DALIBO
L'expertise PostgreSQL

Optimisations

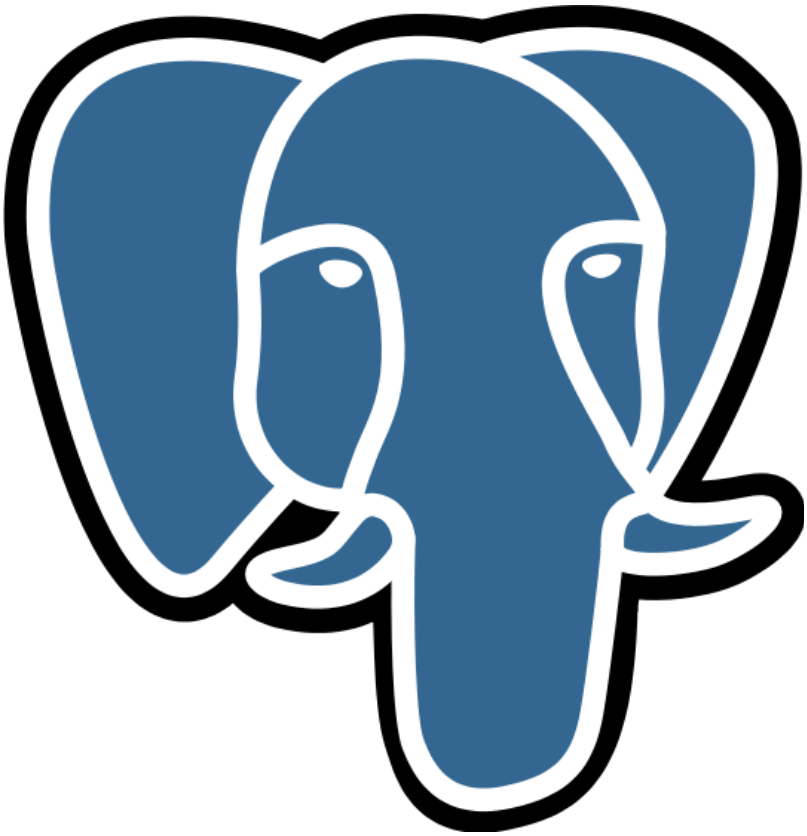
Ce document est téléchargeable gratuitement dans la base de connaissance DALIBO.

Pour toute information complémentaire, contactez :
formation@dalibo.com

Table des matières

PostgreSQL : Optimisations....3
1 Introduction....4
1.1 Menu....4
1.2 Licence Creative Commons CC-BY-NC-SA....5
1.3 Pré-requis (1/2)....6
1.4 Pré-requis (2/2)....6
2 Matériel....8
2.1 CPU....8
2.2 RAM....10
2.3 Disques....10
2.4 RAID....12
2.5 SAN....13
2.6 Virtualisation 1/2....14
2.7 Virtualisation 2/2....16
3 Système d'exploitation....17
3.1 Choix du système d'exploitation....17
3.2 Choix du noyau....18
3.3 Configuration du noyau....19
3.4 Configuration de la mémoire partagée....19
3.5 Contrôle du cache disque système....21
3.6 Configuration du OOM....22
3.7 Configuration du swap....23
3.8 Configuration de l'affinité processeur / mémoire....23
3.9 Comment les configurer....25
3.10 Choix du système de fichiers....26
3.11 Configuration du système de fichiers....27
4 Serveur de bases de données....29
4.1 Version....29
4.2 Configuration - mémoire....30
4.3 Configuration - planificateur....32
4.4 Configuration - WAL....33
4.5 Configuration - statistiques....34
4.6 Configuration - autovacuum....34
4.7 pgtune....35
4.8 Outil pgbench....37
4.9 Environnement de test avec pgbench....38
4.10 Environnement réel avec pgbench....40
4.11 Emplacement des journaux de transactions....41
4.12 Emplacement des fichiers de données....42
5 Conclusion....44
5.1 Questions....44

PostgreSQL : Optimisations



1 Introduction

- L'optimisation doit porter sur les différents composants
  - Le serveur qui héberge le SGBDR : le matériel, la distribution et le kernel
  - Le moteur de la base de données : postgresql.conf
  - La base de données : l'organisation des fichiers de PostgreSQL
  - L'application en elle-même : le schéma et les requêtes

Pour qu'une optimisation soit réussie, il faut absolument tenir compte de tous les éléments ayant une responsabilité dans les performances. Cela commence avec le matériel. Il ne sert à rien d'améliorer la configuration du serveur PostgreSQL ou les requêtes si, physiquement, le serveur ne peut tenir la charge, que cela soit la cause des processeurs, de la mémoire, du disque ou du réseau. Le matériel est donc un point important à vérifier dans chaque recherche de performances. De même, le système d'exploitation est pour beaucoup dans les performances de PostgreSQL : son choix et sa configuration ne doivent pas être laissés au hasard. La configuration du moteur a aussi son importance et ce module permettra de faire la liste des paramètres importants dans le seul cadre des performances. Même l'organisation des fichiers dans les partitions des systèmes disques a un intérêt.

L'optimisation (aussi appelé `tuning`) doit donc être réalisée sur tous ces éléments **à la fois** pour être optimal ! De même, ces optimisations font partie d'un processus **itératif**.

1.1 Menu

- Partie 1 : Matériel
- Partie 2 : Système d'exploitation
- Partie 3 : Serveur de bases de données

1.2 Licence Creative Commons CC-BY-NC-SA

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Cette formation (diapositives, manuels et travaux pratiques) est sous licence **CC-BY-NC-SA**.

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre).

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web.

Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre.

Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible à cette adresse:
http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

1.3 Pré-requis (1/2)

- Deux points déterminants:
  - Vision globale du système d'information
  - Compréhension de l'utilisation de la base

Il est très difficile d'optimiser un serveur de bases de données sans savoir comment ce dernier va être utilisé. Par exemple, le nombre de requêtes à exécuter simultanément et leur complexité est un excellent indicateur pour mieux apprécier le nombre de cœurs à placer sur un serveur. Il est donc important de connaître la façon dont les applications travaillent avec les bases. Cela permet de mieux comprendre si le matériel est adéquat, s'il faut changer telle ou telle configuration, etc. Cela permet aussi de mieux configurer son système de supervision.

1.4 Pré-requis (2/2)

- L'optimisation n'est pas un processus unique
  - il s'agit au contraire d'un processus itératif
- La base doit être surveillée régulièrement !

Après avoir installé le serveur et l'avoir optimisé du mieux possible, la configuration optimale réalisée à ce moment ne sera bonne que pendant un certain temps. Si le service gagne en popularité, le nombre d'utilisateurs peut augmenter. La base va de toute façon grossir. Autrement dit, les conditions initiales vont changer. Un serveur optimisé pour dix utilisateurs en parallèle ne le sera plus pour 50 utilisateurs en parallèle. La configuration d'une base de 10 Go n'est pas la même que celle d'une base de 1 To.

Cette évolution doit donc être surveillée à travers un système de supervision et métrologie approprié et compris. Lorsqu'un utilisateur se plaint d'une impression de lenteur sur le système, ces informations collectées rendent parfois la tâche d'inspection plus rapide. Ainsi, l'identification du ou des paramètres modifier, ou plus généralement des actions réaliser pour corriger le problème est plus aisée et repose sur une vision fiable et réelle de l'activité de l'instance.

Le plus important est donc de bien comprendre qu'un SGBD ne s'optimise pas qu'une seule fois, mais que ce travail d'optimisation sera à faire plusieurs fois.

À une échelle beaucoup plus petite, un travail d'optimisation sur une requête peut forcer à changer la configuration d'un paramètre. Cette modification peut

faire gagner énormément sur cette requête… et perdre encore plus sur les autres. Là-aussi, tout travail d'optimisation doit être fait prudemment et ses effets surveillés sur une certaine période pour s'assurer que cette amélioration ne s'accompagne pas de quelques gros inconvénients.

2 Matériel

- Performances très liées aux possibilités du matériel
- Quatre composants essentiels
  - les processeurs
  - la mémoire
  - les disques
  - le système disque (RAID, SAN)

PostgreSQL est un système qui se base fortement sur le matériel et le système d'exploitation. Il est donc important que ces deux composants soient bien choisis et bien configurés pour que PostgreSQL fonctionne de façon optimale pour les performances.

Au niveau du matériel, les composants essentiels sont :

- les processeurs (CPU) ;
- la mémoire (RAM) ;
- les disques ;
- le système disque (carte RAID, baie SAN, etc).

2.1 CPU

- Trois critères importants
  - nombre de cœurs
  - fréquence
  - cache
- Privilégier le nombre de cœurs ou la fréquence
- 64 bits

PostgreSQL est un système multi-processus. Chaque connexion d'un client est gérée par un processus, responsable de l'exécution des requêtes et du renvoi des données au client. Ce processus n'est pas multi-threadé. Par conséquent, chaque requête exécutée est traitée par un cœur de processeur. Plus vous voulez pouvoir exécuter de requêtes en parallèle, plus vous devez avoir de

processeurs (ou plus exactement de cœurs). On considère habituellement qu'un cœur peut traiter de 4 à 20 requêtes simultanément. Cela dépend notamment beaucoup des requêtes, de leur complexité, de la quantité de donnée manipulée et retournée, etc. Il est donc essentiel de connaître le nombre de requêtes traitées simultanément pour le nombre d'utilisateurs connectés. S'il s'agit d'un SGBD pour une application web, il y a de fortes chances que le nombre de requêtes en parallèle soit assez élevé. Dans ce contexte, il faut prévoir un grand nombre de cœur processeurs. En revanche, sur un entrepôt de données, nous trouvons habituellement peu d'utilisateurs avec des requêtes complexes et gourmandes en ressources. Dans ce cas, beaucoup de processeurs n'apporteront rien. Mieux vaut peu de cœur, mais que ces derniers soient plus puissants afin de répondre plus efficacement aux besoins important de calculs.

