Module J1

Configuration du système et de l’instance

24.04

Table des matières

Sur ce document

Formation	Module J1
Titre	Configuration du système et de l'instance
Révision	24.04
PDF	https://dali.bo/j1_pdf
EPUB	https://dali.bo/j1_epub
HTML	https://dali.bo/j1_html
Slides	https://dali.bo/j1_slides
TP	https://dali.bo/j1_tp
TP (solutions)	https://dali.bo/j1_solutions

Vous trouverez en ligne les différentes versions complètes de ce document.

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com[1] !

À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur https://dalibo.com/formations

[1] mailto:formation@dalibo.com

Remerciements

Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment :

Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées[3] par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

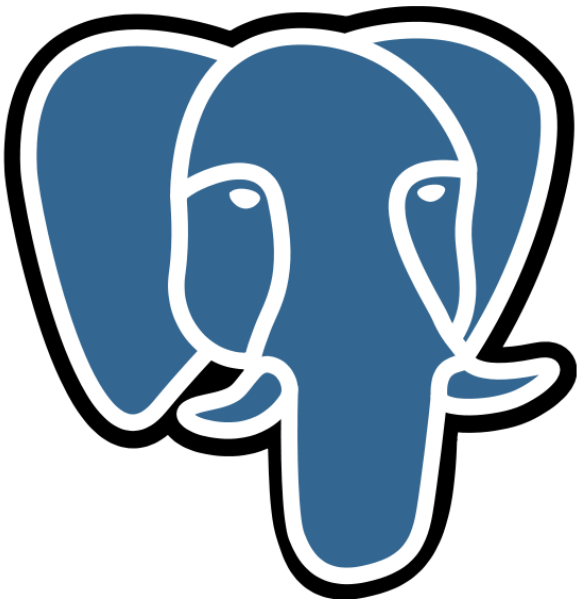
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

[3] https://www.postgresql.org/about/policies/trademarks/

1/ Configuration du système et de l’instance



1.1 INTRODUCTION

- L'optimisation doit porter sur les différents composants
  - le serveur qui héberge le SGBDR : le matériel, la distribution, le noyau, les systèmes de fichiers
  - 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 tentative d'optimisation. 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 cette partie 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 appelée *tuning*) doit donc être réalisée sur tous ces éléments **à la fois** pour être optimale !

1.1.1 Menu

- Quelques considérations générales sur l'optimisation
- Choix et configuration du matériel
- Choix et configuration du système d'exploitation
- Configuration du serveur de bases de données
- Outils

1.1.2 Considérations générales - 1

- 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.

1.1.3 Considérations générales - 2

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

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 10 utilisateurs en concurrence ne le sera plus pour 50 utilisateurs en concurrence. 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 souvent la tâche d'inspection plus rapide. Ainsi, l'identification du ou des paramètres à modifier, ou plus généralement des actions à 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 au fur et à mesure de la vie du serveur.

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

1.2 MATÉRIEL

- Performances très liées aux possibilités du matériel et de l'OS
- 4 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.).

1.2.1 CPU

- Trois critères importants
  - nombre de cœurs
  - fréquence
  - cache
- Privilégier
  - le nombre de cœurs si le nombre de sessions parallèles est important
  - ou la fréquence si les requêtes sont complexes
- 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.

Ces processus ne sont pas multi-threadés. Par conséquent, chaque requête exécutée est généralement traitée par un seul processus, sur un cœur de processeur. Mais dans certains cas, d'autres processus peuvent intervenir sur la même requête pour utiliser d'autres cœurs. Lorsque la requête est en lecture seule (et dans quelques rares cas en écriture) et que la parallélisation est activée, ce processus peut être aidé le temps de l'exécution de certains nœuds par un ou plusieurs processus appelés *workers*. Les détails figurent plus loin.

