DALIBO

L'Expertise PostgreSQL

PostgreSQL : Administration système

Formateur : Jean-Paul Argudo
Contact : jean-paul.argudo@dalibo.com
Date : mars 2010

Table des matières

Introduction

Ce module est organisé en sept parties :

- Profils matériels
- Gestion des ressources du noyau
- Systèmes de fichiers
- Manipulations du serveur
- Le cluster PostgreSQL
- Les processus d'un serveur PostgreSQL
- Migration

Licence Creative Commmons CC-BY-NC-SA

Partie 1 : Profils matériels

- CPU
- Types de disques
- Gestion des disques
- Technologie RAID
- 5 principes

Cette première partie présente les principes de base que le DBA devra avoir en tête lors de la définition de ses serveurs de données PostgreSQL puis de leur initialisation.

CPU

- Le plus possible
  Il est toujours intéressant de disposer d'une grande puissance de calcul... Cependant PostgreSQL ne dispose pas encore de l'exécution de calculs en parallèle.
  En revanche, étant donné que chaque utilisateur connecté bénéficie de son propre *backend spawné* par le processus `postmaster`, sur un système multi-processeur, ceux-ci vont se répartir naturellement. Cela est d'autant plus important depuis l'avénement des processeurs multicoeurs.
- Mais la puissance processeur n'est pas forcément le plus important
  Si la base de données ne contient pas de fonctions et de procédures stockées très complexes, alors il est plus judicieux de privilégier la mémoire et les disques.
- Opteron / Xeon
  En termes de choix processeurs, après une forte domination d'AMD et de ses opterons durant la période 2003-2008 en termes de performances pour des serveurs de bases de données, les deux fabricants fournissent aujourd'hui des offres comparables. Cette période correspond à la période 'Pentium 4' d'Intel, lancé dans une course aux GigaHertz, de laquelle il est revenu depuis avec les architectures Core, Nehalem.
  On ne parle ici bien sûr que des architectures dérivées du jeu d'instruction x86 d'Intel, mais PostgreSQL est aussi compatible avec les architectures SPARC et POWER par exemple.

RAM

- Plus il y aura de RAM
- Plus le cache disque sera grand
- Plus les performances seront élévées
  Les temps d'accès d'une donnée en mémoire sont environ un million de fois plus petits que les entrées/sorties disque. La taille du cache a donc un impact décisif sur les performances de l'ensemble du système.

Type de disques

- Meilleur choix : `SAS`
  Évidemment, l'idéal est de disposer des disques les plus rapides possible. Actuellement, les disques durs les plus performants sont les disques de type `SCSI`.
  Il y a cependant une différence de prix *notable* entre des disques dont la vitesse de rotation est de 10 000 tours par minute et ceux qui en affichent 15 000 ! Ces derniers disposent surtout de temps d'accès moyens inférieurs, du fait de leur plus grande vitesse de rotation.
- Meilleur rapport performances/prix : `Serial-ATA` (*aka* `SATA`)
  Les disques `SATA` sont nettement moins chers que leurs homologues en `SCSI`. Ainsi, vous pourrez disposer pour le même budget d'un nombre plus important de disques `SATA`, et accroître ainsi soit la capacité de stockage, soit la redondance (et donc les performances en lecture).
  L'encombrement et la consommation électrique de cette solution sont toutefois aussi à prendre en compte.
- Ordre de grandeur
  Les temps d'accès sont de cet ordre : - SATA : 8ms - SAS 10k : 3ms - SAS 15k : 2ms
  Les performances en lecture séquentielle sont de cet ordre : - SATA : 100 à 150 Mo/s - SAS 10k : 100 Mo/s - SAS 15k : 130 Mo/s Ces derniers chiffres sont très dépendants de la densité des plateaux. C'est d'ailleurs la principale raison des excellentes performances des disques SATA en opérations séquentielles.

Gestions des disques

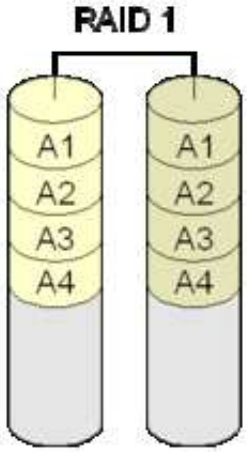
- Placer les tables ou les bases sur différents disques
  Grâce aux *tablespaces*, il est possible de répartir les données sur différents disques et ainsi améliorer les temps d'accès. Par exemple, placer une table souvent utilisée en lecture séquentielle sur le disque `SCSI` le plus rapide et les tables moins sollicitées sur des disques plus lents.
- Séparer le code applicatif et la base
  Dans le cadre d'une application multi-tiers, il est toujours judicieux de bien séparer le *middleware* et la base de données. Disposer d'un serveur entièrement dédié à l'hébergement de PostgreSQL simplifie nettement les tâches d'optimisation.

Cache disque

- Disponible sur les cartes `SCSI` modernes
  La plupart des cartes proposent des fonctions de cache tant en écriture qu'en lecture. Se méfier de *toutes* les cartes embarquées sur les cartes-mères des serveurs : les constructeurs de matériel aiment à rogner sur ces composants pour réduire les coûts de leurs serveurs.
  Préférer une carte `SCSI` séparée (type *LSI Megaraid*), même si leur coût peut dépasser les 1000 euros.
- Cache en lecture
  La plupart de ces caches en lecture sont gérés par un algorithme de type `LRU` (*Least Recently Used*). C'est-à-dire que les blocs les moins récemment utilisés sont sortis du cache en lecture.
- Cache en écriture
  Ce cache stocke les écritures et les écrit d'un coup sur le disque, optimisant ainsi les temps d'écriture. Étant donné que PostgreSQL attend le résultat (positif/négatif) d'une écriture (en mode synchrone du moins, c'est-à-dire `fsync=on` !), on peut alors obtenir des gains en performance impressionnants.
- **Attention** : *battery backup unit* obligatoire sur le contrôleur
  Dans le cas où le cache en écriture est activé, vous devez absolument disposer de piles sur la carte. En cas de défaillance électrique, les données de celui-ci seront en effet effacées.

RAID 1

- `RAID 1` est le plus simple à mettre en place
- *Mirroring*

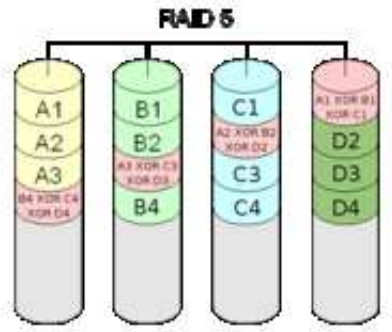


Le `RAID 1` consiste en l'utilisation de disques redondants, c'est-à-dire *n* disques (en général deux) sur lesquels sont copiées exactement les mêmes données.

Si cette solution n'apporte aucun gain de performance en écriture, elle permet en revanche de sécuriser les données en cas de défaillance d'un des disques, et d'augmenter sensiblement les performances en lecture.

RAID 5

- Pas très performant avec PostgreSQL



Le `RAID 5` associe le *stripping* et un système à parité répartie. La parité est en effet répartie circulairement sur les différents disques. En cas de défaillance d'un des disques, les données sont toujours accessibles.

Par exemple, considérons trois disques durs `A`, `B` et `C`, de taille identique. Le système va enregistrer le premier bloc en le répartissant sur les disques `A` et `B` comme en mode `RAID 0` (*stripping*) et, sur le disque `C`, le résultat de l'opération `xor` (ou exclusif) entre `A` et `B`.

Ensuite il va enregistrer le deuxième bloc en le répartissant sur les disques `B` et `C`, puis la parité (`B xor C`) sur le disque `A`, et ainsi de suite en faisant permuter circulairement les disques, à chaque bloc. La parité se trouve alors répartie sur tous les disques. En cas de défaillance d'un disque, les données qui s'y trouvaient peuvent être reconstituées par l'opération `xor`.

