text))" »

« " Filter: (("Statut")::text = 'Commune simple'::text)" »

« " -> BitmapAnd (cost=58.77..58.77 rows=6 width=0)" »

« " -> Bitmap Index Scan on id_code_commune (cost=0.00..4.94 rows=87 width=0)" »

« " Index Cond: (("Code_Commune")::text = '023'::text)" »

« " -> Bitmap Index Scan on id_nom_region (cost=0.00..53.57 rows=2304 width=0)" »

« " Index Cond: (("Nom_Région")::text = 'AQUITAINE'::text)" »

Le planificateur a choisi une stratégie :

1. balayage de l'index sur le champ *Nom_Région* en 53.57
2. balayage de l'index *id_commune* en 4.94
3. ET (Bitmap And) entre les deux résultats ci-dessus en 0
4. balayage des lignes de la table *FR_Communes* correspondantes en ajoutant le filtre sur le Statut en 23.15

Pour un total de 81.92.

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

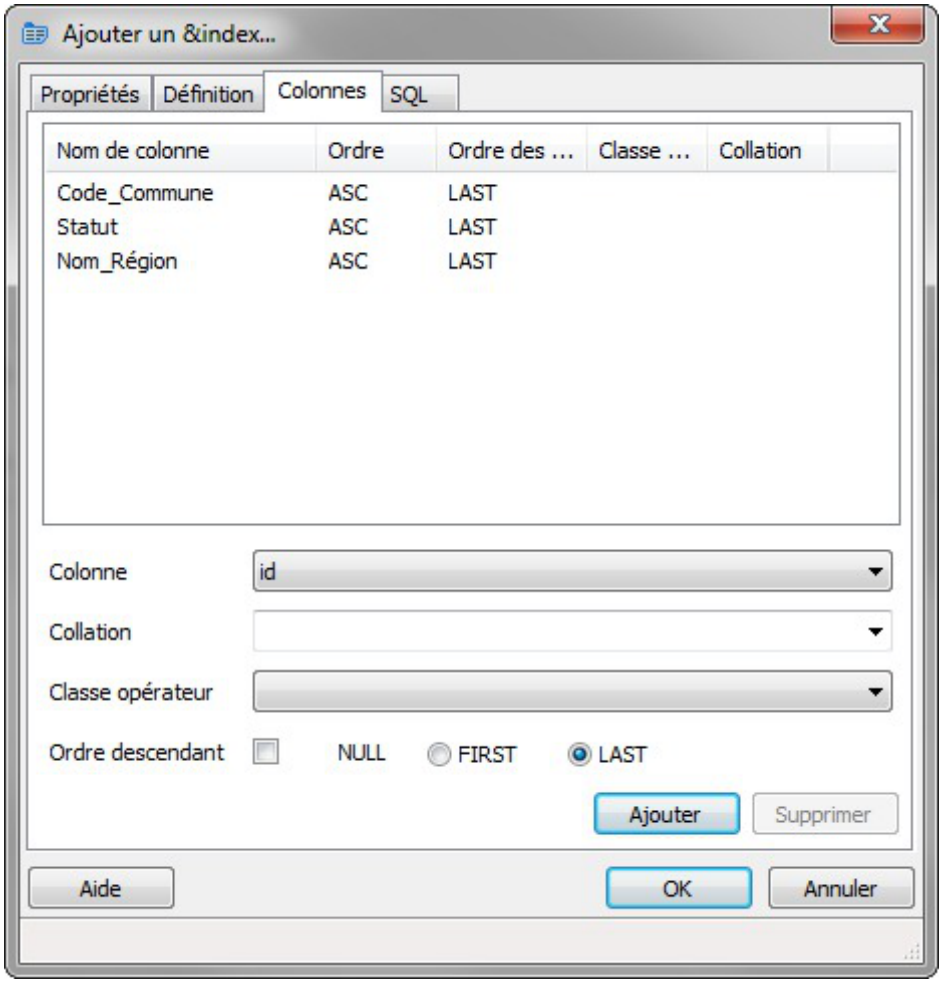
Une commande **VACCUM ANALYZE** que l'on peut lancer sur la base *stageXX* à partir du menu *Outils -> Maintenance*

permet de remettre à jour les statistiques sur les tables.