Ainsi, la fréquence (et donc la puissance) des processeurs est un point important à considérer. Il peut faire la différence si les requêtes à exécuter sont complexes : temps de planification réduit, calculs plus rapides donc plus de requêtes exécutées sur une période de temps donnée. Généralement, un système utilisé pour des calculs (financiers, scientifiques, géographiques) a intérêt à avoir des processeurs à fréquence élevée.

Le cache processeur est une mémoire généralement petite mais excessivement rapide et située au plus près du processeur. Il en existe plusieurs niveaux. Tous les processeurs ont un cache de niveau L2, certains ont même un cache de niveau L3. Plus cette mémoire est importante, plus le processeur peut conserver de données utiles et éviter des aller-retours en mémoire RAM coûteux en temps. Le gain en performance pouvant être important, le mieux est de privilégier les processeurs avec beaucoup de cache.

Le choix processeur se fait donc suivant le type d'utilisation du serveur :

- une majorité de petites requêtes en très grande quantité : privilégier le nombre de cœurs ;
- une majorité de grosses requêtes en très petite quantité : privilégier la puissance processeurs.

Dans tous les cas, choisissez la version des processeurs avec le plus de mémoire cache embarquée.

La question 32 bits/64 bits ne se pose plus : il n'existe pratiquement plus que du 64 bits. De plus, les processeurs 64 bits sont naturellement plus performant pour traiter des données sur 8 octets (bigint, double precision, numeric, timestamps, etc) qui tiennent dans un registre mémoire.

Il existe une autre question qui ne se pose plus tellement : vaut-il mieux Intel ou AMD ? cela a très peu d'importance. AMD a une grande maîtrise des systèmes multi-cœurs, et Intel est souvent puissant et optimisé sur les échanges avec la mémoire. Cela pourrait être des raisons de les sélectionner, mais la différence devient de plus en plus négligeable de nos jours.

2.2 RAM

- Essentiel pour un serveur de bases de données
- Plus il y en a, mieux c'est
- Pour le système comme pour PostgreSQL

Toute opération sur les données doit se faire en mémoire. Il est donc nécessaire qu'une bonne partie de la base tienne en mémoire, ou tout du moins la partie active. La partie passive est rarement présente en mémoire car généralement composée de données historiques qui sont peu ou pas lues et jamais modifiées.

Un cache disque permet de limiter les accès en lecture et écriture vers les disques. L'optimisation des accès aux disques est ainsi intimement lié à la quantité de mémoire physique disponible. Par conséquent, plus il y a de mémoire, mieux c'est. Cela permet de donner un cache disque plus important à PostgreSQL, tout en laissant de la place en mémoire aux sessions pour traiter les données (faire des calculs de hachage par exemple).

Il est à noter que, même avec l'apparition des disques SSD, l'accès à une donnée en mémoire est bien plus rapide qu'une donnée sur disque. Nous aborderons ce point dans le chapitre consacré aux disques.

2.3 Disques

- Trois grandes technologies: SATA, SAS et SSD

Technologie	Temps d'accès	Débit en lecture
RAM	~ 1ns	~ 5Go/s
SSD	~ 0.1ms	~ 200Mo/s
SCSI 15ktpm	~ 1ms	~ 100Mo/s
SATA	~ 5ms	~ 100Mo/s

Il existe actuellement trois types de modèles de disques :

- SATA, dont la principale qualité est d'être peu cher ;
- SAS, rapide, fiable, mais cher ;
- SSD, très rapide en temps d'accès, très cher.

Les temps d'accès sont très importants pour un SGBD. Effectivement, ces derniers conditionnent les performances des accès aléatoires, utilisés lors des parcours d'index. Le débit en lecture, lui, influe sur la rapidité de parcours des tables de façon séquentielle (bloc par bloc, de proche en proche).

Il est immédiatement visible que la mémoire est toujours imbattable, y compris face aux disques SSD avec un facteur 100,000 en performance de temps d'accès entre les deux ! À l'autre bout de l'échelle se trouvent les disques SATA. Leur faible performance en temps d'accès ne doit pas pour autant les disqualifier. Leur prix est là-aussi imbattable et il est souvent préférable de prendre un grand nombre de disques pour avoir de bonnes performances. Cependant, la fiabilité des disques SATA impose de les considérer comme des consommable et de toujours avoir des disques de secours prêt à remplacer une défaillance.

Il est souvent préconisé de se tourner vers des disques SAS (SCSI). Leurs temps d'accès et leur fiabilité ont fait de cette technologie un choix de prédilection dans le domaine des SGBD. Mais si le budget ne le permet pas, des disques SATA en plus grand nombre permet d'en gommer les défauts.

Dans tous les cas, le nombre de disques est un critère important car il permet de créer des groupes RAID efficaces ou de placer les fichiers de PostgreSQL suivant leur utilisation. Par exemple les journaux de transactions sur un système disque, les tables sur un autre et les index sur un dernier.

Le gros intérêt du disque SSD est d'avoir un temps d'accès très rapide. Il se démarque des disques magnétiques (comme SAS ou SATA) par une durée d'accès à une page aléatoire aussi rapide que celle à une donnée contiguë (ou séquentielle). C'est parfait pour accéder à des index. Ils avaient une durée de vie plutôt limitée par rapport aux disques magnétiques. De nos jours, ce n'est plus tellement le cas grâce à des algorithmes d'écriture complexes permettant d'atteindre des durées de vie équivalentes, voir plus importantes, que celles des disques magnétiques. Néanmoins, ces mêmes algorithmes mettent en péril la durabilité des données en cas d'interruption brutale. Il est ainsi déconseillé d'utiliser cette technologie sans avoir au préalable effectué des tests de fiabilité intensifs. Les disques SSD les plus onéreux (souvent au détriment de leurs performances pures) réussiront ces tests. Les autres peuvent servir à stocker des données volatiles, comme les fichiers temporaires pour le tri et le hachage, ainsi que les tables et index temporaires. Il est possible de configurer le noyau pour optimiser l'utilisation des SSD :

```
echo noop > /sys/block/<device>/queue/scheduler
echo 0 > /sys/block/<device>/queue/rotational
```

Plus d'informations sont disponibles sur les disques SSD et Linux sur http://lwn.net/Articles/408428/.

2.4 RAID

- Différents niveaux de RAID
- Les plus intéressants pour un SGBD
  - RAID 1 (système, journaux de transactions)
  - RAID 10 (fichiers de données)
- Attention au cache
  - Toujours activer le cache en lecture
  - Activer le cache en écriture que si batterie présente

Il existe différents niveaux de RAID. Le plus connu est le RAID 5. C'est aussi le plus décrié pour un SGBD pour ses performances car il est très lent en écriture. Il est nettement préférable de se baser sur du RAID 10. Effectivement, ce dernier est tout aussi intéressant en terme de fiabilité, mais a de bien meilleures performances en lecture et écriture. En contrepartie, à volumétrie égale, il nécessite plus de disques et est donc beaucoup plus cher. Ainsi, il peut être préférable de choisir des disques SATA et mettre en œuvre un RAID 10 avec un budget moyen. Il est à noter que le système et les journaux de transactions n'ont pas besoin de RAID 10. Leur utilisation peut se satisfaire d'un simple RAID 1.

Les cartes RAID ne sont pas toutes aussi performantes et fiables. Les cartes intégrées aux cartes mères sont généralement de très mauvaise qualité. Il ne faut **jamais** transiger sur la qualité de la carte RAID.

La majorité des cartes RAID offre maintenant un système de cache de données en mémoire. Ce cache peut être simplement en lecture ou en lecture/écriture. Dans ce deuxième cas, ce cache étant volatile, la carte RAID **doit** posséder une batterie pour que les données en cache ne disparaissent pas en cas de coupure de courant. Ceci est obligatoire pour des raisons de fiabilité du service. Les meilleures cartes RAID permettent de superviser l'état de la batterie et désactivent le cache en écriture par mesure de sécurité si la batterie est vide ou morte.

2.5 SAN

- Pouvoir sélectionner les disques dans un groupe RAID
- Attention au cache
  - Toujours activer le cache en lecture
  - Activer le cache en écriture que si batterie présente
- Attention à la latence réseau
- Attention au système de fichiers: pas de NFS

Les SAN sont très appréciés en entreprise. Ils permettent de fournir le stockage pour plusieurs machines de manière fiable. Bien configurés, ils permettent d'atteindre de bonnes performances. Il est cependant important de comprendre les problèmes qu'ils peuvent poser.

Certains SAN ne permettent pas de sélectionner les disques placés dans un volume logique. Ils peuvent placer différentes partitions du même disque dans plusieurs volumes logiques. C'est un problème quand il devient impossible de dire si deux volumes logiques utilisent les mêmes disques. En effet, PostgreSQL permet de répartir des objets (tables ou index) sur plusieurs tablespaces différents. Cela n'a un intérêt en terme de performances que s'il s'agit de disques physiquement différents.

De même, certaines grappes de disques (eg. *RAID GROUP*) accueillent trop de volumes logiques pour de multiple serveurs (virtualisés ou non). Les performances des différents volumes dépendent alors directement de l'activité des autres serveurs connectés aux mêmes grappes.

Les SAN utilisent des systèmes de cache. L'avertissement concernant les cartes RAID et leur batterie vaut aussi pour les SAN qui proposent un cache en écriture.

Les SAN ne sont pas attachés directement au serveur. L'accès aux données accusera donc en plus une pénalité due à la latence réseau. Il est ainsi important de bien considérer son architecture réseau. Les équipements choisis doivent avoir une latence la plus faible possible, un débit important et les chemins entre les serveurs et la baie SAN multipliés.

Ces différentes considérations et problématiques (et beaucoup d'autres) font de la gestion de baies SAN un métier à part entière. Il **faut** y consacrer du temps de mise en œuvre, de configuration et de supervision important. En contrepartie de cette complexité et de leurs coûts, les SAN apportent beaucoup en fonctionnalités (snapshot, réplication, virtualisation. ..), en performances et en souplesse.