Ce système nécessite impérativement un minimum de trois disques durs. Ceux-ci doivent généralement être de même taille, mais un grand nombre de cartes `RAID` modernes autorisent des disques de tailles différentes.

Le problème du `RAID 5` est sa lenteur.

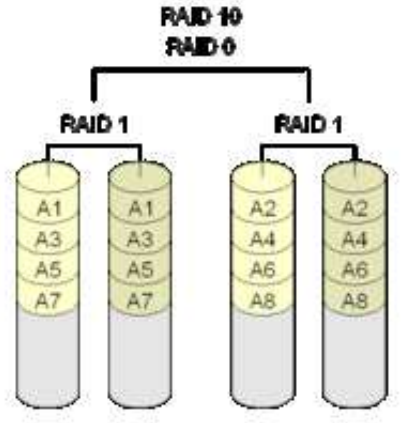
Les performances sont généralement mauvaises en écritures, notamment si la quantité d'informations écrite est plus petite que l'unité de base du `RAID5` (le `stripe`). En effet, à chaque écriture, la parité doit être mise à jour, ce qui se termine par des séquences lectures/modifications/écritures au lieu d'une simple écriture.

Les performances en lecture sont légèrement inférieures à du `RAID0` (à nombre de disques équivalents) car, dans le cas du `RAID5`, il est nécessaire de sauter les blocs de parité.

Voir http ://en.wikipedia.org/wiki/Raid5#RAID_5_performance pour plus de détails.

RAID 10 (1+0)

- `RAID 10` est la configuration optimale pour PostgreSQL
- Ajouter un maximum de disques



`Le RAID 10` est la mise en cascade d'un `RAID 1` (*mirroring*) et d'un `RAID 0` (*stripping*). *n* groupes de deux disques montés en `RAID 1` sont montés entre eux en `RAID 0`.

Ici, il faut que deux disques d'un même groupe rendent l'âme pour que le tout soit perdu, ce qui réduit la probabilité de défaillance.

Le `RAID 10` est plus fiable et plus rapide à reconstruire en cas de panne que le `0+1`. Il est donc plus courant. Comme pour le `RAID 0+1`, un minimum de quatre disques est nécessaire.

SAN

SAN signifie Storage Array Network

- Gestion souple et évolutive du stockage

Il est plus facile de gérer un grand nombre de disques au sein d'un SAN. Ainsi, lorsque la quantité de disque devient problématique, cette solution devient envisageable.

Les SAN possèdent beaucoup plus de mémoire cache que les disques internes. Ainsi ils peuvent offrir des temps d'accès nettement inférieurs et supporter de grosses rafales d'écritures.

- Confort d'utilisation

Suivant les constructeurs, les baies SAN ont des fonctionnalités additionnelles qui facilitent le travail des administrateurs : sauvegarde, réplication, images instantanées (« snapshots ») sont autant de tâches que la technologie SAN peut simplifier.

- Des solutions abordables mais ....

... attention aux coûts cachés !

Les tarifs des solutions SAN sont en baisse depuis plusieurs années (ou les fonctionnalités en hausse, suivant les fabricants). Cependant, la flexibilité d'un SAN implique également une complexité indéniable. Vous aurez peut-être besoin d'un consultant extérieur pour effectuer certaines opérations (installation, maintenance, optimisation). C'est même souvent exigé par le constructeur du matériel. Cela peut alourdir nettement le coût global de votre baie de stockage.

L'autre problématique récurrente avec les technologies SAN Fiber Channel est le manque de compatibilité entre les différentes solutions des différents constructeurs : matrices de compatibilité gigantesque sur l'ensemble de la chaîne allant du disque au firmware du contrôleur, au modèle et firmware du ou des switch, carte, driver, multipathing sur le système d'exploitation. Le respect de l'ensemble de ces contraintes sur l'ensemble d'un réseau SAN peut rendre toute migration fastidieuse.

- Attention à la latence !

Le stockage dans un SAN introduit une certaine latence (le temps de transport sur la liaison « fiber channel »), ce qui peut devenir un facteur limitant pour les écritures. Dans certains cas, l'écriture sur un disque interne reste plus performante.

Il convient de préférer la fibre plutot que les liaisons iSCSI, et de s'assurer que les interfaces sont de bonne qualité.

Plus d'information sur :

http ://wiki.postgresql.org/wiki/Direct_Storage_vs._SAN

Solid State Drive

- Pas de mécanique = moins d'énergie, moins de chaleur !
- Temps d'accès : environ 300 fois plus rapide (0.1 ms contre 3 ms)

Les SDD n'ayant pas de partie mécanique, leur temps d'accès est environ 300 fois plus faibles qu'un disque classique.

Attention : Les disques SSD sont beaucoup plus chers que les disques à plateau. On trouve donc sur le marché des SSD bas de gamme ayant des performances et une durée de vie faibles.

Les prix rapportés à l'opération d'entrée sortie sont très avantageux :

En mars 2010 : - SSD Intel X25-E 64 Go (environ 10 000 IO/s) : 700 euros - Seagate Cheetah 15k 148Go (environ 350 IO/s) : 180 euros

Si votre volumétrie est faible, mais vos besoins en entrées sorties forts, le coût des disques SSD est facile à justifier : O.07 euro l'IO/s en SSD pour 0.5 euro l'IO/s en disque classique.

- Changement de contexte (SSD = RAM ?)

Les disques SSD n'apportent pas seulement une amélioration des performances Il s'agit également de re-penser complètement notre vision du stockage des données. En effet, les disques durs étant 1 millions de fois plus lents que la mémoire RAM en termes de temps d'accès, il était jusqu'ici avantageux de mettre énormément d'information en cache. Les algorithmes étaient pensés pour éviter les accès aléatoires, plus coûteux que les séquentiels.

Désormais, RAM et SSD ont des temps d'accès plus proche, c'est donc toute la « philosophie » de configuration du serveur qui doit être revue !

- Longévité ?

Les cellules des disques SSD ont un nombre de réécritures limité, mais connue et prévisible. Les tous premiers disques durs SSD souffraient d'un problème de longévité, problème résolu depuis quelques années par les algorithmes de 'Wear Levelling', c'est à dire de répartition d'usure. Ils permettent d'éviter qu'un bloc unique soit la cause de la mort du disque dur. Un disque dur 'Entreprise' comme le X25-E d'Intel est un disque SLC (Single Level Cell), dont chaque cellule supporte 100 000 réécritures environ. Avec l'algorithme de Wear Levelling, un disque de ce type supporte la réécriture de n'importe quel bloc en continu pendant 10 ans sans tomber en panne.

Par ailleurs, si un disque SSD "meurt", il reste accessible en lecture seule. Donc il n'y a pas de perte de donnée.

Plus d'informations sur la technologie SSD :

http ://blog.jeb.be/2008/12/08/ssd/

Virtualisation

- Points forts : partage CPU et partage RAM

Ce qui n'est généralement pas important pour les bases de données.

- Point faible : utilisation disque

À moins que la solution de virtualisation permette d'utiliser un disque réel plutôt qu'un disque virtualisé, les performances disques des serveurs virtualisés sont ridicules.

Or, un serveur de bases de données doit ses performances principalement aux à la rapidité des disques contenant les données.

- Peut être valable pour un esclave.

Dans le cadre d'une réplication asynchrone, il peut être intéressant de placer votre esclave sur un serveur virtualisé. Néanmoins, cela voudra dire qu'en cas de failover ou switchover, vous vous retrouverez en mode dégradé. De plus, la réplication asynchrone utilise généralement des tables de logs sur le maître. Il faudra s'assurer que ces dernières ne grossissent pas trop, ce qui laisserait à penser que l'esclave ne tient pas le rythme des écritures.