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

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

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



Relancer l'analyse sur la requête :

« "Bitmap Heap Scan on "FR_ communes" (cost=4.44..12.29 rows=2 width=442)" »

« " Recheck Cond: ((("Code_ Commune")::text = '023'::text) AND (("Statut")::text = 'Commune simple'::text) AND (("Nom_ Région")::text = 'AQUITAINE'::text))" »

« " -> Bitmap Index Scan on id_ index_ triple (cost=0.00..4.44 rows=2 width=0)" »

« " Index Cond: ((("Code_ Commune")::text = '023'::text) AND (("Statut")::text = 'Commune simple'::text) AND (("Nom_ Région")::text = 'AQUITAINE'::text))" »

La stratégie est maintenant de balayer l'index composite en 4.44, puis de rechercher les enregistrements correspondant (Bitmap Heap scan) dans la table *FR_ communes* pour un coût total de 12.29

A comparer avec les résultats précédents.

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

Complément

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

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

Complément

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

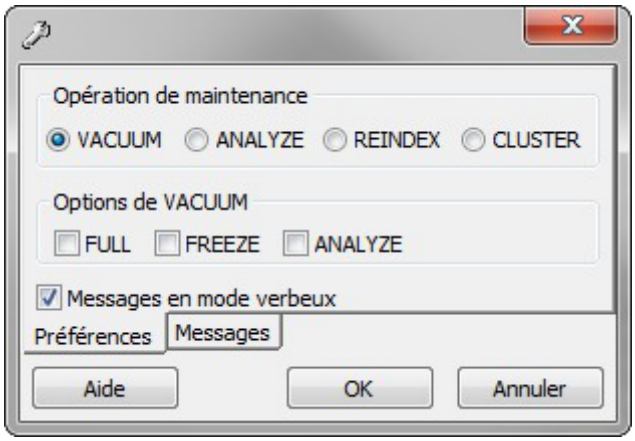
A noter également un *site* qui permet de mettre en exergue les parties les plus coûteuses d'un plan d'analyse.

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

> Opérations de maintenance sur les bases

Sous PgAdmin il est possible par clic droit sur une base d'utiliser le menu Maintenance...

celui-ci permet de lancer différentes opération de maintenance.



Méthode : VACUUM

VACUUM récupère l'espace inutilisé et, optionnellement, analyse une base.

VACUUM récupère l'espace de stockage occupé par des lignes mortes. Lors des opérations normales de PostgreSQL, les lignes supprimées ou rendues obsolètes par une mise à jour ne sont pas physiquement supprimées de leur table. Elles restent présentes jusqu'à ce qu'un VACUUM soit lancé. C'est pourquoi, il est nécessaire de faire un VACUUM régulièrement, spécialement sur les tables fréquemment mises à jour.

Sans paramètre, VACUUM traite toutes les tables de la base de données courante pour lequel l'utilisateur connecté dispose du droit d'exécution du VACUUM. Avec un paramètre, VACUUM ne traite que cette table.

Les options :

- **FULL** récupère plus d'espace (compactage des tables), mais est beaucoup plus long et prend un verrou exclusif sur la table (inaccessible à d'autres requêtes pendant le VACCUM). Cette méthode requiert aussi un espace disque supplémentaire, car elle écrit une nouvelle copie de la table et ne supprime l'ancienne copie qu'à la fin de l'opération. Habituellement, cela doit seulement être utilisé quand une quantité importante d'espace doit être récupérée de la table. .
- **FREEZE :** les versions des lignes sont gelées si elles sont suffisamment vieilles pour être visibles de toutes les transactions en cours. En particulier, sur une base en lecture

seulement, VACUUM FREEZE aura pour résultat de geler toutes les lignes de la base. Donc, tant que la base n'est pas modifiée, aucun nettoyage supplémentaire n'est nécessaire.