De plus, les disques étant distants et la technologie onéreuse, la tentation est grande d'utiliser un NAS, avec par exemple un accès NFS aux partitions. Il ne

faut pas, pour des raisons de performance et de fiabilité. Utilisez plutôt iSCSI, toujours peu performant, mais plus fiable et moins complexe.

2.6 Virtualisation 1/2

- Non recommandé
- Masque les ressources physiques au système
  - Investigation pour l'optimisation des performances plus difficiles
- Propose généralement des fonctionnalités d'overcommit
  - grandes difficultés à trouver la cause du problème du point de vue de la VM
  - dédier un minimum de ressources aux VM PostgreSQL
- En pause tant que l'hyperviseur ne dispose pas de l'ensemble des vCPU alloués à la machine virtuelle
- Mutualise les disques = problèmes de performances
  - Configurer les disques de PostgreSQL en « Thick Provisionning »

L'utilisation de machines virtuelles n'est pas recommandée avec PostgreSQL. En effet, la couche de virtualisation cache totalement les ressources physiques au système, ce qui rend l'investigation et l'optimisation des performances beaucoup plus difficiles qu'avec des serveurs physiques dédiés. Il est néanmoins possible d'utiliser des machines virtuelles avec PostgreSQL. Leur configuration doit alors être orientée vers la stabilité des performances. Cette configuration est complexe et difficile à suivre dans le temps. Les différentes parties de la plate-forme (virtualisation, système et bases de données) sont généralement administrées par des équipes techniques différentes, ce qui rend le diagnostique et la résolution de problèmes de performances plus difficiles. Les outils de supervision de chacun sont séparés et les informations plus difficiles à corréler.

Les solutions de virtualisation proposent généralement des fonctionnalités d'overcommit : les ressources allouées ne sont pas réservées à la machine virtuelle, la somme des ressources de l'ensemble des machines virtuelles peut donc être supérieure aux capacités du matériel. Dans ce cas, les machines peuvent ne pas disposer des ressources qu'elles croient avoir en cas de forte charge. Cette fonctionnalité est bien plus dangereuse avec PostgreSQL car la configuration du serveur est basée sur la mémoire disponible sur la VM. Si PostgreSQL utilise de la mémoire alors qu'elle se trouve en swap sur l'hyperviseur, les performances seront médiocres, et l'administrateur de bases de données aura de grandes difficultés à trouver la cause du problème du point de vue de la VM. Par conséquent, il est fortement conseillé de dédier un minimum de ressources aux VM PostgreSQL, et de superviser constamment

l'overcommit du côté de l'hyperviseur pour éviter ce « trashing ».

Il est généralement conseillé d'utiliser au moins quatre cœurs physiques. En fonction de la complexité des requêtes, du volume de données, de la puissance du CPU, un cœur physique sert en moyenne de 1 à 20 requêtes simultanées. L'ordonnancement des cœurs par les hyperviseurs a pour conséquence qu'une machine virtuelle est en « pause » tant que l'hyperviseur ne dispose pas de l'ensemble des vCPU alloués à la machine virtuelle pour la faire tourner. Dans le cas d'une configuration contenant des machines avec très peu de vCPU et d'une autre avec un nombre de vCPU plus important, la VM avec beaucoup de vCPU risque de bénéficier de moins de cycles processeurs lors des périodes de forte charge. Ainsi, les petites VM sont plus faciles à ordonnancer que les grosses, et une perte de puissance due à l'ordonnancement est possible dans ce cas. Cet effet, appelé *Steal Time* dans différents outils système (top, sysstat, …), se mesure en temps processeur où la VM a un processus en attente d'exécution, mais où l'hyperviseur utilise ce temps processeur pour une autre tâche. C'est pourquoi il faut veiller à configurer les VM pour éviter ce phénomène, avec un nombre de vCPU inférieurs au nombre de cœurs physiques réel sur l'hyperviseur.

Le point le plus négatif de la virtualisation de serveurs de bases de données concerne la performance des disques. La mutualisation des disques pose généralement des problèmes de performances car les disques sont utilisés pour des profils d'I/O généralement différents. Le RAID 5 offre le meilleur rapport performance/coût sauf pour les bases de données qui effectuent de nombreux accès aléatoires. De ce fait, le RAID 10 est préconisé car il est plus performant sur les accès aléatoires en écriture pour un nombre de disques équivalent. Avec la virtualisation, peu de disques mais de grande capacité sont généralement prévus sur les hyperviseurs, or cela implique un coût supérieur pour l'utilisation de RAID 10 et des performances inférieures sur les SGDB qui tirent de meilleures performances des disques lorsqu'ils sont nombreux. Enfin, les solutions de virtualisation effectuent du « Thin Provisioning » sur les disques pour minimiser les pertes d'espace. Pour cela, les blocs sont alloués et initialisés à la demande, ce qui apporte une latence particulièrement perceptible au niveau de l'écriture des journaux de transaction (in fine, cela détermine le nombre maximum de commits en écriture par seconde possible). Il est donc recommandé de configurer les disques de PostgreSQL en « Thick Provisionning ».

De plus, dans le cas de disques virtualisés, bien veiller à ce que l'hyperviseur respecte les appels de synchronisation des caches disques (appel système sync).

De préférence, dans la mesure du possible, évitez de passer par la couche de virtualisation pour les disques et préférez des attachements SAN, plus sûr et performants.

2.7 Virtualisation 2/2

- Éviter l'effet dit de “time drifts” en utilisant la même source NTP sur les OS invité (VM) et l'hôte ESXi;
- Utiliser les adaptateurs réseau para-virtualisé de type VMXNET3.
- Utiliser l'adaptateur para-virtualisé PVSCSI pour les disques dédiés aux partitions PostgreSQL;
- En cas de présence d'une architecture matérielle de type NUMA :
  - dimensionner la mémoire de chaque VM pour qu'elle ne dépasse pas le volume de mémoire physique au sein d'un groupe NUMA;

Il est aussi recommandé d'utiliser la même source NTP sur les OS invité (VM) et l'hôte ESXi afin d'éviter l'effet dit de “time drifts”. Il faut être attentif à ce problème des tops d'horloge. Si une VM manque des tops d'horloges sous une forte charge ou autre raison, elle va percevoir le temps qui passe comme étant plus lent qu'il ne l'est réellement. Par exemple, un OS invité avec un top d'horloge à 1 ms attendra 1000 top d'horloge pour une simple seconde. Si 100 top d'horloge sont perdu, alors 1100 top d'horloge seront délivré avant que la VM ne considère qu'une seconde soit passée. C'est ce qu'on appelle le “time drift”.

Il est recommander d'utiliser le contrôleur vSCSI VMware Paravirtual (aka PVSCSI). Ce contrôleur est intégré à la virtualisation et a été conçu pour supporter de très hautes bandes passantes avec un coût minimal, c'est le driver le plus performant. De même pour le driver réseau il faut privilégier l'adaptateur réseau para-virtualisé de type VMXNET3 pour avoir les meilleurs performances.

Un aspect très important de la configuration de la mémoire des machines virtuelles est l’accès mémoire non uniforme (NUMA). Cet accès permet d’accélérer l’accès mémoire en partitionnant la mémoire physique de telle sorte que chaque coeur d’un socket dispose de sa propre mémoire. Par exemple, avec un système à 16 cœurs et 128 Go de RAM, chaque cœur ou nœud possède 8 Go de mémoire physique.

Si une VM est configurée pour utiliser 12 Go de RAM, le système doit utiliser la mémoire d’un autre nœud. Le franchissement de la limite NUMA peut réduire les performances virtuelles jusqu’à 8 %, une bonne pratique consiste à configurer une VM pour utiliser les ressources d’un seul nœud NUMA.

3 Système d'exploitation

- Quel système choisir ?
- Quelle configuration réaliser ?

Le choix du système d'exploitation n'est pas anodin; les développeurs de PostgreSQL ont fait le choix de bien segmenter les rôles entre le système et le SGBD. Ainsi, PostgreSQL requiert que le système travaille de concert avec lui dans la gestion des accès disques, l'ordonnancement, etc.

PostgreSQL est principalement développé sur et pour Linux. Il fonctionne aussi sur d'autres systèmes, mais n'aura pas forcément les mêmes performances. De plus, la configuration du système et sa fiabilité jouent un grand rôle dans les performances et la robustesse de l'ensemble. Il est donc nécessaire de bien maîtriser ces points-là pour avancer dans l'optimisation.

3.1 Choix du système d'exploitation

- PostgreSQL fonctionne sur différents systèmes:
  - Linux, BSD, Windows, Solaris, HPUX, etc.
- Principalement développé et testé sous Linux
- Windows intéressant pour les postes des développeurs
  - Mais moins performant que Linux

PostgreSQL est écrit pour être le plus portable possible. Un grand nombre de choix dans son architecture a été fait en fonction de cette portabilité. Il est donc disponible sur la majorité des systèmes : Linux, BSD, Windows, Solaris, HPUX, etc. Cette portabilité est vérifiée en permanence avec la ferme de construction (BuildFarm, http://buildfarm.postgresql.org/).

Cela étant dit, il est malgré tout principalement développé sous Linux et la majorité des utilisateurs travaillent aussi avec Linux. Ce système est probablement le plus ouvert de tous, permettant ainsi une meilleure compréhension de ses mécaniques internes et ainsi une meilleure interaction. Ainsi, Linux est certainement le système le plus fonctionnel et performant avec PostgreSQL. La distribution Linux a généralement peu d'importance en ce qui concerne les performances. Les deux distributions les plus fréquemment utilisées sont RedHat (et ses dérivés CentOS, Scientific Linux) et Debian.

Un autre système souvent utilisé est Windows. Ce dernier est très intéressant mais beaucoup moins performant avec PostgreSQL que Linux. Cela est principalement dû à sa gestion assez mauvaise de la mémoire partagée. Cela a pour conséquence qu'il est difficile d'avoir un cache disque important pour PostgreSQL sous Windows.