- Performances aléatoires

L'autre gros inconvénient des serveurs virtualisés, c'est qu'il devient très difficile de comprendre d'où vient une lenteur. Est-ce parce que le serveur de bases de données est mal configuré ou est-ce que, à ce moment-là, un autre serveur virtualisé avait besoin de beaucoup de mémoire ?

5 principes de base

1. Disques > RAM > CPU

   Lors du choix du matériel, les temps d'accès aux données doivent prévaloir sur la puissance de calcul.

2. Plus de têtes de lecture, c'est mieux

   Il est plus intéressant de disposer de plusieurs disques de tailles moyennes plutôt qu'un disque unique et énorme.

3. Séparer les journaux de transaction (`pg_xlog`) de la base

   Pour la sécurité et pour les performances, il est très pertinent d'isoler les données et les journaux de transactions.

4. `RAID 1+0/0+1` > `RAID 5`

   Les combinaisons *mirroring* + *stripping* ou *stripping* + *mirroring* sont plus intéressantes que le `RAID5`.
   Le `RAID 5` rend perplexes les *DBA* : « Où sont mes données ? »

5. Séparer les applications

   Il est essentiel de séparer le code applicatif (*traitements*) et PostgreSQL (*données*).
   De la même façon, l'installation concurrente de différents serveurs de données sur la même machine est à proscrire !

Partie 2 : Kernel

- Recommandations
- Configuration
- Memory Overcommit
- Sémaphores

Cette partie détaille les spécificités du noyau linux (`kernel`) qu'un administrateur système doit surveiller sur un serveur Linux hébergeant une ou plusieurs bases PostgreSQL.

Ces spécificités concernent principalement la gestion de la mémoire du système par le noyau Linux.

Kernel : recommandations

- Utiliser un `kernel` à jour est un gage de performances !

  `2.2 << 2.4 << 2.6`

  Le noyau 2.6.26 apporte une série d'améliorations intéressantes pour PostgreSQL, notamment en permettant de demander l'écriture sur disque des modifications réalisées dans un fichier sans mettre à jour les métadonnées.
  Plus d'informations :

  http ://kernelnewbies.org/Linux_2_6_26#head-1ed58790729b4b289d0612fb05c367f59491e884

- Attention avec la version 2.6, utiliser de préférence une version ultérieure à la 2.6.8
  Les versions 2.6.1 à 2.6.8 sont reconnues pour leur problème de lenteur.
- Éviter les `kernels` recompilés, *allégés*
  Quelle est la « maintenabilité » d'une telle pratique ?
  - ⇒ À l'arrivée d'un nouveau matériel, une recompilation est nécessaire
  - ⇒ On oublie toujours un détail...
- Préférer les `kernels` officiels des distributions. Ils sont mis à jour régulièrement.
  Le gain en performances, sauf dans de rares exceptions, d'un kernel personnalisé ne vaut pas la peine de prendre des risques !

Kernel : configuration

- Les paramètres du noyau se trouvent dans `/etc/sysctl.conf`

La syntaxe du fichier `/etc/sysctl.conf` est la suivante :

```
kernel.<parametre> = <valeur>
```

- `sysctl -p` pour que les modifications soient appliquées

Pour que le *kernel* prenne en compte une telle modification, utiliser la commande `sysctl` en tant que *root* :

```
# sysctl -p
```

Memory Overcommit (1/2)

- L'implémentation du memory overcommit sous Linux n'est pas optimale pour PostgreSQL

Certaines applications ont tendance à demander beaucoup plus de mémoire au système que ce qu'elles utilisent réellement. Dans certaines circonstances, la méthode du `memory overcommit` permet d'éviter des échecs lors des requêtes d'allocation mémoire.

- Peut provoquer l'arrêt du serveur PostgreSQL

Cependant, cette implémentation du `memory overcommit` n'est pas idéale pour PostgreSQL. Le noyau pourrait arrêter le serveur PostgreSQL si les demandes d'allocation des autres processus provoquaient une saturation de la mémoire virtuelle.

Lire http ://article.gmane.org/gmane.comp.audio.jackit/19998 pour plus de détails.

Memory Overcommit (2/2)

- Utiliser PostgreSQL sur un serveur dédié à lui seul.

Une façon d'éviter le problème revient à lancer PostgreSQL sur une machine où l'on peut s'assurer que les autres processus ne provoqueront pas de manques de mémoire.

- Sur Linux version 2.6 et supérieures, il est possible et préférable de désactiver l'overcommit

Une meilleure solution est de modifier le comportement du noyau de façon à ce qu'il ne tue pas les processus « au hasard » par manque de mémoire. Ceci se fait en sélectionnant le mode « `don't overcommit` » dans `/etc/sysctl.conf` :

```
vm.overcommit_memory=2
```

De cette façon, l'espace adressable ne pourra pas être supérieure à la taille du swap auquel est ajouté un pourcentage de la mémoire physique. Pour connaître la taille de cette mémoire, `grep CommitLimit /proc/meminfo`.

Pour de plus amples informations, se reporter à la documentation du noyau

```
Documentation/vm/overcommit-accounting
```

Mémoire partagée : shmall et shmmax

Le fichier postgresql.conf est par défaut idéal pour une machine possédant... 64 Mo de RAM !

Cela s'est arrangé depuis la version 8.2 : la configuration par défaut de PostgreSQL exploite un peu mieux les ressources des serveurs modernes, sans toutefois être optimale.

Il faudra donc l'adapter. Certaines modifications du fichier postgresql.conf nécessitent d'augmenter les valeurs de :

```
/proc/sys/kernel/shmall
/proc/sys/kernel/shmmax
```

Deux entrées de /proc déterminant la taille des zones de mémoire partagé utilisables par un processus et mises à disposition par le noyau Linux.

SHMALL indique la quantité totale de mémoire partagée allouable par l'ensemble des processus de la machine. Son unité est la page mémoire (4ko par défaut). Par défaut sous Linux, la valeur est de 2097152, soit 8 Go.

SHMMAX indique la taille maximum d'un segment de mémoire partagée. Son unité est l'octet. PostgreSQL alloue un seul segment de mémoire partagée pour les shared buffers, les wal buffers, les verrous, et en versions <= 8.3, la Free Space Map. Un segment doit être de taille inférieure à SHMALL. Attention, son unité et celle de SHMALL sont différentes (d'un facteur 4096).

Par exemple pour 512Mo, on pourra dupliquer les valeurs dans le fichier /etc/sysctl.conf comme suit :

```
# 512*1024*1024 = 536870912
kernel.shmmax = 536870912
```

Dans ce contexte, inutile de changer SHMALL, il est suffisant.

Pour que le *kernel* prenne en compte une telle modification, utiliser la commande sysctl en tant que *root* :

```
# sysctl -p
```

Si vous voulez vérifier les valeurs, il existe un format plus lisible que l'interface sysctl (ici vous voyez les valeurs par défaut) :

```
# ipcs -l -m ------ Shared Memory Limits -------- max number of segments = 4096 max seg
size (kbytes) = 32768 max total shared memory (kbytes) = 8388608 min seg size (bytes) =
1
```

Partie 3 : Systèmes de fichiers

Cette partie présente les systèmes de fichiers les plus communs et leurs apports éventuels dans le cadre de leur utilisation avec PostgreSQL.

- La dynastie `ext`

Le système de fichier le plus répandu sur les systèmes Linux est EXT (`ext2` / `ext3` / `ext4`)

Nous allons voir que `ext3` est certainement le meilleur choix.

- Les outsiders

D'autres solutions moins connues peuvent apporter des performances intéressantes, notamment `XFS`, `JFS`, `BTRFS`

- Les systèmes de fichiers à snapshot

Les systèmes `ZFS` et `LVM` permettent de faire des images instantanées (snapshots) et sont très utiles pour faire des sauvegardes à chaud de PostgreSQL

- Les systèmes à éviter !