Parallélisation mise à part, plus vous voulez pouvoir exécuter de requêtes en simultané, plus vous devez avoir de processeurs (ou plus exactement de cœurs). On considère habituellement qu'un cœur peut traiter de 1 à 20 requêtes simultanément. Cela dépend notamment beaucoup des requêtes, de leur complexité, de la quantité de données 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'une instance pour une application web, il y a de fortes chances que le nombre de requêtes (simples) en parallèle soit assez élevé. Dans ce contexte, il faut prévoir un grand nombre de cœurs ou de processeurs. Par contre, sur un entrepôt de données, il y a généralement peu d'utilisateurs, mais des requêtes complexes et gourmandes en ressources, sur de gros jeux de données, mais ces requêtes sont, à priori, facilement parallélisables. Il est alors possible d'opter pour des processeurs avec une fréquence plus élevée (qui ont souvent moins de cœurs), mais plus le système aura de cœurs, plus sa capacité à pouvoir paralléliser les requêtes qui s'y prêtent sera élevé. Ainsi, la fréquence (et donc la puissance) des processeurs est un point important à considérer. Il peut faire la différence pour des requêtes 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 allers-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 fréquence du processeur.

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 performants 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.

1.2.2 RAM

- Essentielle pour un serveur de bases de données
- Plus il y en a, mieux c'est
  - moins d'accès disque
- 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 liée 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.

1.2.3 Disques

Technologie	Temps d'accès	Débit en lecture
RAM	~ 1 ns	~ 5 Go/s
NVMe	~ 100 µs	~ 3 Go/s
SSD (SATA)	~ 100 µs	~ 300 Mo/s
HDD SAS 15ktpm	~ 1 ms	~ 100 Mo/s
HDD SATA	~ 5 ms	~ 100 Mo/s

Les chiffres ci-dessus ne sont que des ordres de grandeurs : la technologie évolue constamment.

Il existe actuellement quatre types de modèles de disques :

- disques magnétiques SATA, dont la principale qualité est d'être peu cher ;
- disques magnétiques SAS : rapides, fiables, mais chers ;
- disques SSD : très rapides en temps d'accès, chers, mais aux prix en baisse ;

- NVMe : très rapide, très cher, nécessite une interface spécialisée.

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 10 000 en performance de temps d'accès entre les deux ! À l'autre bout de l'échelle se trouvent les disques magnétiques avec interface SATA. Leur faible performance en temps d'accès ne doit pas pour autant les disqualifier : leur prix est 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 consommables et de toujours avoir des disques de secours prêts à remplacer une défaillance.

Il est souvent préconisé de se tourner vers des disques SAS (*Serial Attached 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 permettent 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 à des endroits différents 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 des disques SSD (et encore plus NVMe) est un temps d'accès très rapide. Ils se démarquent des disques magnétiques (SAS comme 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.

Il y a quelques années, leur durée de vie était courte par rapport aux disques magnétiques dus essentiellement à la notion de TBW (*Terabytes written*), soit la quantité pouvant être écrite sur le SSD au cours de sa vie, puisque chaque zone mémoire du disque a un nombre maximal de cycles d'écriture ou d'effacement.

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, voire plus importantes, que celles des disques magnétiques. Néanmoins, ces mêmes algorithmes mettent en péril la durabilité des données en cas d'interruption brutale.

Tous les disques ne se valent pas, il y a des gammes pour « grand public » et des gammes « entreprise ». Choisissez toujours des disques de la gamme entreprise qui ont une meilleure durabilité et fournissent des fonctionnalités bien supérieures aux disques non professionnels.

Sur le marché du SSD, il existe plusieurs technologies (eMLC, iSLC, QLC…), certains auront de meilleures performances en lecture, d'autres en écriture, d'autre encore une meilleure durée de vie en écriture. Il est donc important de bien lire la documentation technique des disques avant leur achat.

Certains disques SSD haut de gamme ont une interface en SAS 12 Gbit/s, permettant d'atteindre des débits en lecture très élevés (de l'ordre de 1,5 Go/s) mais leur prix limite leur utilisation (de l'ordre de 10 000 € pour un disque de 3,8 To, fin 2020). À titre de comparaison, l'interface SATA troisième

génération[1] a un débit théorique de 6 Gbits/s, soit environ 750 Mo/s. Ce qui est donc deux fois moins rapide, même avec un SSD.