De plus, vous ne pouvez pas démarrer PostgreSQL en tant que service si vous avez besoin de plus de 125 connexions pour des problématiques d'espace mémoire attribuée à un processus non-interactif. Le seul moyen de contourner ce problème sera de le lancer en mode interactif, depuis la ligne de commande. La limite théorique est alors repoussée à 750 connexions. Cf. http://wiki.postgresql.org/wiki/Running_%26_Installing_PostgreSQL_On_Native_Windows#I_cannot_run_with_more_than_about_125_connections_at_once.2C_despite_having_capable_hardware

Enfin, il est à noter que Windows ne dispose d'une version 64 bits de PostgreSQL que depuis la version 9.0.

3.2 Choix du noyau

- Choisir la version la plus récente du noyau car
  - plus stable
  - plus compatible avec le matériel
  - plus de fonctionnalités
  - plus de performances
- Utiliser la version de la distribution Linux
  - ne pas le compiler soi-même

Il est préférable de ne pas fonctionner avec une très ancienne version du noyau Linux. Les dernières versions sont les plus stables, les plus performantes, les plus compatibles avec les derniers matériels. Ce sont aussi celles qui proposent le plus de fonctionnalités intéressantes, comme la gestion complète du système de fichiers ext4, les « control groups », etc.

Le mieux est d'utiliser la version proposée par votre distribution Linux et de mettre à jour le noyau quand cela s'avère possible.

Le compiler vous-même peut dans certains cas vous apporter un plus en terme de performances. Mais ce plus est difficilement quantifiable et est assorti d'un gros inconvénient : avoir à gérer soi-même les mises à jours, la recompilation en cas d'oubli d'un pilote, etc.

3.3 Configuration du noyau

- En plus du choix du noyau, certains paramètres nécessitent une configuration personnalisée
  - taille de la mémoire partagée
  - contrôle de la mémoire partagée
  - sur-allocation de la mémoire
  - comportement du swap
  - affinité entre les cœurs et les espaces mémoire

Le noyau, comme tout logiciel, est configurable. Certaines configurations sont particulièrement importantes pour PostgreSQL.

3.4 Configuration de la mémoire partagée

- SHMMAX : mémoire partagée pour un processus
- SHMALL : mémoire partagée pour tous les processus
- rien à faire par défaut sur RedHat, mais à revoir sur les autres
- changé en 9.3

Les systèmes UNIX protègent les accès mémoire pour qu'un processus ne puisse ni lire ni écrire sur l'espace mémoire d'un autre processus. Cette protection est essentielle. Cependant, il est parfois nécessaire que plusieurs processus partagent un même espace de mémoire. Cela peut servir comme stockage global aux différents processus (un espèce de cache disque spécifique à ces processus) ou comme moyen de communication interprocessus. Ce type de mémoire est généralement appelée mémoire partagée (ou « shared memory » en anglais). Les systèmes Unix, dont Linux, disposent de différentes implémentations.

PostgreSQL est un système multi-processus, il a besoin de mémoire partagée. Du fait qu'il doit être facilement portable, il utilise l'implémentation la plus courante, à savoir la mémoire partagée « System V ». Ce type de mémoire partagée se configure avec plusieurs paramètres dont nous n'abordons ici que les deux principaux :

Paramètre	Valeur par défaut	Signification
SHMMAX	32Mo	Mémoire partagée allouable pour un

		processus
SHMALL	8Go	Quantité de mémoire partagée allouable, tous processus confondus

Le paramètre `SHMALL` est principalement intéressant dans le cas où le serveur accueille plusieurs instances PostgreSQL exécutées en même temps.

Les valeurs indiquées sont les valeurs par défaut proposées par Linux. Ces deux paramètres sont exprimés dans des unités différentes: `kernel.shmmax` en octets et `kernel.shmall` en pages (dont la taille par défaut est 4096)

```
kernel.shmmax = 33554432
kernel.shmall = 2097152
```

Si la valeur de `kernel.shmmax` est inférieure à la taille de mémoire partagée nécessaire à l'instance, cette dernière ne pourra démarrer et affichera un message d'erreur de ce type:

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5492001, size=30687232, 03600).
HINT:  This error usually means that PostgreSQL's request for a shared
memory segment exceeded your kernel's SHMMAX parameter.  You can either
reduce the request size or reconfigure the kernel with larger SHMMAX.  To
reduce the request size (currently 30687232 bytes), reduce PostgreSQL's
shared memory usage, perhaps by reducing shared_buffers or max_connections.
If the request size is already small, it's possible that it is less than
your kernel's SHMMIN parameter, in which case raising the request size or
reconfiguring SHMMIN is called for.
The PostgreSQL documentation contains more information about shared memory
configuration.
```

Certaines distributions, orientées serveur (comme la RedHat), ont déjà fortement augmenté ces valeurs. Dans les autres cas, il faut modifier les paramètres `kernel.shmmax` et `kernel.shmall` du fichier `/etc/sysctl.conf`.

À partir de la version 9.3, seul un tout petit peu de mémoire est allouée en System V et le reste est alloué anonymement par mmap ce qui permet de s'affranchir de la limite imposée par les paramètres `kernel.shmmax` et `kernel.shmall`.

Pour aller plus loin sur le sujet: http://rhaas.blogspot.fr/2012/06/absurd-shared-memory-limits.html

3.5 Contrôle du cache disque système

- Gestion de l'écriture des « dirty pages »
- Paramètres
  - vm.dirty_ratio
  - vm.dirty_background_ratio
  - vm.dirty_bytes
  - vm.dirty_background_bytes

La gestion de l'écriture des « dirty pages » (pages modifiées en mémoire mais non synchronisées) du cache disque système s'effectue à travers les paramètres `vm.dirty_ratio`, `vm.dirty_background_ratio`, `vm.dirty_bytes` et `vm.dirty_background_bytes`.

`vm.dirty_ratio` exprime le pourcentage de pages mémoires modifiées à atteindre avant que les processus écrivent eux-mêmes les données du cache sur disque afin de les libérer. Ce comportement est à éviter. `vm.dirty_background_ratio` définit le pourcentage de pages mémoires modifiées forçant le noyau à commencer l'écriture les données du cache système en tâche de fond. Ce processus est beaucoup plus léger et à encourager. Ce dernier est alors seul à écrire alors que dans le premier cas, plusieurs processus tentent de vider le cache système en même temps. Ce comportement provoque alors un encombrement de la bande passante des disques dans les situations de forte charge en écriture, surtout lors des opérations provoquant des synchronisations de données modifiées en cache sur le disque, comme l'appel à fsync. Fsync est utilisé par PostgreSQL lors des `CHECKPOINT`, ce qui peut provoquer des latences supplémentaires à ces moments-là.

Pour réduire les conséquences de ce phénomène, il est conseillé d'abaisser `vm.dirty_ratio` à 10 et `vm.dirty_background_ratio` à 5. Ainsi, lors de forte charges en écriture, nous demandons au noyau de reporter plus régulièrement son cache disque sur l'espace de stockage mais pour une volumétrie plus faible. Ainsi, l'encombrement de la bande passante vers les disques sera moins longue si ceux-ci ne sont pas capables d'absorber ces écritures rapidement. Dans les situations où la quantité de mémoire physique est importante, ces paramètres peuvent même être encore abaissés à 2 et 1 respectivement. Ainsi, avec 32 Go de mémoire, ils représentent 640 Mo et 320 Mo de données à synchroniser, ce qui peut nécessiter plusieurs secondes d'écritures en fonction de la configuration disque utilisée.

Dans les cas plus extrêmes, 1% de la mémoire représentent des volumétries trop importantes (eg. 1,3 Go pour 128 Go de mémoire physique). Les paramètres `vm.dirty_bytes` et `vm.dirty_background_bytes` permettent alors

de contrôler ces mêmes comportements, mais en fonction d'une quantité de dirty pages exprimée en octet et non plus en pourcentage de la mémoire disponible. Notez que ces paramètres ne sont pas complémentaires entre eux. Le dernier paramètre ayant été positionné prend le pas sur le précédent.

Enfin, plus ces valeurs sont basses, plus les synchronisations sont fréquentes, plus la durée des opérations `VACUUM` et `REINDEX`, qui déclenchent beaucoup d'écritures sur disque, augmentera.

3.6 Configuration du OOM

- Supervision de la sur-allocation par le noyau
- Si cas critique, l'OOM fait un kill -9 du processus
- À désactiver pour un serveur dédié
  - vm.overcommit_memory
  - vm.overcommit_ratio

Certaines applications réservent souvent plus de mémoire que nécessaire. Plusieurs optimisations noyau permettent aussi d'économiser de l'espace mémoire. Ainsi, par défaut, le noyau Linux s'autorise à allouer aux processus plus de mémoire qu'il n'en dispose réellement, le risque de réellement utiliser cette mémoire étant faible. On appelle ce comportement l'*Overcommit Memory*. Si celui-ci peut être intéressant dans certains cas d'utilisation, il peut devenir dangereux dans le cadre d'un serveur PostgreSQL dédié.

Effectivement, si le noyau arrive réellement à court de mémoire, il décide alors de tuer certains processus en fonction de leur impact sur le système. Il est alors fort probable que ce soit un processus PostgreSQL qui soit tué. Dans ce cas, les transactions en court seront annulées, et une perte de données est parfois possible en fonction de la configuration de PostgreSQL. Une corruption est par contre plutôt exclue.

Il est possible de modifier ce comportement grâce aux paramètres `vm.overcommit_memory` et `vm.overcommit_ratio` du fichier `/etc/sysctl.conf`. En plaçant `vm.overcommit_memory` à 2, le noyau désactivera complètement l'*overcommit memory*. La taille maximum de mémoire utilisable par les applications se calcule alors grâce à la formule suivante:

```
(RAM * vm.overcommit_ratio / 100) + SWAP
```

Attention, la valeur par défaut du paramètre `vm.overcommit_ratio` est 50. Ainsi, sur un système avec 32 Go de mémoire et 2 Go de swap, nous obtenons

seulement 18 Go de mémoire allouable ! Ne pas oublier de modifier ce paramètre ; avec `vm.overcommit_ratio` positionné à 75, nous obtenons 26 Go de mémoire utilisable par les applications sur les 32 Go disponibles.
Avoir un tel paramétrage permet de garantir qu'il y aura toujours au moins 20% du total de la RAM disponible pour le cache disque, qui est très bénéfique à PostgreSQL.

3.7 Configuration du swap

- Taille de la swap
  - pas plus de 2 Go
- Contrôler son utilisation
  - paramètre sysctl `vm.swappiness`

Il convient de déterminer la taille du swap de façon judicieuse. En effet, le swap n'est plus que rarement utilisé sur un système moderne, et 2 Go suffisent amplement en temps normal. Avoir trop de swap a tendance à aggraver la situation dans un contexte où la mémoire devient rare : le système finit par s'effondrer à force de swapper et dé-swapper un nombre de processus trop élevé par rapport à ce qu'il est capable de gérer. Ne pas avoir de swap est un autre problème : cela ne permet pas de s'apercevoir d'une surconsommation de mémoire. Il convient donc de créer un espace de swap de 2 Go sur la machine.

Le paramètre `vm.swappiness` contrôle le comportement du noyau vis-à-vis de l'utilisation du swap. Plus ce pourcentage est élevé, plus le système a tendance à swapper facilement. Un système hébergeant une base de données ne doit swapper qu'en dernière extrémité. La valeur par défaut de 60 doit donc être abaissée à 10 pour éviter l'utilisation du swap dans la majorité des cas.

3.8 Configuration de l'affinité processeur / mémoire

- pour architecture NUMA (multi-sockets)
- chaque socket travaille plus efficacement avec une zone mémoire allouée
- peut pénaliser le cache disque système
  - `vm.zone_reclaim_mode` pour le contrôler

Attention, ne pas confondre multi-cœurs et multi-sockets ! Chaque processeur physique occupe un socket et peut contenir plusieurs cœurs. Le nombre de processeurs physiques peut être trouvé grâce au nombre d'identifiants dans le label `physical id` du fichier “/proc/cpuinfo”. Par exemple, sur un serveur bi-processeur :

```
root@serveur:~# grep "^physical id" /proc/cpuinfo | sort -u | wc -l
2
```

Plus simplement, si la commande `lscpu` est présente, cette information est représentée par le champ “CPU socket(s)” :

```
root@serveur:~# lscpu | grep 'CPU socket'
CPU socket(s):         2
```

Sur une architecture NUMA (*Non Uniform Memory Access*), il existe une notion de distance entre les sockets processeurs et les “zones” mémoires (bancs de mémoire). La zone mémoire la plus proche d'un socket est alors définie comme sa zone “locale”. Il est plus coûteux pour les cœurs d'un processeur d'accéder aux zones mémoires distantes, ce qui implique des temps d'accès plus importants, et des débits plus faibles.

Le noyau Linux détecte ce type d'architecture au démarrage. Si le coût d'accès à une zone distante est trop important, il décide d'optimiser le travail en mémoires depuis chaque socket, privilégiant plus ou moins fortement les allocations et accès dans la zone de mémoire locale. Le paramètre `vm.zone_reclaim_mode` est alors supérieur à 0. Les processus étant exécutés sur un cœur processeur donné, ces derniers héritent de cette affinité processeur/zone mémoire. Le processus préfère alors libérer de l'espace dans sa zone mémoire locale si nécessaire plutôt que d'utiliser un espace mémoire distant libre, sapant par la même le travail de cache.

Si ce type d'optimisation peut être utile dans certains cas, il ne l'est pas dans un contexte de serveur de base de données où tout y est fait pour que les accès aux fichiers de données soient réalisés en mémoire, au travers des caches disque PostgreSQL ou système. Or, comme expliqué, les mécanismes du cache disque système sont impactés par les optimisations de `vm.zone_reclaim_mode`. Cette optimisation peut alors aboutir à une sous-utilisation de la mémoire, pénalisant notamment le cache avec un ratio d'accès y étant moins important côté système. De plus, elles peuvent provoquer des variations aléatoires de performances en fonction du socket où un processus serveur est exécuté et des zones mémoires qu'il utilise.

Ainsi, sur des architectures multi-sockets, il est conseillé de désactiver ce paramètre en positionnant `vm.zone_reclaim_mode` à `0`.

Pour illustrer les conséquences de cela, un test avec pg_dump sur une architecture NUMA montre les performances suivantes :

- avec zone_reclaim_mode à 1, temps de dump: 20h, CPU utilisé par le COPY: 3 à 5%
- avec zone_reclaim_mode à 0, temps de dump: 2h, CPU utilisé par le COPY: 95 à 100%

Le problème a été diagnostiqué à l'aide de l'outil système `perf`. Ce dernier a permis de mettre en évidence que la fonction `find_busiest_group` représentait le gros de l'activité du serveur. Dans le noyau Linux, cette fonction est utilisée en environnement multi-processeurs pour équilibrer la charge entre les différents processeurs.

Pour plus de détails, voir:

- http://www.postgresql.org/message-id/500616CB.3070408@2ndQuadrant.com
- http://blog.jcole.us/2010/09/28/mysql-swap-insanity-and-the-numa-architecture/

3.9 Comment les configurer

- Outil sysctl
- Configuration /etc/sysctl.conf

Tous les paramètres expliqués ci-dessus sont à placer dans le fichier “/etc/sysctl.conf”. Ainsi, à chaque redémarrage du serveur, Linux va récupérer le paramétrage et l'appliquer.

Sur les systèmes Linux modernes, un répertoire “/etc/sysctl.d” existe où tout fichier ayant l'extension “.conf” est lu et pris en compte. Ces fichiers ont la même syntaxe que “/etc/sysctl.conf”. Il est ainsi préconisé d'y créer un ou plusieurs fichiers pour vos configuration spécifiques afin que ces dernières ne soient pas accidentellement écrasées lors d'une mise à jour système par exemple.

Il est possible d'appliquer vos modifications sans redémarrer tout le système grâce à la commande suivante :

```
$ sysctl -p
```

3.10 Choix du système de fichiers

- Windows:
  - NTFS
- Linux
  - ext2, ext3, ext4, reiserfs, jfs, xfs, btrfs
- Solaris
  - ZFS
- Utiliser celui préconisé par votre système d'exploitation/distribution
  - ... et oublier NFS

Quelque soit le système d'exploitation, les systèmes de fichiers ne manquent pas. Linux en est la preuve avec pas moins d'une dizaine de systèmes de fichiers. Le choix peut paraître compliqué mais il se révèle fort simple : il est préférable d'utiliser le système de fichiers préconisé par votre distribution Linux. Ce système est à la base de tous les tests des développeurs de la distribution : il a donc plus de chances d'avoir moins de bugs, tout en proposant plus de performances. En règle générale, cela voudra dire le système ext3 ou ext4 (suivant la date de sortie de votre distribution). Les systèmes reiserfs et jfs ne sont pratiquement plus développés et doivent dans tous les cas être évités. ext2 peut fournir un brin supplémentaire de performance car il n'est pas journalisé mais cela a un prix : si le système s'arrête brutalement, une opération de vérification appelée `fsck` devra être réalisée sur toutes les partitions ext2, ce qui augmente **fortement** la durée d'immobilisation du serveur. btrfs est encore au stade expérimental mais il est très prometteur. Enfin, XFS est un système très intéressant pour les performances. Il est de nouveau proposé par RedHat, ce qui laisse penser que ses défauts de jeunesse ont disparu.

Pour Windows, la question ne se pose pas. Le système VFAT n'est pas suffisamment stable pour qu'il puisse être utilisé avec PostgreSQL. De plus, il ne connaît pas le concept des liens symboliques, important lors de la création de tablespaces avec PostgreSQL. La seule solution disponible sous Windows est donc NTFS. L'installeur fourni par EnterpriseDB dispose d'une protection qui empêche l'installation d'une instance PostgreSQL sur une partition VFAT.

Quant à Solaris, ZFS est un système très intéressant grâce à son panel fonctionnel et son mécanisme de Copy On Write permettant de faire une copie des fichiers sans arrêter PostgreSQL (aka. *Snapshot*). C'est l'un des rares systèmes à le proposer (avec XFS, LVM et bientôt btrfs).

NFS peut sembler intéressant, vu ses fonctionnalités. Cependant, ce système de fichiers est source de nombreux problèmes avec PostgreSQL. La

documentation l'indique très clairement (http://www.postgresql.org/docs/9.3/static/creating-cluster.html) :

> Many installations create database clusters on network file systems. Sometimes this is done directly via NFS, or by using a Network Attached Storage (NAS) device that uses NFS internally. PostgreSQL does nothing special for NFS file systems, meaning it assumes NFS behaves exactly like locally-connected drives (DAS, Direct Attached Storage). If client and server NFS implementations have non-standard semantics, this can cause reliability problems (see http://www.time-travellers.org/shane/papers/NFS_considered_harmful.html). Specifically, delayed (asynchronous) writes to the NFS server can cause reliability problems; if possible, mount NFS file systems synchronously (without caching) to avoid this. Also, soft-mounting NFS is not recommended. (Storage Area Networks (SAN) use a low-level communication protocol rather than NFS.)

Si la base est petite et que l'intégrité des données n'est pas importante, on peut éventuellement utiliser NFS.

Par contre, NFS est donc totalement déconseillé dans les environnements critiques avec PostgreSQL. Greg Smith, contributeur très connu, spécialisé dans l'optimisation de PostgreSQL, parle plus longuement des soucis de NFS avec PostgreSQL sur http://www.postgresql.org/message-id/4D2285CF.3050304@2ndquadrant.com. En fait, il y a des dizaines d'exemples de gens ayant eu des problèmes avec NFS. Les problèmes de performance sont quasi-systématiques, et les problèmes de fiabilité fréquents, et compliqués à diagnostiquer (comme illustré par http://www.postgresql.org/message-id/4D40DDB7.1010000@credativ.com, où le problème venait du noyau Linux).

3.11 Configuration du système de fichiers

- Quelques options à connaître :
  - noatime, nodiratime
  - dir_index
  - data=writeback
  - nobarrier
- Permet de gagner un peu en performance

Quel que soit le système de fichiers choisi, il est possible de le configurer lors du montage, via le fichier `/etc/fstab`.

Certaines options sont intéressantes en termes de performances. Ainsi, `noatime` évite l'écriture de l'horodatage du dernier accès au fichier. `nodiratime`

fait de même au niveau du répertoire. Depuis plusieurs années maintenant, `nodiratime` est inclus dans `noatime`.

L'option `dir_index` permet de modifier la méthode de recherche des fichiers dans un répertoire en utilisant un index spécifique pour accélérer cette opération. L'outil `tune2fs` permet de s'assurer que cette fonctionnalité est activée ou non. Par exemple, pour une partition “/dev/sda1” :