`ReiserFS` et `FAT` sont des mauvais choix. Nous verrons en quoi ils sont mal ou pas adaptés pour PostgreSQL

ext2

- Rapide
  ext2 est reconnu comme étant un des systèmes les plus performants en terme de temps de réponse.
  Étant donné que PostgreSQL dispose de son propre système de journalisation (WAL), la journalisation au niveau disque n'a que peu d'intérêt.
- **MAIS** problèmes de récupération de données
  ext2 n'est pas très fiable. En cas de redémarrage accidentel, la commande fsck peut bloquer le redémarrage et entrainer une coupure de service d'autant plus importante que les disques sont gros.

ext3

- Fiable et performant
  ext3 reprend les avantages de ext2 et y ajoute un système de journalisation qui rend le système plus résistant aux pannes. Le ralentissement en écriture est de l'ordre de 5 à 10%.
- Système par défaut sur RedHat et debian, entre autres
- noatime, data=writeback, stride, dir_index
  La plupart des grandes distributions Linux considèrent ext3 comme le meilleur choix. Il faut veiller cependant à optimiser la configuration du système de fichiers qui contient les données (répertoire $PGDATA)
  L'option data=writeback peut être activée lors du montage de la partition. Cette option permet de conserver la performance de l'ext2 en utilisation normale avec la rapidité du fsck de l'ext3 en cas de crash (autrement dit : il ne journalise que les méta-données et laisse les écritures disque au processus "sync" habituel).
  De plus, par défaut, ext3 conserve la date du dernier accès (access time) de tous les fichiers. Cette information est inutile pour PostgreSQL et elle ralentit les écritures. On peut donc la désactiver dans le fichier fstab avec une ligne du type

  ```
  /dev/sda4    /data    ext3    noatime    0    0
  ```

  Il est également possible d'optimiser la valeur de l'option « stride » pour l'adapter aux spécificités des disques.
  On peut enfin activer l'indexation des répertoires avec l'option « dir_index », ce qui améliore les lectures.
  Sources :

  http ://momjian.us/main/writings/pgsql/hw_performance/#SECTION00013000000000000000
  http ://www.powerpostgresql.com/download/TFCKUpload/5.x-pdf http ://wiki.centos.org/HowTos/Disk_
  http ://en.opensuse.org/Speeding_up_Ext3

ext4

- Simple amélioration d'ext3
  ext4 est le successeur du système de fichiers ext3, principalement destiné aux systèmes basés sur GNU/Linux. Il est cependant considéré par ses propres concepteurs comme une solution intérimaire en attendant le vrai système de nouvelle génération : Btrfs.
- Système par défaut sur Ubuntu (version 9.10), Fedora (version 12), OpenSuSE (version 11.2), Mandriva (version 2010), entre autres.
- Compatible avec ext3

Une partition ext4 est compatible avec ext3, ce qui signifie qu'elle peut être montée comme une partition ext3 (en utilisant le type de système de fichiers « ext3 » lors du montage).

L'inverse est également possible, le montage d'un système de fichiers ext3 comme un ext4 (en utilisant le type de système de fichiers "ext4dev").

- Mêmes conseils de tuning que pour ext3
- Dans la majorité des cas, ext4 est plus rapide qu'ext3

http ://gmrpgsql.tumblr.com/post/73798984/initial-ext3-vs-ext4-results

BTRFS

Prononcez « Better FS »
- Developpé par Oracle !

BTRFS est considéré comme un système de fichier de nouvelle génération. Il est en position très favorable pour être le remplaçant de la lignée des « Ext » dont « Ext4 » est la dernière mouture. Une étape importante a été franchie en janvier 2009 avec l'intégration de Btrfs dans la branche principale de Linux version 2.6.29.

On dit souvent que Ext4 est chargé de faire l'intérim entre Ext3 et BTRFS
- Le concept d' `extent`

BTRFS reposte sur la notion d'extent, c'est-à-dire une zone contiguë qui est réservée chaque fois qu'un fichier est enregistré sur le disque dur. Cela permet, en cas d'écriture ultérieure sur le fichier, d'ajouter les nouvelles données dans l'extent au lieu de devoir écrire dans une autre zone du disque dur (ce qui augmente la fragmentation). Les gros fichiers sont ainsi stockés de façon bien plus efficace et rationnelle. Pour les petits fichiers, Btrfs utilise une astuce puisqu'il stocke les données directement dans le système d'extent lui-même sans avoir à allouer un bloc mémoire séparé.

Par ailleurs, BTRFS est prévu pour être utilisé sur des disques durs de très forte capacité : tous les blocs comportent une somme de contrôle, les données pouvant être redondées sur plusieurs disques durs, et les sommes de contrôle permettant de garantir la qualité de la donnée lue, et rejeter un bloc comportant une erreur, ce qui devient de plus en plus fréquent à mesure que les capacités des disques augmentent.
- Copy on Write

Le Copy-on-write ou copie sur écriture (souvent désigné par son sigle anglais "COW") est une stratégie d'optimisation utilisée en programmation informatique, mais assez récente dans son emploi pour les systèmes de fichiers.

Un système de fichier COW n'écrase jamais une donnée à modifier. Si une zone d'un fichier doit être réécrite, une seconde version est allouée ailleurs sur le périphérique et écrite, l'ancienne version n'étant détruite que plus tard (quand elle n'est plus nécessaire). Cette technique présente de nombreux avantages : le système de fichiers n'a plus besoin de maintenir un journal puisqu'il est en permanence cohérent, et peut proposer un grand nombre d'instantanés (snapshots) simultanément, sans dégradation notable de performance.

Cette technique présente par contre quelques défauts :
- éviter la fragmentation est assez complexe, et souvent impossible. Toutefois, par nature, ces systèmes de fichiers peuvent être défragmentés en ligne avec un verrouillage minimal, et donc en impactant peu les applications hébergées ;
- le système de fichiers a tendance à réaliser le même travail que la base de données : présenter plusieurs versions de la même donnée aux utilisateurs. On voudra donc probablement (quand btrfs sera prêt pour la production) désactiver le 'COW' au niveau des données pour un SGBD, en montant le système btrfs avec l'option -o nodatacow, prévue justement pour ces cas de figure.
- Expérimental mais prometteur

XFS

- Linux >= 2.6.x
  XFS est inclus par défaut avec les versions du noyau Linux 2.5.xx et 2.6.xx, mais n'est pas disponible dans les versions 2.4.xx, excepté par le biais d'un *patch*
- Système journalisé
- Très bonnes performances
  XFS est un système de fichiers 64-bit journalisé créé par *SGI* pour son système d'exploitation IRIX. En mai 2000, *SGI* place XFS sous la licence *GPL*.

JFS

JFS (pour *Journaled File System*) est un système de fichiers journalisé mis au point par *IBM* et disponible sous licence *GPL*.

- Rapide
  Selon de nombreux *benchmarks*, JFS est très performant et fiable
- **MAIS** peu répandu et peu supporté
  Malgré ses qualités, JFS reste un système marginal.

ZFS

- Développé pour `Solaris`

ZFS est un système de fichiers 128 bits, ce qui signifie qu'il peut fournir 16 milliards de milliard de fois ce que fournissent les systèmes de fichiers 64 bits actuels. Ainsi, les limites de ZFS sont pratiquement impossibles à atteindre.

Le 'Z' initial fait référence à Zettaoctet (1 million de petaoctets).

- Images instantanées (« Snapshot »)

  ZFS permet d'effectuer une copie cohérente (« clone ») du répertoire PGDATA, même lorsque le serveur PostgreSQL fonctionne. On parle alors de « copie à chaud cohérente ».

  Cette fonctionnalité rend les opérations de sauvegarde triviales. Elle est également très utile pour cloner un maître PITR et lancer des traitements importants (« data mining ») sur le clone.

Plus d'information :

http ://lethargy.org/~jesus/archives/92-PostgreSQL-warm-standby-on-ZFS-crack.html

- Attention à configurer ZFS **avant** d'installer PostgreSQL !