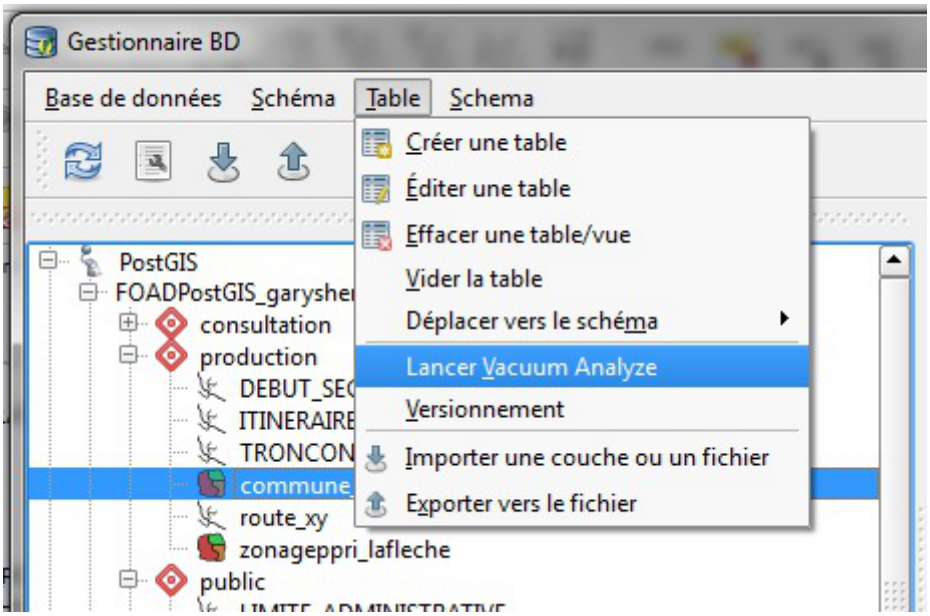
- **ANALYZE :** Met à jour les statistiques utilisées par l'optimiseur pour déterminer la méthode la plus efficace pour exécuter une requête.

Pour exécuter un VACUUM sur une table, vous devez habituellement être le propriétaire de la table ou un superutilisateur. Néanmoins, les propriétaires de la base de données sont autorisés à exécuter VACUUM sur toutes les tables de leurs bases de données, sauf sur les catalogues partagés. Cette restriction signifie qu'un vrai VACUUM sur une base complète ne peut se faire que par un superutilisateur.)

Il est recommandé que les bases de données actives de production soient traitées par VACUUM fréquemment (au moins toutes les nuits), pour supprimer les lignes mortes. Après avoir ajouté ou supprimé un grand nombre de lignes, il peut être utile de faire un VACUUM ANALYZE sur la table affectée. Cela met les catalogues système à jour de tous les changements récents et permet à l'optimiseur de requêtes de PostgreSQL™ de faire de meilleurs choix lors de l'optimisation des requêtes

PostgreSQL inclut un « *autovacuum* » qui peut automatiser la maintenance par VACUUM. Certains administrateurs de bases de données voudront suppléer ou remplacer les activités du démon avec une gestion manuelle des commandes VACUUM, qui seront typiquement exécutées suivant un planning par des scripts cron ou par le planificateur de tâches.

DBManager permet de lancer un VACUUM ANALYZE sur une table donnée.



À noter que le VACCCUM est également disponible pour l'administrateur système sous forme de la commande système vacuumdb.

Méthode : ANALYZE

ANALYZE collecte des statistiques sur le contenu des tables de la base de données et stocke les résultats dans le catalogue système *pg_ statistic*.

L'optimiseur de requêtes les utilise pour déterminer les plans d'exécution les plus efficaces.

ANALYZE peut être exécuté comme une option de VACUUM.

Méthode : REINDEX

REINDEX reconstruit un index en utilisant les données stockées dans la table, remplaçant l'ancienne copie de l'index. Il y a plusieurs raisons pour utiliser REINDEX :

- Un index a été corrompu et ne contient plus de données valides
- L'index en question a « explosé », c'est-à-dire qu'il contient beaucoup de pages d'index mortes ou presque mortes

- Vous avez modifié un paramètre de stockage (par exemple, fillfactor) pour un index et vous souhaitez vous assurer que la modification a été prise en compte.
- La construction d'un index avec l'option `CONCURRENTLY` a échoué, laissant un index « invalide »

Dans les vieilles versions de PostgreSQL, le gain avec REINDEX peut-être énorme.

Nota bene : disponible pour l'administrateur système avec la commande *reindexdb*.

Méthode : CLUSTER

CLUSTER commande permet de réécrire les données d'une table dans un ordre donné (suivant un index). Cela peut être utile pour optimiser des requêtes SQL, selon la façon dont les données sont accédées.

Pour des tables spatiales, elle permet d'ordonner physiquement la table selon l'index géométrique et permet donc un accès aux enregistrements "proches" plus rapide.

Attention toutefois : cette opération reste lourde puisqu'elle pause un verrou exclusif sur la table traitée (aucune autre action possible sur la table durant ce temps).