```
# tune2fs -l /dev/sda1 | grep features
Filesystem features:      has_journal resize_inode **dir_index** filetype
needs_recovery sparse_super large_file
```

`dir_index` est activé par défaut sur ext3 et ext4. Il ne pourrait être absent que si le système de fichiers était originellement un système ext2, qui aurait été mal migré.

Pour l'activer, il faut utiliser l'outil `tune2fs`. Par exemple :

```
# tune2fs -O dir_index /dev/sda1
```

. Enfin, il reste à créer ces index à l'aide de la commande `e2fsck`, eg. :

```
# e2fsck -D /dev/sda1
```

Les options `data=writeback` et `nobarrier` sont souvent cités comme optimisation potentielle. Le mode `writeback` de journalisation des ext3 et ext4 est à **éviter**. Effectivement, dans certains cas rare, en cas d'interruption brutale certains fichiers peuvent conserver des blocs “fantômes” ayant été normalement supprimés juste avant le crash.

L'option `nobarrier` quant à elle peut-être utilisée mais avec précaution. Cette dernière peut apporter une différence significative en terme de performance, mais mets en péril vos données en cas de coupure soudaine où les caches disques, RAID ou baies sont alors perdus. Cette option ne peut être utilisé qu'à la seule condition que tout ces différents caches soient sécurisés par une batterie.

4 Serveur de bases de données

- Version
- Configuration
- Emplacement des fichiers

Après avoir vu le matériel et le système d'exploitation, il est temps de passer au serveur de bases de données. Lors d'une optimisation, il est important de vérifier trois points essentiels :

- la version de PostgreSQL ;
- sa configuration (uniquement le fichier `postgresql.conf`) ;
- et l'emplacement des fichiers (journaux de transactions, tables, index, stats).

4.1 Version

- Chaque nouvelle version majeure a des améliorations de performance
  - Mettre à jour est un bon moyen pour gagner en performances
- Ne pas compiler
  - Sauf pour les Intel Itanium

Il est généralement conseillé de passer à une version majeure plus récente qu'à partir du moment où les fonctionnalités proposées sont suffisamment intéressantes. C'est un bon conseil en soi mais il faut aussi se rappeler qu'un gros travail est fait pour améliorer le planificateur. Ces améliorations peuvent être une raison suffisante pour changer de version majeure.

Voici quelques exemples frappants :

- La version 9.0 dispose d'une optimisation du planificateur lui permettant de supprimer une jointure `LEFT JOIN` si elle est inutile pour l'obtention du résultat. C'est une optimisation particulièrement bienvenue pour tous les utilisateurs d'ORM.
- La version 9.1 dispose du SSI, pour `Serializable Snapshot Isolation`. Il s'agit d'une implémentation très performante du mode d'isolation sérialisée. Ce mode permet d'éviter l'utilisation des `SELECT FOR UPDATE`.

- La version 9.2 dispose d'un grand nombre d'améliorations du planificateur et des processus postgres qui en font une version exceptionnelle pour les performances, notamment les parcours d'index seuls.

Compiler soi-même PostgreSQL ne permet pas de gagner réellement en performance. Même s'il peut y avoir un gain, ce dernier ne peut être que mineur et difficilement identifiable. La compilation n'a un impact réellement identifié **que** sur les architecture Itanium (*IA-32* et *IA-64*) avec le compilateur propriétaire Intel (appelé `ICC`), cf. .

Dans certain cas, ce compilateur apporte de meilleures performances au niveau de PostgreSQL. On a observé jusqu'à 10% de gain par rapport à une compilation “classique” (GCC). Il faut toutefois prendre deux éléments importants en compte avant de remplacer les binaires de PostgreSQL par des binaires recompilés avec ICC :

1. La taille des fichiers recompilés est nettement plus grande;
2. La compilation avec ICC est moins documentée et moins testée qu'avec GCC.

Il est donc nécessaire de préparer avec soin, de documenter la procédure de compilation et de réaliser des tests approfondis avant de mettre une version recompilée de PostgreSQL dans un environnement de production.

Plus d'informations :

http://www.developpez.net/forums/d647904/bases-donnees/postgresql/faut-til-compiler-postgresql-differemment/

4.2 Configuration - mémoire

- shared_buffers
- wal_buffers
- work_mem
- maintenance_work_mem
- max_fsm_pages/max_fsm_relations

Ces six paramètres concernent tous la quantité de mémoire que PostgreSQL utilisera pour ses différentes opérations.

`shared_buffers` permet de configurer la taille du cache disque de PostgreSQL. Chaque fois qu'un utilisateur veut extraire des données d'une table (par une requête `SELECT`) ou modifier les données d'une table (par exemple avec une requête `UPDATE`), PostgreSQL doit d'abord lire les lignes impliquées et les mettre dans son cache disque. Cette lecture prend du temps. Si ces lignes sont déjà dans le cache, l'opération de lecture n'est plus utile, ce qui permet de

renvoyer plus rapidement les données à l'utilisateur. Ce cache est commun à tous les processus PostgreSQL, il n'existe donc qu'en un exemplaire. Généralement, il faut lui donner une grande taille, tout en conservant malgré tout la majorité de la mémoire pour le cache disque du système, à priori plus efficace pour de grosses quantités de données. Le pourcentage généralement préconisé est de 25% de la mémoire totale pour un serveur dédié. Donc, par exemple, pour un serveur contenant 8 Go de mémoire, nous configurerons le paramètre shared_buffers à 2 Go. Néanmoins, on veillera à ne pas dépasser 8 Go. Des études ont montré que les performances décroissaient avec plus de mémoire.

PostgreSQL dispose d'un autre cache disque. Ce dernier concerne les journaux de transactions. Il est généralement bien plus petit que `shared_buffers` mais, si le serveur est multi-processeurs et qu'il y a de nombreuses connexions simultanées au serveur PostgreSQL, il est important de l'augmenter. Le paramètre en question s'appelle `wal_buffers`. Plus cette mémoire est importante, plus les transactions seront conservées en mémoire avant le `COMMIT`. À partir du moment où le `COMMIT` d'une transaction arrive, toutes les modifications effectuées dans ce cache par cette transaction sont enregistrées dans le fichier du journal de transactions. La valeur par défaut est de 64 Ko mais une valeur de 16 Mo sera plus intéressante. Il est à noter qu'à partir de la version 9.1, cette taille est gérée automatiquement par PostgreSQL si `wal_buffers` vaut -1.

Deux autres paramètres mémoire sont essentiels pour de bonnes performances mais eux sont valables par processus. `work_mem` est utilisé comme mémoire de travail pour les tris et les hachages. S'il est nécessaire d'utiliser plus de mémoire, le contenu de cette mémoire est stocké sur disque pour permettre la réutilisation de la mémoire. Par exemple, si une jointure demande à stocker 52 Mo en mémoire alors que le paramètre `work_mem` vaut 10 Mo, à chaque utilisation de 10 Mo, cette partie de mémoire sera copiée sur disque, ce qui fait en gros 50 Mo écrit sur disque pour cette jointure. Si, par contre, le paramètre `work_mem` vaut 60 Mo, aucune écriture n'aura lieu sur disque, ce qui accélérera forcément l'opération de jointure. Cette mémoire est utilisée par chaque processus du serveur PostgreSQL, de manière indépendante. Suivant la complexité des requêtes, il est même possible qu'un processus utilise plusieurs fois cette mémoire (par exemple si une requête fait une jointure et un tri). Il faut faire très attention à la valeur à donner à ce paramètre et le mettre en relation avec le nombre maximum de connexions (paramètre max_connections). Si la valeur est trop petite, cela forcera des écritures sur le disque par PostgreSQL. Si elle est trop grande, cela pourrait faire swapper le serveur. Généralement, une valeur entre 10 et 50 Mo est concevable. Au delà de 100 Mo, il y a probablement un problème ailleurs : des tris sur de trop gros volumes de données, une mémoire insuffisante, un manque d'index (utilisé pour les tris), etc. Des valeurs vraiment grandes ne sont valables que sur des systèmes d'infocentre.

Quant à `maintenance_work_mem`, il est aussi utilisé par chaque processus PostgreSQL réalisant une opération particulière : un VACUUM, une création

d'index ou l'ajout d'une clé étrangère. Comme il est peu fréquent que ces opérations soient effectuées en simultané, la valeur de ce paramètre est très souvent bien supérieure à celle du paramètre work_mem. Sa valeur se situe fréquemment entre 128 Mo et 1 Go, voire plus.

Enfin, en ce qui concerne `max_fsm_pages` et `max_fsm_relations`, le premier indique le nombre de pages qu'une structure en mémoire, appelée FSM, est capable d'enregistrer et le deuxième son nombre de tables. Cette structure est remplie par l'instruction `VACUUM` pour conserver en mémoire les emplacements libres disponibles pour être réécrits. Si cette structure n'est pas assez grande (ce qui est souvent le cas), des emplacements libres sont oubliés par PostgreSQL, ce qui empêche la réutilisation des espaces libres. Cela augmente la fragmentation. À partir de la version 8.4, ces deux paramètres disparaissent, la structure étant maintenant sur disque (il n'est donc plus nécessaire de la dimensionner).

4.3 Configuration - planificateur

- effective_cache_size
- random_page_cost

Le planificateur dispose de plusieurs paramètres de configuration. Les deux principaux sont `effective_cache_size` et `random_page_cost`.

Le premier permet d'indiquer la taille du cache disque du système d'exploitation. Ce n'est donc pas une mémoire que PostgreSQL va allouer, c'est plutôt une simple indication de ce qui est disponible en dehors de la mémoire taillée par le paramètre `shared_buffers`. Le planificateur se base sur ce paramètre pour évaluer les chances de trouver des pages de données en mémoire. Une valeur plus importante aura tendance à faire en sorte que le planificateur privilégie l'utilisation des index, alors qu'une valeur plus petite aura l'effet inverse. Généralement, il se positionne à 2/3 de la mémoire d'un serveur pour un serveur dédié.

Une meilleure estimation est possible en parcourant les statistiques du système d'exploitation. Sur les systèmes Unix, ajoutez les nombres `buffers+cached` provenant des outils `top` ou `free`. Sur Windows, voir la partie « System Cache » dans l'onglet « Performance » du gestionnaire des tâches. Par exemple, sur un portable avec 2 Go de mémoire, il est possible d'avoir ceci :