Certains paramètres (notamment « ZFS recordsize ») doivent être définis avant l'initialisation de l'instance PostgreSQL.

Plus d'information :

http ://www.solarisinternals.com/wiki/index.php/ZFS_Best_Practices_Guide

LVM

LVM signifie « Logical volume management » (en français : Gestion par volumes logiques)

- Couche d'abstraction entre le disque et le FS

En soi, LVM n'est un système de fichiers mais plutôt une virtualisation des espaces physiques de stockages (disques durs, SAN).

- Souple et pratique

LVM apporte un certain confort au niveau des tâches d'administration système : on peut notamment modifier la taille des volumes logiques sans perte de données. Suivant le système de fichiers, cette opération peut se dérouler « à chaud », c'est-à-dire sans démonter la partition.

LVM permet également de réaliser des images instantanées (« snapshots ») qui sont très utiles pour la sauvegarde à chaud du répertoire PGDATA de PostgreSQL.

- Attention aux performances !

LVM est une couche d'abstraction au-dessus des disques. En tant que telle, elle peut avoir un coût, la plupart du temps négligeable, mais qu'il convient de mesurer et vérifier avant la mise en production. Sur certaines baies SAN par exemple, le schéma de partitionnement du disque pourrait entrainer un problème d'alignement avec le LUN mis à disposition par la baie, et dégrader les performances. Il existe bien entendu des options pour forcer un alignement, mais tout cela demande d'avoir pris conscience du problème suite à un test de performance.

Il convient aussi de ne pas mélanger de disques de niveaux de performance différents dans un volume group.

NTFS

- Windows 2000, XP, 2003, Vista et 2008
  NTFS est le système de fichier standard de Windows NT et de ses descendants
- Testé seulement sur des systèmes 32 bits.
- Pas de *tablespaces* natifs sous WINDOWS NT4
  On ne peut en effet pas disposer des *tablespaces* sur les systèmes ne possédant pas des liens symboliques (ln -s sous un *UNIX*). Toutefois, le logiciel junction disponible ici :

  http ://www.microsoft.com/technet/sysinternals/FileAndDisk/Junction.mspx

  permet d'utiliser les liens symboliques.
  Pour plus d'informations sur les limitations du système NTFS, consultez la FAQ du logiciel pginstaller

  http ://pginstaller.projects.postgresql.org/faq/FAQ_windows.html
- *tablespaces* possible avec la 8.2 (versions antérieures déconseillées sous windows). Attention à leur sauvegarde, peu de logiciels de sauvegarde sous windows les reconnaissent pour ce qu'ils sont (des liens) et les confondent avec de vrais répertoires.

FAT

- À proscrire.
  La priorité numéro 1 de PostgreSQL est l'intégrité des données. Les systèmes `FAT` et `FAT32` n'offrent pas les garanties nécessaires pour assurer cette intégrité.
  De plus, la sécurité des données est également compromise.
  Pour toutes ces raisons, il est formellement déconseillé d'installer PostgreSQL sur une partition `FAT`.

ReiserFS

- Par défaut sur SuSE pendant longtemps (mais plus maintenant)
  De fait, cette distribution a fait le choix de ce système de fichier, notamment grâce au couple LVM/ReiserFS, qui permet le repartitionnement à chaud.
  Désormais, Novell intègre EXT4 par défaut : http ://www.novell.com/linux/filesystems/faq.html
- Système controversé, avenir incertain
  À tort ou à raison, ReiserFS est probablement le système de fichier le plus critiqué, notamment par certains développeurs du kernel Linux. Il est de fait proposé en option seulement sur la plupart des distributions Linux.
  Par ailleurs, l'auteur du projet, Hans Reiser a été condamné à 15 ans de prison, ce qui laisse planer de sérieux doutes sur la pérénité de ce système
- Versions = 3
  Il est fortement conseillé d'utiliser une version récente de la version 3 de ReiserFS. La version 4 est encore en cours de développement. Par ailleurs, la maintenance de la version 3 n'est plus très active, depuis la condamnation de Hans Reiser.
- Très performant avec les petits fichiers
  ReiserFS est beaucoup plus efficace qu'ext2 ou ext3 pour ce qui concerne le stockage des petits fichiers (moins de quelques ko). Ainsi, lors de la conversion vers ReiserFS d'une partition contenant 2 Go de données stockées en ext2, 200 Mo d'espace libre ont été gagnés.
  Plus d'information :

  http ://www.ubuntugeek.com/how-to-increase-ext3-and-reiserfs-filesystems-performance.html

NFS

NFS est un protocole développé par *Sun Microsystems* qui permet à un ordinateur d'accéder à des fichiers via un réseau.

- À proscrire !
  Utiliser NFS, notamment pour répartir les disques de stockage de PostgreSQL via un réseau est **une très mauvaise idée**.
  Exception faite de l'écriture des journaux applicatifs de PostgreSQL (/var/log/postgresql/) : cette technique peut être idéale dans le cas où l'on trace l'ensemble de l'activité du serveur (requêtes SQL complètes, temps de réponse, utilisateurs, etc.), comme c'est le cas lors de l'utilisation de pgFouine par exemple.
  Plus d'information sur les dangers du protocole NFS :

  http ://www.time-travellers.org/shane/papers/NFS_considered_harmful.html

  Un exemple de problème entre PostgreSQL et NFS :

  http ://archives.postgresql.org/pgsql-novice/2007-08/msg00123.php

- Pour partager l'accès à une base, utiliser Slony et pgpool
  Il existe des projets qui permettent de répondre aux besoins de partage de données, de réplication ou de serveurs distribués.
  Le site de Slony : http ://www.slony.info.
  Le site de pgpool : http ://pgpool.projects.postgresql.org.
- ... ou mettre en place un SAN robuste
  Si le budget accordé aux achats de matériel le permet, les solutions basées sur une baie SAN (Fiber Channel, iSCSI) **robuste** sont nettement préférables à un simple montage NFS.

Recommandations

- Utiliser `ext3`, `ZFS` ou `XFS`
- Choisir le système de fichiers le mieux supporté par la distribution
- Les système de fichiers avec journalisation ne sont pas beaucoup plus lents
- Monter les volumes avec l'option `noatime`

Sur windows :

```
fsutil behavior set disablelastaccess 1
```

Partie 4 : Manipulation du serveur

- Utilisateur `postgres`
- Utilisation de `initdb`
- Démarrer le serveur
- Recharger la configuration
- Arrêter le serveur

Cette partie présente les commandes basiques utiles lors de l'initialisation du *cluster* PostgreSQL et de son utilisation au quotidien.

Utilisateur `postgres`

- PostgreSQL doit être lancé sous un compte utilisateur séparé
- Le nom `postgres` est souvent utilisé.
  Ce compte devrait seulement être le propriétaire des données gérées par le serveur (`chmod -R 700` sur les répertoires) et ne devrait pas être partagé avec d'autres démons : par exemple, utiliser l'utilisateur `nobody` est une mauvaise idée.
  Pour les distributions intégrant un système de packages (`mandriva`, `debian`), cet utilisateur est créé automatiquement.

Créer le groupe de base de données

1ère opération : initialiser un emplacement de stockage

Commande

```
$ initdb -D /chemin/vers/emplacement/de/stockage
```

Un groupe de bases de données (aussi connu sous le nom de *cluster*, référencé par `$PGDATA` dans la plupart des scripts et applications) est une collection de bases données et est géré par une seule instance d'un serveur de bases de données en cours d'exécution. Après initialisation, un groupe de bases de données contiendra une base de données nommée `postgres`, qui a pour but d'être la base de données par défaut utilisée par les outils, les utilisateurs et les applications tiers.

Pour les distributions intégrant un système de packages (`mandriva`, `debian`), cette opération est effectuée automatiquement.

La version 8.3 a ajouté l'option `-X` qui permet de créer le répertoire des journaux de transaction en dehors de `$PGDATA`. Il est ainsi possible de le placer directement sur un autre disque, ce qui va améliorer les performances du système disque.