Il existe aussi des supports de stockage moins courants, encore onéreux, mais extrêmement rapides : ce sont les cartes basées sur la technologie NVMe (comme par exemple ceux commercialisés par Fusion-IO). Il s'agit de stockage en mémoire Flash sur support PCIe pouvant aller au-delà de 6 To en volume de stockage, avec des temps d'accès et des débits bien supérieurs aux SSD. On évoque des temps d'accès environ dix fois inférieurs et des débits presque dix fois supérieur à ce que l'on peut avoir sur une interface SATA standard. Leur utilisation reste cependant encore limitée en raison du coût de cette technologie.

Les disques bas de gamme mais rapides peuvent néanmoins 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 système d'exploitation pour optimiser l'utilisation des SSD. Par exemple sous Linux, les deux optimisations courantes dans le noyau sont :

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

1.2.4 RAID

- Pour un SGBD :
  - RAID 1 : système, journaux de transactions
  - RAID 10 : fichiers de données
- RAID 5 déconseillé pour les bases de données (écritures)
- RAID *soft* déconseillé !
- Qualité des cartes !
- Cache :
  - en lecture : toujours
  - en écriture : **si batterie** présente & supervisée

Il existe différents niveaux de RAID[2]. Le plus connu est le RAID 5, qui autorise de perdre un des disques sans interrompre le service, au prix d'une perte de capacité plus faible que le RAID 1 (disques redondants en miroir). Cependant, le RAID 5 est plutôt déconseillé pour les bases de données (PostgreSQL comme les autres) en raison de mauvaises performances en écriture, en temps normal et encore plus lors de la reconstruction d'un disque.

[1] https://sata-io.org/developers/sata-naming-guidelines
[2] https://fr.wikipedia.org/wiki/RAID_(informatique)

Pour les performances en écriture, il est généralement préférable de se baser sur du RAID 10, soit deux grappes de disques en RAID 1 (en miroir) agrégés dans un RAID 0 : c'est tout aussi intéressant en termes de fiabilité, mais avec de bien meilleures performances en lecture et écriture. En contrepartie, à volumétrie égale, il nécessite plus de disques et est donc beaucoup plus cher que le RAID 5. Pour réduire le budget, il peut être envisageable de choisir des disques SATA en RAID 10. Cela dit, un RAID 5 peut très bien fonctionner pour votre application. Le plus important est d'obtenir un RAID fiable avec de nombreux disques, surtout s'ils sont magnétiques.

Lors du choix du RAID, il est impératif de suivre les recommandations du constructeur par rapport à la compatibilité de la taille des disques et aux performances des différents modes de RAID. En effet, certains constructeurs déconseillent tel ou tel niveau de RAID par rapport à la capacité des disques ; par exemple, suivant l'algorithme implémenté, certains constructeurs conseilleront un RAID 6 plutôt qu'un RAID 5 si la capacité du disque dépasse une certaine taille. Ceci est justifié par exemple par une reconstruction plus rapide.

Lors de l'utilisation d'un RAID, il est important de prévoir un disque de *hot spare*. Il permet au système de reconstruire le RAID automatiquement et sans intervention humaine. Cela réduit la période pendant laquelle la grappe RAID est dans un mode dégradé.

Il est à noter que le système et les journaux de transactions n'ont pas besoin de RAID 10. Il y a peu de lectures, ils peuvent se satisfaire d'un simple RAID 1.

Le RAID 0 (simple addition de disques pour maximiser l'espace, sans aucune redondance) est évidemment à proscrire.

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. En lecture, il faut évidemment toujours l'activer.