```
$ free
             total       used       free     shared    buffers     cached
Mem:       2066152    1525916     540236          0     190580     598536
-/+ buffers/cache:     736800    1329352
Swap:      1951856          0    1951856
```

Soit 789166 Ko, résultat de l'addition de 190580 (colonne `buffers`) et 598536 (colonne `cached`).

Le paramètre `random_page_cost` permet de faire appréhender au planificateur le fait qu'une lecture aléatoire (autrement dit avec déplacement de la tête de lecture) est autrement plus coûteuse qu'une lecture séquentielle. Par défaut, la lecture aléatoire a un coût quatre fois plus important que la lecture séquentielle. Ce n'est qu'une estimation, cela n'a pas à voir directement avec la vitesse des disques. Ça le prend en compte, mais ça prend aussi en compte l'effet du cache. Cette estimation peut être revue. Si elle est revue à la baisse, les parcours aléatoires seront moins coûteux et, par conséquent, les parcours d'index seront plus facilement sélectionnés. Si elle est revue à la hausse, les parcours aléatoires coûteront encore plus cher, ce qui risque d'annuler toute possibilité d'utiliser un index. La valeur 4 est une estimation basique. En cas d'utilisation de disque rapide, il ne faut pas hésiter à descendre un peu cette valeur (entre 2 et 3 par exemple). Si les données tiennent entièrement en cache où sont stockées sur des disques SSD, il est même possible de descendre encore plus cette valeur.

4.4 Configuration - WAL

- fsync
- checkpoint_segments
- checkpoint_timeout
- checkpoint_completion_target

`fsync` est le paramètre qui assure que les données sont non seulement écrites mais aussi forcées sur disque. En fait, quand PostgreSQL écrit dans des fichiers, cela passe par des appels systèmes pour le noyau qui, pour des raisons de performances, conserve dans un premier temps les données dans un cache. En cas de coupure courant, si ce cache n'est pas vidé sur disque, il est possible que des données enregistrées par un COMMIT implicite ou explicite n'aient pas atteint le disque et soient donc perdues une fois le serveur redémarré, ou pire, que des données aient été modifiées dans des fichiers de données, sans avoir été auparavant écrites dans le journal, ce qui entraînera dans ce cas des incohérences dans les fichiers de données au redémarrage. Il est donc essentiel que les données enregistrées dans les journaux de transactions soient non seulement écrites mais que le noyau soit forcé de les écrire réellement sur disque. Cette opération s'appelle `fsync`. Par défaut, ce comportement est activé. Évidemment, cela coûte en performance mais ce que ça apporte en terme de fiabilité est essentiel. Il est donc obligatoire en production de conserver ce paramètre activé.

Chaque bloc modifié dans le cache disque de PostgreSQL doit être écrit sur

disque au bout d'un certain temps. Ce temps dépend de deux paramètres `checkpoint_segments` et `checkpoint_timeout`. Le deuxième permet de s'assurer d'avoir au minimum un `CHECKPOINT` toutes les X minutes (5 par défaut). Tout surplus d'activité doit aussi être géré. Un surplus d'activité engendrera des journaux de transactions supplémentaires. Le meilleur moyen dans ce cas est de préciser au bout de combien de journaux traités il faut lancer un `CHECKPOINT`. Cela se fait via le paramètre `checkpoint_segments`.

4.5 Configuration - statistiques

- track_activities
- track_count

Ces deux paramètres ne permettent pas de gagner en performances. En fait, ils vont même faire un peu perdre car ils ajoutent une activité supplémentaire de récupération de statistiques sur l'activité des processus de PostgreSQL. `track_counts` permet de compter, par exemple, le nombre de transactions validées et annulées, le nombre de blocs lus dans le cache de PostgreSQL et en dehors, le nombre de parcours séquentiels (par table) et d'index (par index). La charge supplémentaire n'est généralement pas importante mais elle est là. Cependant, les informations que cela procure sont essentielles pour travailler sur les performances et pour avoir un système de supervision (là-aussi, la base pour de l'optimisation ultérieure).

Désactiver ces paramètres peut vous faire un peu gagner en performance mais les informations qui vous perdrez vous empêcheront d'aller très loin en matière d'optimisation.

4.6 Configuration - autovacuum

- autovacuum

L'autovacuum doit être activé. Ce processus supplémentaire coûte un peu en performances mais il s’acquitte de deux tâches importantes pour les performances : éviter la fragmentation dans les tables et index, et mettre à jour les statistiques sur les données.

4.7 pgtune

- Outil écrit en Python, par Greg Smith
- Propose quelques meilleures valeurs pour certains paramètres
- Quelques options pour indiquer des informations systèmes

Le site du projet se trouve sur http://pgfoundry.org/projects/pgtune/, mais il est préférable de récupérer le code à partir du dépôt GIT (http://git.postgresql.org/gitweb/?p=pgtune.git).

pgtune est capable de trouver la quantité de mémoire disponible sur le système. À partir de cette information et de quelques règles internes, il arrive à déduire une configuration bien meilleure que la configuration par défaut. Il est important de lui indiquer le type d'utilisation principale : Web, DW (pour DataWarehouse), mixed, etc.

Sur le serveur de tests se trouvent 8 Go de RAM. Commençons par une configuration pour une utilisation par une application web :

```
server:~$ ./pgtune -i /opt/postgresql-9.1/data/postgresql.conf -T Web
[... contenu de l'ancien fichier ...]
#------------------------------------------------------------------------
# pgtune wizard run on 2012-07-25
# Based on 8102944 KB RAM in the server
#------------------------------------------------------------------------

default_statistics_target = 100
maintenance_work_mem = 480MB
checkpoint_completion_target = 0.7
effective_cache_size = 5632MB
work_mem = 36MB
wal_buffers = 4MB
checkpoint_segments = 8
shared_buffers = 1920MB
max_connections = 200
```

Une application web, c'est beaucoup d'utilisateurs qui exécutent de petites requêtes simples, très rapides, non consommatrices. Du coup, le nombre de connexions a été doublé par rapport à sa valeur par défaut. Le paramètre `work_mem` est augmenté mais raisonnablement par rapport à la mémoire totale et au nombre de connexions. Le paramètre `shared_buffers` se trouve à un peu moins que le quart de la mémoire, alors que le paramètre `effective_cache_size` est au deux-tiers évoqué précédemment. Le paramètre `wal_buffers` est aussi augmenté mais peu. Il arrive à 4 Mo. Il peut y avoir beaucoup de transactions en même temps, mais elles seront généralement peu coûteuses en écriture. D'où le fait que les paramètres `checkpoint_segments` et `checkpoint_completion_target` sont augmentés mais là-aussi très

raisonnablement.

Voyons maintenant avec un profil OLTP (Online transaction processing) :

```
server:~$ ./pgtune -i /opt/postgresql-9.1/data/postgresql.conf -T OLTP
[... contenu de l'ancien fichier ...]
#------------------------------------------------------------------------
# pgtune wizard run on 2012-07-25
# Based on 8102944 KB RAM in the server
#------------------------------------------------------------------------

default_statistics_target = 100
maintenance_work_mem = 480MB
checkpoint_completion_target = 0.9
effective_cache_size = 5632MB
work_mem = 26MB
wal_buffers = 8MB
checkpoint_segments = 16
shared_buffers = 1920MB
max_connections = 300
```

Une application OLTP doit gérer un plus grand nombre d'utilisateurs. Ils font autant d'opérations de lecture que d'écriture. Tout cela est transcrit dans la configuration. Un grand nombre d'utilisateurs simultanés veut dire une valeur importante pour le paramètre `max_connections` (maintenant à 300). De ce fait, le paramètre `work_mem` ne peut plus avoir une valeur si importante. Sa valeur est donc baissée tout en restant fortement au-dessus de la valeur par défaut. Dû au fait qu'il y aura plus d'écritures, la taille du cache des journaux de transactions (paramètre `wal_buffers`) est augmentée. Il faudra essayer de tout faire passer par les `CHECKPOINT`, d'où la valeur maximale pour `checkpoint_completion_target`. Quant à `shared_buffers` et `effective_cache_size`, ils restent aux valeurs définies ci-dessus (respectivement un quart et deux-tiers de la mémoire).

Et enfin avec un profil entrepôt de données (Data Wharehouse) :