Démarrer

Il existe trois méthodes pour démarrer son cluster PostgreSQL.

- /etc/init.d/postgresql start
  Cette commande est disponible sur la plupart des distributions **Linux**.
- `postmaster -d /chemin/vers/emplacement/de/stockage`
  Cette commande se révèle fastidieuse à utiliser. Notamment parce qu'il faut penser à rediriger la sortie standard et lancer la commande en tâche de fond. Exemple

  ```
  postmaster -d /usr/local/pgsql/data >journaux_trace 2>&1 &
  ```

- `pg_ctl start -l journaux_trace`
  `pg_ctl` est un *wrapper* dont le but est de simplifier la manipulation du démon `postmaster`.

Recharger la configuration *online*

On peut utiliser au choix l'une des quatre commandes ci-après.

- `SIGHUP` sur le processus `postmaster`
- `/etc/init.d/postgresql reload`
- `pg_ctl reload`
  La commande `reload` permet de prendre en compte certaines modifications des fichiers de configuration (`pg_hba.conf`, etc.) sans arrêter le service.
- `select pg_reload_conf()`
  Utilisable uniquement pas les super-utilisateurs.

Arrêter le serveur

À nouveau, on peut utiliser pour cela plusieurs méthodes.

- `/etc/init.d/postgresql stop`
  Cette commande est disponible sur les principales distributions Linux.
- `pg_ctl stop`
  Le programme `pg_ctl` fournit une interface agréable pour arrêter le serveur.
- `kill -int $(head -1 /usr/local/pgsql/data/postmaster.pid)`
  Il est préférable de ne pas utiliser *sigkill* (`kill -9`) pour arrêter le serveur. Le faire empêchera le serveur de libérer la mémoire partagée et les sémaphores. Si le serveur est redémarré alors que d'autres processus postgres utilisent toujours l'ancienne mémoire partagée, il existe un risque important de corruption des données. Voir http ://dalibo.org/septembre_2008 pour plus de détails.

Partie 5 : Le cluster PostgreSQL

- Présentation
- `initdb`
- `PG_VERSION`
- `base/`
- `pg_clog/`
- `pg_multixact/`
- `pg_subtrans/`
- `pg_tblspc/`
- `pg_twophase/`
- `pg_xlog/`
- `pg_postmaster.opts`
- `postmaster.pid`

Cette partie du module présente un à un les fichiers et sous-répertoires présents dans le *cluster*.

Cluster : présentation

- *cluster* PostgreSQL = emplacement physique contenant les bases du serveur PostgreSQL
- Créé par la commande `initdb`
- Plusieurs *clusters* PostgreSQL sur un même serveur possible

Certaines distributions `Linux` proposent des paquets additionnels permettant d'avoir plusieurs versions de PostgreSQL installées :

```
postgresql-client-common - manager for multiple PostgreSQL client versions
postgresql-common - manager for PostgreSQL database clusters
postgresql-7.4 - object-relational SQL database, version 7.4 server
postgresql-8.1 - object-relational SQL database, version 8.1 server
postgresql-8.2 - object-relational SQL database, version 8.2 server
```

Cependant, ce n'est pas recommandé pour un serveur de production.

Cluster : `initdb`

- Crée l'ensemble des fichiers d'un *cluster* PostgreSQL
- À ne pas confondre avec la commande `SQL CLUSTER`
- Peut être appelée soit automatiquement
  généralement par des scripts contenus dans les paquets binaires des distributions linux ou l'installateur pour Microsoft Windows.
- soit manuellement
  par l'administrateur de base de données.

Cluster : `pg_hba.conf`

- Fichier gérant les droits d'accès aux bases de données
- Configuration souple basée sur des règles simples
- Lecture du fichier par l'utilisateur et le groupe `postgres` uniquement

Des explications plus détaillées sur ce fichier sont données dans d'autres modules de la formation PostgreSQL.

Cluster : `pg_ident.conf`

- Fichier gérant les authentifications basées sur `ident`
  Il permet de *mapper* des noms d'utilisateurs pour `ident` (des noms d'utilisateur système) à leur nom d'utilisateur PostgreSQL correspondant
- Lecture du fichier par l'utilisateur et le groupe `postgres` uniquement
  Des explications plus détaillées sur ce fichier sont données dans d'autres modules de la formation PostgreSQL

Cluster : PG_VERSION

- Numéro de version majeure du serveur PostgreSQL auquel appartient le `cluster`
- Généralement lu par des outils d'administration
  Ils peuvent ainsi détecter rapidement leur compatibilité avec le `cluster`

Cluster : `postgresql.conf`

- Fichier gérant la configuration du serveur PostgreSQL
- Configuration basée sur un système `clé = 'valeur'`

Des explications plus détaillées sur ce fichier sont données dans d'autres modules de la formation PostgreSQL

Cluster : `postmaster.pid`

- Numéro de processus du `postmaster`
- Il contient aussi l'*id* du sémaphore utilisé

Cluster : `postmaster.opts`

- Fichier contenant les options passées en ligne de commande lors du démarrage du `postmaster`
  Exemple sous `debian` :

```
$ cat postmaster.opts

/usr/lib/postgresql/8.1/bin/postgres \
-D /var/lib/postgresql/8.1/main \
-c config_file=/etc/postgresql/8.1/main/postgresql.conf \
-c hba_file=/etc/postgresql/8.1/main/pg_hba.conf \
-c ident_file=/etc/postgresql/8.1/main/pg_ident.conf \
-c external_pid_file=/var/run/postgresql/8.1-main.pid
```

Cluster : répertoire base/

- Contient autant de sous-répertoires que de bases de données
- Les sous-répertoires de base/ sont nommés de manière numérique
- Le numéro correspond à l'oid de pg_database

Voici la requête permettant de trouver la correspondance entre nom de la base et oid :

```
template1=# select   oid, datname
            from     pg_database
            order by oid ;
```

Voici un exemple de la correspondance entre pg_database.oid et les sous-répertoires de base/ :

```
$ sudo ls -C1 /var/lib/postgresql/8.1/main/base
1
10792
10793
232583
232649
233108

template1=# select oid, datname from pg_database order by oid ;
  oid   |  datname
--------+-----------
      1 | template1
  10792 | template0
  10793 | postgres
 232583 | test
 232649 | cave
 233108 | benchs
(7 lignes)
```

- Dans le répertoire d'une base, se trouvent tous les fichiers correspondants aux tables et index.

Les fichiers sont nommés suivant le pg_class.relfilenode de la relation (table ou index) associée. Les données sont découpées en segments de segment_size octets (1 Go par défaut). Un suffixe numérique est ajouté dans le cas où plusieurs segments sont nécessaires.

La version 8.4 apporte deux autres suffixes :

- _fsm : ce fichier contient les informations sur les blocs libres de la table associée (ce qui était auparavant conservé en mémoire).
- _vm : ce fichier contient la liste des pages qui n'ont pas de lignes mortes (et qui du coup ne nécessitent pas de VACUUM).