Par contre, la carte RAID **doit** posséder une batterie (ou équivalent) pour utiliser le cache en écriture : les données du cache ne doivent pas disparaître en cas de coupure de courant. Ceci est obligatoire pour des raisons de fiabilité du service. La majorité des cartes RAID permettent de superviser l'état de la batterie et désactivent le cache en écriture par mesure de sécurité si la batterie est défaillante.

Pensez donc à toujours superviser l'état de vos contrôleurs RAID et de vos disques.

Le RAID *soft*, intégré à l'OS, qui gère directement les disques, a l'avantage d'un coût nul. Il est cependant déconseillé sur un serveur de production : les performances peuvent souffrir du partage du CPU avec les applications, surtout en RAID 5 ; une reconstruction d'un disque passe par le CPU et ralentit énormément la machine ; et surtout la fiabilité est impactée par l'impossibilité de rajouter une batterie.

1.2.5 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
  - NFS : risques de corruptions et problèmes de performance

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 termes de performances que s'il s'agit bien de disques physiquement différents.

De même, certaines grappes de disques (eg. *RAID GROUP*) accueillent trop de volumes logiques pour de multiples 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.

Le débit n'est pas tout !
Les SAN ne sont pas attachés directement au serveur. L'accès aux données accusera donc en plus une pénalité due à la latence réseau ! L'architecture et les équipements choisis doivent donc prévoir de multiples chemins entre serveur et baie, pour mener à une latence la plus faible possible, surtout pour la partition des journaux de transaction.

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 (*snapshots*, réplication, virtualisation…), en performances et en souplesse.

Dans un registre moins coûteux, la tentation est grande d'utiliser un simple NAS[3], avec par exemple un accès NFS aux partitions. Il faut l'éviter, pour des raisons de performance et de fiabilité. Utilisez plutôt iSCSI, peu performant, mais plus fiable et moins complexe.

1.2.6 Virtualisation

- Masque les ressources physiques au système
  - plus difficile d'optimiser les performances
- 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 (*steal time*)
- Mutualise les disques = problèmes de performances
  - préférer attachement direct au SAN
  - 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, et très courant, 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 diagnostic 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,

[3] https://fr.wikipedia.org/wiki/Serveur_de_stockage_en_r%C3%A9seau

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 4 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 est réputé offrir 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 sûrs et performants.

1.2.7 Virtualisation : les bonnes pratiques

- Éviter le *time drift* : même source NTP sur les VM et l'ESXi
- Utiliser les adaptateurs réseau paravirtualisés de type VMXNET3
- Utiliser l'adaptateur paravirtualisé PVSCSI pour les disques dédiés aux partitions PostgreSQL
- Si architecture matérielle 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 tops d'horloge pour une simple seconde. Si 100 tops d'horloge sont perdus, alors 1100 tops d'horloge seront délivrés avant que la VM ne considère qu'une seconde soit passée. C'est ce qu'on appelle le *time drift*.

Il est recommandé 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 paravirtualisé de type VMXNET3 pour avoir les meilleures 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 socket dispose de sa propre mémoire. Par exemple, avec un système à 2 sockets et 128 Go de RAM, chaque socket ou nœud possède 64 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.

Pour approfondir : Fiche KB préconisations pour VMWare[4]

[4] https://kb.dalibo.com/vmware

1.3 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.

1.3.1 Choix du système d'exploitation

- PostgreSQL fonctionne sur différents systèmes
  - principalement développé et testé sous Linux
  - *BSD, macOS, Windows, Solaris, etc.
- Windows intéressant pour les postes des développeurs
  - mais moins performant que Linux
  - moins d'outillage

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. La liste des plate-formes officiellement supportées[5]) comprend donc Linux, FreeBSD, OpenBSD, macOS, Windows, Solaris, etc. Cette portabilité est vérifiée en permanence avec la ferme de construction (*BuildFarm*[6]), qui comprend même de vieilles versions de ces systèmes sur différentes architectures avec plusieurs compilateurs.