```
server:~$ ./pgtune -i /opt/postgresql-9.1/data/postgresql.conf -T DW
[... contenu de l'ancien fichier ...]
#------------------------------------------------------------------------
# pgtune wizard run on 2012-07-25
# Based on 8102944 KB RAM in the server
#------------------------------------------------------------------------
default_statistics_target = 500
maintenance_work_mem = 960MB
checkpoint_completion_target = 0.9
effective_cache_size = 5632MB
work_mem = 192MB
wal_buffers = 32MB
checkpoint_segments = 64
shared_buffers = 1920MB
max_connections = 20
```

Pour un entrepôt de données, il y a généralement peu d'utilisateurs à un instant t. Par contre, ils exécutent des requêtes complexes sur une grosse volumétrie. Du coup, la configuration change en profondeur cette fois. Le paramètre `max_connections` est diminué très fortement. Cela permet d'allouer beaucoup de mémoire aux tris et hachages (paramètre `work_mem` à 32 Mo). Les entrepôts de données ont souvent des scripts d'import de données (batchs). Cela nécessite de pouvoir écrire rapidement de grosses quantités de données, autrement dit une augmentation conséquente du paramètre `wal_buffers` et du `checkpoint_segments`. Du fait de la grosse volumétrie des bases dans ce contexte, une valeur importante pour le `maintenance_work_mem` est essentiel pour que les créations d'index et les `VACUUM` se fassent rapidement. De même, la valeur du `default_statistics_target` est sérieusement augmentée car le nombre de lignes des tables est conséquent et nécessite un échantillon plus important pour avoir des statistiques précises sur les données des tables.

Évidemment, tout ceci n'est qu'une recommandation générale. L'expérimentation permettra de se diriger vers une configuration plus personnalisée.

4.8 Outil pgbench

- outil pour réaliser rapidement des tests de performances
- fourni dans les modules de “contrib” de PostgreSQL
- travail sur une base de test créée par l'outil…
- …ou sur une vraie base de donnée
- on peut faire varier différents paramètres, tel que:
  - le nombre de clients
  - le nombre de transactions par client
  - faire un test de performance en `SELECT only`, `UPDATE only` ou `TPC-B`
  - faire un test de performance dans son contexte applicatif
  - exécuter le plus de requêtes possible sur une période de temps donné
  - etc.

4.9 Environnement de test avec pgbench

- pgbench est capable de créer son propre environnement de test
- adapté pour des tests de type TPC-B
- permet de rapidement tester une configuration PostgreSQL
  - en terme de performance
  - en terme de charge
- ou expérimenter

L'option `-i` demande à pgbench de créer un schéma et le peupler de donnée dans la base indiquée (à créer au préalable). La base ainsi créée est composée de 4 tables : `pgbench_history`, `pgbench_tellers`, `pgbench_accounts` et `pgbench_branches`. Dans ce mode, l'option `-s` permet alors de d'indiquer un facteur d'échelle permettant de maîtriser la volumétrie de la base de donnée. Ce facteur est un multiple de 100,000 lignes dans la table `pgbench_accounts`.

Une fois créée, il est possible de réaliser différents tests avec cette base de donnée en faisant varier plusieurs paramètres tel que le nombre de de transaction, le nombre de clients, le type de requêtes (simple, étendue, préparée) ou la durée du test de charge.

Quelques exemples. Le plus simple :

```
$ createdb benchs

$ pgbench -i -s 2 benchs
NOTICE:  table "pgbench_branches" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_history" does not exist, skipping
creating tables...
10000 tuples done.
20000 tuples done.
30000 tuples done.
40000 tuples done.
50000 tuples done.
60000 tuples done.
70000 tuples done.
80000 tuples done.
90000 tuples done.
100000 tuples done.
110000 tuples done.
120000 tuples done.
130000 tuples done.
140000 tuples done.
150000 tuples done.
160000 tuples done.
170000 tuples done.
```

```
180000 tuples done.
190000 tuples done.
200000 tuples done.
set primary key...
NOTICE:  ALTER TABLE / ADD PRIMARY KEY will create implicit index
"pgbench_branches_pkey" for table "pgbench_branches"
NOTICE:  ALTER TABLE / ADD PRIMARY KEY will create implicit index
"pgbench_tellers_pkey" for table "pgbench_tellers"
NOTICE:  ALTER TABLE / ADD PRIMARY KEY will create implicit index
"pgbench_accounts_pkey" for table "pgbench_accounts"
vacuum...done.



$ pgbench benchs
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 2
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
tps = 107.390623 (including connections establishing)
tps = 114.761812 (excluding connections establishing)
```

À partir de la base bench initialisée ci-dessus, on réalise un autre test avec 10 clients et 200 transactions pour chacun :

```
$ pgbench -c 10 -t 200 benchs
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 2
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 200
number of transactions actually processed: 2000/2000
tps = 100.995676 (including connections establishing)
tps = 101.152552 (excluding connections establishing)
```

Après avoir spécifié `fsync=off`, on fait un autre test sur la base avec les mêmes paramètres que ci-dessus :

```
$ pgbench -c 10 -t 200 benchs
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 2
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 200
number of transactions actually processed: 2000/2000
tps = 2436.463133 (including connections establishing)
```

```
tps = 2548.069328 (excluding connections establishing)
```

Toujours avec les mêmes paramètres, mais en effectuant le test durant 10 secondes:

```
$ pgbench -c 10 -T 10 benchs
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 2
query mode: simple
number of clients: 10
number of threads: 1
duration: 10 s
number of transactions actually processed: 27409
tps = 2738.335822 (including connections establishing)
tps = 2745.770998 (excluding connections establishing)
```

4.10 Environnement réel avec pgbench

- pgbench est capable de travailler avec une base existante
- lit les requêtes depuis un ou plusieurs fichiers
- utilisation possible de variables et commandes

L'outil pgbench est capable de travailler avec une base de donnée existante. Cette fonctionnalité permet ainsi de tester les performances dans un contexte plus représentatif de la ou les bases présentent dans une instance.

Pour effectuer de tels tests, il faut créer un ou plusieurs script SQL contenant les requêtes à exécuter sur la base de donnée. Chaque requête doit être écrite sur **UNE** seule ligne, un script peut contenir plusieurs requêtes. Toutes les requêtes du fichier seront exécutées dans leur ordre d'apparition. Si plusieurs scripts SQL sont indiqués, chaque transaction sélectionne le fichier à exécuter de façon aléatoire. Enfin, il est possible d'utiliser des variables dans vos scripts SQL afin de faire varier le set de donnée manipulé dans vos tests. Ce dernier point est essentiel afin d'éviter les effets de cache ou encore pour simuler la charge lorsque qu'un sous ensemble des données de la base est utilisé en comparaison avec la totalité de la base (en utilisant un champs de date par exemple).

Par exemple, le script exécuté par défaut par pgbench pour son test TPC-B en mode requête « simple », sur sa propre base, est le suivant (extrait de la page de manuel de pgbench) :

```
\SET nbranches :scale
\SET ntellers 10 * :scale
```

```
\SET naccounts 100000 * :scale
\setrandom aid 1 :naccounts
\setrandom bid 1 :nbranches
\setrandom tid 1 :ntellers
\setrandom delta -5000 5000
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid,
:bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

Ici, la variable `:scale` a comme valeur celle indiquée lors de l'initialisation de la base de donnée.

4.11 Emplacement des journaux de transactions

- Placer les journaux sur un autre disque
- Option -X de l'outil initdb
- Lien symbolique

Chaque donnée modifiée est écrite une première fois dans les journaux de transactions et une deuxième fois dans les fichiers de données. Cependant, les écritures dans ces deux types de fichiers sont très différentes. Les opérations dans les journaux de transactions sont uniquement des écritures séquentielles, sur de petits fichiers (d'une taille de 16 Mo), alors que celles des fichiers de données sont des lectures et des écritures fortement aléatoires, sur des fichiers bien plus gros (au maximum 1 Go). Du fait d'une utilisation très différente, avoir un système disque pour l'un et un système disque pour l'autre permet de gagner énormément en performances. Il faut donc pouvoir les séparer.

Avant la version 8.3, il est nécessaire d'arrêter PostgreSQL, de déplacer le répertoire des journaux de transactions, de créer un lien vers ce répertoire, et enfin de redémarrer PostgreSQL. Voici un exemple qui montre le déplacement dans /pgxlog.

```
$ /etc/init.d/postgresql stop
$ cd $PGDATA
$ mv pg_xlog /pgxlog
$ ln -s /pgxlog pg_xlog
$ /etc/init.d/postgresql start
```

À partir de la version 8.3, il est aussi possible de faire en sorte que la

commande initdb le fasse elle-même. Pour cela, il faut utiliser l'option -X :

```
$ initdb -X /pgxlog
```

Cependant le résultat est le même. Un lien symbolique existe dans le répertoire de données pour que PostgreSQL retrouve le répertoire des journaux de transactions.

4.12 Emplacement des fichiers de données

- Séparer les objets suivant leur utilisation
- Tablespaces
- Quelques stratégies
  - séparer tables et index
  - séparer archives et données vivantes
- Configuration possible des tablespaces
  - `seq_page_cost`, `random_page_cost`

Dans le même ordre d'idées, il est aussi possible de séparer les objets SQL dans des disques différents. Par défaut, PostgreSQL se charge du placement des objets sur le disque. Tout a lieu dans le répertoire des données, mais il est possible de créer des répertoires de stockage supplémentaires. Le nom de ces répertoires, au niveau SQL, est tablespace. Pour placer un objet dans un tablespace, il faut créer ce tablespace si ce n'est pas déjà fait, puis lancer l'ordre SQL de déplacement d'objet. Voici un exemple complet :

```
$ mkdir /opt/tablespace1
$ chown postgres:postgres /opt/tablespace1
$ chmod 700 /opt/tablespace1
$ psql postgres
postgres =# CREATE TABLESPACE grosdisque LOCATION '/opt/tablespace1';
postgres =# ALTER TABLE t1 TABLESPACE grosdisque;
```

L'idée est de séparer les objets suivant leur utilisation. Une configuration assez souvent utilisée est de placer les tables dans un tablespace et les index dans un autre. Cela permet des écritures quasi-simultanées sur différents fichiers.

La seule configuration possible au niveau des tablespaces se situe au niveaux des paramètres `seq_page_cost` et `random_page_cost`. Ils sont utilisés par le planificateur pour évaluer la vitesse de récupérer une page séquentielle et une page aléatoire. C'est essentiellement intéressant avec les SSD qui ont normalement une vitesse sensiblement équivalente pour les accès séquentiels et aléatoires, contrairement aux disques magnétiques.

```
ALTER TABLESPACE disque_ssd SET ( random_page_cost = 1 );
```

5 Conclusion

PostgreSQL propose de nombreuses voies d'optimisation.

Cela passe en priorité par un bon choix des composants matériels et par une configuration pointilleuse.

5.1 Questions

N'hésitez pas, c'est le moment !