Cluster : répertoire `global/`

- Contient autant de sous-répertoires et fichiers que nécessaire pour stocker les objets communs à toutes les bases de données du cluster
- Les fichiers de `global/` sont nommés de manière numérique
- Le numéro correspond à l'`oid` de l'objet, qui est obligatoirement contenu dans le `tablespace` nommé `pg_global`

```
test=# select relname, relfilenode
test-# from   pg_class
test-# where  reltablespace =
        ( select oid from pg_tablespace where spcname = 'pg_global' )
test-# order by relfilenode ;
```

Voici comment trouver la correspondance entre les fichiers contenus dans `global/` et la base de données :

```
$ sudo ls -C1 /var/lib/postgresql/8.1/main/global/
10290
10292
10296
10298
1136
1137
1213
1214
1232
1233
1260
1261
1262
2671
2672
2676
2677
2694
2695
2697
2698
pg_auth
pg_control
pg_database
pgstat.stat

test=# select relname, relfilenode
test-# from pg_class
```

```
test-# where reltablespace= (select oid from pg_tablespace where
spcname='pg_global')
test-# order by relfilenode;
              relname               | relfilenode
------------------------------------+-------------
 pg_pltemplate                      |        1136
 pg_pltemplate_name_index           |        1137
 pg_tablespace                      |        1213
 pg_shdepend                        |        1214
 pg_shdepend_depender_index         |        1232
 pg_shdepend_reference_index        |        1233
 pg_authid                          |        1260
 pg_auth_members                    |        1261
 pg_database                        |        1262
 pg_database_datname_index          |        2671
 pg_database_oid_index              |        2672
 pg_authid_rolname_index            |        2676
 pg_authid_oid_index                |        2677
 pg_auth_members_role_member_index  |        2694
 pg_auth_members_member_role_index  |        2695
 pg_tablespace_oid_index            |        2697
 pg_tablespace_spcname_index        |        2698
 pg_toast_1260                      |       10290
 pg_toast_1260_index                |       10292
 pg_toast_1262                      |       10296
 pg_toast_1262_index                |       10298
(21 lignes)
```

Cluster : répertoire `pg_clog/`

- Il contient toutes les données d'état de validation des transactions
- Il s'agit d'un fichier binaire : ne pas le modifier

Cluster : répertoire `pg_log/`

- Peut se trouver ailleurs (dépend de la configuration de `postgresql.conf`
- Il contient les journaux applicatifs
- Il s'agit de fichiers de traces dont la rotation, le volume et le contenu peuvent être configurés dans le fichier `postgresql.conf`
  Attention à la volumétrie dans ce dossier.

Cluster : répertoire `pg_multixact/`

- Il contient des données utilisées pour l'implémentation de SELECT FOR SHARE.
- Ne pas toucher aux fichiers sous peine de corruption des données

Cluster : répertoire `pg_subtrans/`

- Il contient les données d'états des sous-transactions
- Utilisé depuis que PostgreSQL possède la fonctionnalité d'imbrication des transactions (SAVEPOINT)

Cluster : répertoire pg_tblspc/

- Il contient les liens symboliques vers les tablespaces
- On peut trouver facilement la correspondance dans la base de données grâce à un requêtage de pg_tablespace

```
test=# select oid, *
       from   pg_tablespace
       where  spclocation <> '' ;
```

Il ne faut pas oublier d'inclure oid dans la requête sur pg_tablespace pour faire cette correspondance :

```
test=# select oid,* from pg_tablespace where spclocation <>'' ;
  oid   |  spcname  | spcowner |  spclocation   | spcacl
--------+-----------+----------+----------------+--------
 232711 | tbs_index |    16386 | /opt/tbs_index |
   (1 ligne)

$ ls -l pg_tblspc/
total 0
lrwxrwxrwx 1 postgres postgres 14 2006-08-25 11 :50 232711 -> /opt/tbs_index
```

Cluster : répertoire pg_twophase/

- Il contient les fichiers d'état pour les transactions préparées
- Ce répertoire n'existe que depuis l'apparition du `Two Phase Commit` dans PostgreSQL : les transactions sont écrites sur disques pour un `COMMIT` ou un `ROLLBACK` ultérieur. Ce mécanisme permet donc de reprendre une transaction qui a été préparée, ultérieurement, dans une autre transaction, et permet donc de garantir le bon déroulement d'une transaction distribuée sur plusieurs bases, pouvant elles-mêmes être distribuées sur plusieurs machines.
- Il est aussi parfois utile d'être sûr que certaines opérations sont atomiques

On peut créer une transaction préparée comme suit :

```
test=# PREPARE transaction '2PC';
PREPARE TRANSACTION
test=# select * from foobar;
 a | b
---+---
 3 | 4
 5 | 6
 7 | 6
(3 lignes)

test=# insert into foobar values (10,10);
INSERT 0 1
```

Puis quitter brutalement la session... et faire un `ls -lah` dans le répertoire `$PGDATA/pg_twophase/` :

```
hector :/var/lib/postgresql/8.1/main# ls -l pg_twophase/
total 4
-rw------- 1 postgres postgres 280 2006-09-05 17 :32 000875D1
```

Ensuite, on se connecte à nouveau sur la base de tests, pour *commiter* la transaction préparée :

```
test=# commit prepared '2PC';
COMMIT PREPARED
test=# select * from foobar;
 a  | b
----+----
  3 |  4
  5 |  6
  7 |  6
 10 | 10
(4 lignes)
```

Un nouveau `ls -lah` dans le répertoire `$PGDATA/pg_twophase/` montre que le fichier a disparu après le `COMMIT PREPARED` :

```
hector :/var/lib/postgresql/8.1/main# ls -l pg_twophase/
total 0
```

Cluster : répertoire `pg_xlog/`

- Il contient les journaux de transaction (suivant la technique du *Write Ahead Log*)
- Il s'agit de fichiers de `16 Mo`, *en rotation*, dont le nombre maximal est déterminé par la variable de configuration `checkpoint_segments` (au maximum 3 fois la valeur de ce paramètre plus 1)

Partie 6 : Processus

- Architecture basée sur les processus
- Plusieurs types de processus

Certains sont dédiés à certaines tâches, comme l'écriture des fichiers sur le disque. D'autres sont dédiés aux connexions utilisateur. Il y a un processus par connexion.

- Gestion mémoire différente
- Certains obligatoires, d'autres activables

Cette avant-dernière partie du module présente un à un les différents types de processus d'un serveur PostgreSQL.

Processus postmaster

- Processus père de tous les autres
- Commence par réclamer la mémoire partagée
- Exécute certains démons au démarrage

Ce sont les suivants :
- processus de gestion des journaux applicatifs (si activé) ;
- processus de collecte des statistiques ;
- processus autovacuum ;
- processus d'écriture en tâche de fond ;
- processsus d'écriture des journaux de transactions en tâche de fond.

- Écoute les connections entrantes, soit via le socket soit via le port TCP/IP soit via les deux
- Deux fonctions : pg_cancel_backend et pg_terminate_backend

La première demande l'annulation de la requête en cours d'exécution sur le processus fourni en paramètre. La seconde, disponible seulement depuis la 8.4, ferme la connexion du processus passé en paramètre avec le client.

Processus bgwriter

- Processus d'écriture en tâche de fond des fichiers de données
- Existe aussi depuis la 8.3 un « wal writer », processus d'écriture en tâche de fond des journaux de transactions
- S'occupe de l'écriture des blocs modifiés en mémoire cache dans les fichiers de données
- Plusieurs paramètres indiquent la fréquence des écritures sur disque

Processus de stockage des statistiques

- Récupère les statistiques sur le nombre de lignes et le nombre de blocs affectés par les opérations INSERT/UPDATE/DELETE
- Récupère les dates et heures des VACUUM manuels et automatiques
- Activable via le paramètre track_activities
- Discute avec les autres processus via un port UDP

Ce port est configuré en mode non bloquant, ce qui fait que certaines statistiques peuvent être « oubliées » en cas de très forte activité.

- Données enregistrées dans le fichier global/pgstat.stat

Processus autovacuum

- autovacuum launcher : exécuté en permanence pour lancer à intervalle régulier un processus autovacuum worker

L'intervalle dépend du paramètre `autovacuum_naptime`. En 8.3 toutes les bases de données sont traitées dans cet intervalle. Avant la 8.3, une seule base était traitée sur cet intervalle.

- autovacuum_worker : se connecte à une base, lance des `VACUUM` et des `ANALYZE`
- Désactivable via le paramètre `autovacuum`
- Peut utiliser `maintenance_work_mem` en cas d'exécution d'un `VACUUM`
- Plusieurs autovacuum worker peuvent être exécutés en même temps

Le nombre dépend du paramètre `autovacuum_max_workers`.