Cela étant dit, il est malgré tout principalement développé sous Linux et la majorité des utilisateurs, et surtout des développeurs, 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

[5] https://www.postgresql.org/docs/current/supported-platforms.html
[6] https://buildfarm.postgresql.org/cgi-bin/show_status.pl

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 Red Hat (et ses dérivés CentOS, Rocky Linux…) et Debian (et ses dérivés, notamment Ubuntu). Sauf exception, nous ne traiterons plus ici que de Linux.

Un autre système souvent utilisé est Windows. PostgreSQL est beaucoup moins performant lorsqu'il est installé sur ce dernier que sur 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.

Un autre problème connu avec les instances PostgreSQL sous Windows est lié à l'architecture multiprocessus, où chaque connexion à l'instance crée un processus. Avant Windows 2016, plus de 125 connexions simultanées peuvent mener à l'épuisement de la *Desktop Heap Memory*, réduite pour les services non interactifs, et à de surprenants problèmes de mémoire (message *Out of memory* dans les traces PostgreSQL et/ou les événements de Windows). Pour les détails, voir la KB295902 de Microsoft[7], cet article Technet[8] et le wiki PostgreSQL[9].

Toujours sous Windows, il est fortement recommandé de laisser le paramètre `update_process_title` à `off` (c'est le défaut sous Windows, pas sous Linux). Le nom des processus ne sera plus dynamique, mais cela est trop lourd sous Windows. Le wiki ci-dessus pointe d'autres particularités et problèmes.

1.3.2 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.

[7] http://support.microsoft.com/kb/184802
[8] https://techcommunity.microsoft.com/t5/ask-the-performance-team/sessions-desktops-and-windows-stations/bap/372473
[9] https://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

Ce sont aussi celles qui proposent le plus de fonctionnalités intéressantes, comme la gestion complète du système de fichiers ext4, les *control groups*, une supervision avancée (avec `perf` et `bpf`), 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 termes de performances. Mais ce plus est difficilement quantifiable et est assorti d'un gros inconvénient : avoir à gérer soi-même les mises à jour, la recompilation en cas d'oubli d'un pilote, etc.

1.3.3 Configuration du noyau

- À configurer :
  - cache disque système
  - *swap*
  - *overcommit*
  - *huge pages*
  - affinité entre cœurs et mémoire
  - scheduler

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

1.3.4 Contrôle du cache disque système

- Lissage de l'écriture des *dirty pages*
- Paramètres
  - `vm.dirty_ratio` et `vm.dirty_background_ratio`
  - ou : `vm.dirty_bytes` et `vm.dirty_background_bytes`
- Encore utiles malgré les paramètres PostgreSQL `*_flush_after`

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 noyau `vm.dirty_ratio`, `vm.dirty_background_ratio`, `vm.dirty_bytes` et `vm.dirty_background_bytes`.

`vm.dirty_ratio` exprime le pourcentage de pages mémoire 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 mémoire 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`. Celui-ci est utilisé par PostgreSQL lors des *checkpoints*, ce qui peut provoquer des latences supplémentaires à ces moments-là.

Sur les versions de PostgreSQL précédant la 9.6, pour réduire les conséquences de ce phénomène, il était systématiquement conseillé d'abaisser `vm.dirty_ratio` à 10 et `vm.dirty_background_ratio` à 5. Ainsi, lors de fortes charges en écriture, le noyau reporte plus fréquemment son cache disque sur l'espace de stockage, mais pour une volumétrie plus faible, et l'encombrement de la bande passante vers les disques est moins long 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. Avec 32 Go de RAM, cela donne encore 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 représente une volumétrie trop importante (par exemple, 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 octets 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, augmente.