- Ils peuvent être exécutés sur la même base en même temps

Ils ne se bloqueront pas grâce à un bout de mémoire partagée entre tous les processus autovacuum dans lequel ils indiquent sur quelle table ils travaillent.

Processus pgarch

- Archive les journaux de transaction
- Activable avec le paramètre `archive_mode`

Processus de traitement des journaux applicatifs

- Réalise la rotation des journaux applicatifs
- Activable avec le paramètre logging_collector
- Réagit à deux signaux
- Signal SIGHUP pour lui demander de relire la configuration
- Signal SIGUSR1 pour demander une rotation du journal applicatif.

Processus postgres

- Chargé de la communication entre un client et le serveur
- Nombre maximum dépendant du paramètre `max_connections`
- Peut allouer `work_mem` par tri et hachage pour l'exécution d'une requête

Partie 7 : Migration

- Mises à jour mineures (8.3.5 > 8.3.6)
- Mise à jour majeures (8.2 > 8.3)
- Attention à la compatibilité descendante !

Cette dernière partie du module donne quelques recommandations essentielles concernant la mise à jour de PostgreSQL.

Nous vous encourageons à toujours utiliser la dernière version mineure de |PG|, quelle que soit la version majeure que vous avez installée.

Les versions majeures de PostgreSQL (8.0, 8.1, 8.2, etc.) apportent des nouvelles fonctionnalités. Actuellement, une nouvelle version majeure sort chaque année (2008 : 8.3, 2009 : 8.4, ...) Les versions majeures contiennent des modifications du format interne des tables systèmes et des fichiers de données. Ces changements sont souvent complexes, ce qui rend impossible la compatibilité descendante (i.e. un serveur PostgreSQL 8.4 ne peut pas lire les fichiers de données (le répertoire PGDATA) créés par un serveur PostgreSQL 8.0). Il faut donc effectuer un export/import des données à chaque mise à jour majeure.

Les mises à jour mineures sont composées uniquement de correctifs et patchs de sécurité. Tous les utilisateurs devraient effectuer une mise à jour à chaque sortie d'une version mineure.

Bien sûr, toute mise à jour comporte une part de risque, cependant les versions mineures de PostgreSQL se contentent d'apporter des corrections pour réparer les bugs fréquents, renforcer la sécurité ou éviter les corruptions de données. Ainsi il est plus dangereux de ne pas mettre à jour PostgreSQL que d'effectuer les upgrades.

Mettre à jour une version mineure ne nécessite pas l'export/import des données (dumps). Il suffit d'arrêter le serveur, d'installer les nouveaux fichiers binaires, et de relancer le serveur. Dans certains cas, des changements manuels peuvent être nécessaires pour compléter la mise à jour, il faut donc lire attentivement les **notes de versions** avant toute mise à jour.

Pour être tenu au courant des nouvelles mises à jour, il suffit de s'inscrire sur la liste suivante :

http ://archives.postgresql.org/pgsql-announce/

Mises à jour mineure

- Une nouvelle version tous les 2-3 mois
- Pas de nouvelles fonctionnalités
  Consultez les notes de versions pour savoir si vous êtes directement concernés par un correctif :
  http ://docs.postgresql.fr/current/release.html
- **Obligatoire !**
  Il est moins risqué d'effectuer les mises à jour mineures que de ne rien faire.
- Simple et rapide
  La procédure de mise à jour est très simple :
  - ⇒ Arrêter le serveur
  - ⇒ Installer la nouvelle version
  - ⇒ Relancer le serveur

Mises à jour majeure

- Une nouvelle version tous les ans
- Nouvelles fonctionnalités !
  En général, les versions majeures apportent de nombreuses améliorations et de nouvelles possibilités.
  Dans le plupart des situations, migrer vers une version majeure est un bon choix en terme de performance et de stabilité.
  À l'occasion de chaque nouvelle version majeure, la communauté PostgreSQL organise une campagne de communication (annonces dans la presse, conférence, etc.) pour expliquer les apports de cette version.
- Import/Export des données
  La procédure de mise à jour majeure nécessite de sauvegarder les données dans un fichier dump SQL, puis de les recharger dans la nouvelle version :
  - ⇒ Sauvegarde des données
  - ⇒ Arrêter le serveur
  - ⇒ Installer la nouvelle version
  - ⇒ Relancer le serveur
  - ⇒ Restauration des données

  Ceci implique une coupure de service plus longue que pour les mises à jours mineures.
  Par ailleurs, cette opération est plus complexe lorsque le volume de données est conséquent (plusieurs To).
  L'utilisation de Slony permet de diminuer la durée de la coupure de service.

Migrer vers la 8.3

- Attention à la compatibilité ascendante !

  PostgreSQL 8.3 est une version nettement plus stricte que les versions précédentes.

  Il est possible qu'une application middleware fonctionne avec la version 8.2 mais ne soit pas compatible directement avec les versions supérieures.

- Disparition de certains transtypages automatiques
  Dans le but de maintenir les standards élevés d'intégrité et de fiabilité des données du projet PostgreSQL, un certain nombre de transtypages (« casts ») ont été supprimés. Ces modifications peuvent être à l'origine de problèmes lors de la mise à jour d'anciennes applications écrites sans tenir compte des comparaisons inter-types de données, en particulier avec les versions de PostgreSQL d'il y a plusieurs années.
  Les utilisateurs qui migrent de très vieilles applications, ou qui soupçonnent une application ou un code de procédure stockée permissif, doivent obligatoirement effectuer des tests complémentaires avant de mettre à jour les systèmes en production.
  Des gains de performance sont aussi à attendre de ce changement : la plupart des casts implicites qui avaient lieu entrainaient des problèmes de performance : l'ajout d'une fonction de cast implicite entraînait l'impossibilité de se servir de certains index. Ce changement, même s'il impose un travail supplémentaire, permet donc bien d'améliorer la qualité des applications.
- Contournement possible
  Si on ne souhaite pas corriger les codes applicatif, il est possible de "réactiver" les transtypages et ainsi obtenir un comportement compatible avec l'application :

  http ://petereisentraut.blogspot.com/2008/03/readding-implicit-casts-in-postgresql.html

Migrer vers la 8.3

- Comme toujours, attention à la compatibilité ascendante !

  PostgreSQL 8.4 a elle-aussi des incompatibilités avec les versions précédentes. Les notes de version en listent même 31.

  Avant toute migration, pensez à lire les notes de version de la 8.4 :

  http ://docs.postgresql.fr/8.4/release.html

- `pg_migrator`
  Cet outil est un projet d'EDB et de Bruce Momjian. Il a pour but de faciliter le passage de la version 8.3 à la version 8.4. Grâce à lui, il n'est plus nécessaire de faire les étapes pg_dump / initdb / pg_restore. Néanmoins, comme il s'agit de la première version de cet outil, faites des tests avant sur un serveur de tests.

  http ://pgfoundry.org/projects/pg-migrator/

Pour aller plus loin

Tuning PostgreSQL for performance

`http ://www.varlena.com/GeneralBits/Tidbits/perf.html`

Performance Tuning PostgreSQL

`http ://www.revsys.com/writings/postgresql-performance.html`

PostgreSQL hardware Performance Tuning

`http ://momjian.us/main/writings/pgsql/hw_performance/0.html`

Five Steps to PostgreSQL Performance

`http ://www.powerpostgresql.com/download/TFCKUpload/5.x-pdf`

Conclusion

- Les performances de PostgreSQL sont étroitement liées à celles du système et du matériel ;
- Avant de se lancer dans des optimisations logicielles, il faut vérifier si des axes d'optimisations matérielles sont possibles ;
- Certains choix (technologie utilisée pour les disques, la distribution utilisée, etc.) sont difficilement réversibles. Il faut prendre le temps d'étudier les meilleures options **avant** de construire un serveur PostgreSQL.

Questions

N’hésitez pas, c’est le moment !