Depuis la version 9.6, ces options sont moins nécessaires grâce à ces paramètres propres à PostgreSQL :

- `bgwriter_flush_after` (512 ko par défaut) : lorsque plus de `bgwriter_flush_after` octets sont écrits sur disque par le *background writer*, le moteur tente de forcer la synchronisation sur disque ;
- `backend_flush_after` (désactivé par défaut) : force la synchronisation sur disque lorsqu'un processus a écrit plus de `backend_flush_after` octets ; il est préférable d'éviter ce comportement, c'est pourquoi la valeur par défaut est 0 (désactivation) ;
- `wal_writer_flush_after` (1 Mo par défaut) : quantité de données à partir de laquelle le *wal writer* synchronise les blocs sur disque ;
- `checkpoint_flush_after` (256 ko par défaut) : lorsque plus de `checkpoint_flush_after` octets sont écrits sur disque lors d'un checkpoint, le moteur tente de forcer la synchronisation sur disque.

Mais ces derniers paramètres ne concernent que les processus de PostgreSQL. Or PostgreSQL n'est

pas seul à écrire de gros fichiers (exports `pg_dump`, processus d'archivage, copies de fichiers…). Le paramétrage des `vm.dirty_bytes` conserve donc un intérêt.

1.3.5 Configuration du swap

La mémoire sert de cache au disque, pas l'inverse !

- *Swap* : pas plus de 2 Go
- et à décourager :

```
vm.swappiness = 10
```

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.

Il est utile d'en conserver un peu pour swapper des processus inactifs, ou le contenu de systèmes de fichiers `tmpfs` (classiquement, `/var/run`) et les journaux de `systemd-journald` selon la configuration de celui-ci. Ne pas avoir de *swap* amène encore 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 au plus 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 (30 ou 60 suivant les distributions) doit donc être abaissée à 10 pour éviter l'utilisation du *swap* dans la majorité des cas.

1.3.6 Configuration de la sur-réservation mémoire

Le noyau peut autoriser trop de réservation mémoire.

- Si saturation :
  - `kill -9` du processus par l'OOM killer
  - et donc redémarrage
  - purge du cache
- Désactiver sur serveur dédié :

```
vm.overcommit_memory = 2
vm.overcommit_ratio  = ?  # à calculer, souvent 70-80
```

Danger de l'overcommit :

Certaines applications réservent (*commit*) auprès du noyau 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 réserver 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*. Ce peut être intéressant dans certains cas d'utilisation, mais peut devenir dangereux dans le cadre d'un serveur PostgreSQL dédié, qui va réellement allouer (utiliser) toute la mémoire qu'il réservera. Typiquement, cela arrive lors de tris en mémoire trop gros, ou trop nombreux au même moment.

Quand le noyau arrive réellement à court de mémoire, il décide de tuer certains processus en fonction de leur impact sur le système (mécanisme de l'*Out Of Memory Killer*). Il est alors fort probable que ce soit un processus PostgreSQL qui soit tué. Il y a un risque de corruption de la mémoire partagée, donc par précaution toutes les transactions en cours sont annulées, et toute l'instance redémarre. Une perte de données est parfois possible en fonction de la configuration de PostgreSQL. Une corruption est par contre plutôt exclue.
De plus, le cache disque aura été purgé à cause de la consommation mémoire. Pire : le *swap* aura pu être rempli, entraînant un ralentissement général (*swap storm*) avant le redémarrage de l'instance.

Configuration de vm.overcommit_memory et vm.overcommit_ratio :

Il est possible de parer à ces problèmes grâce aux paramètres kernel `vm.overcommit_memory` et `vm.overcommit_ratio` du fichier `/etc/sysctl.conf` (ou d'un fichier dans `/etc/sysctl.conf.d/`). Cela suppose que le serveur est dédié exclusivement à PostgreSQL, car d'autres applications ont besoin d'un *overcommit* laxiste. Le swap devra avoir été découragé comme évoqué ci-dessus.

Pour désactiver complètement l'*overcommit memory* :

```
vm.overcommit_memory = 2
```

La taille maximum de mémoire réservable par les applications se calcule alors grâce à la formule suivante :

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

Ce `CommitLimit` ne doit pas dépasser 80 % de la RAM physiquement présente pour en préserver 20 % pour l'OS et son cache.

`vm.overcommit_ratio` est un pourcentage, entre 0 et 100. Or, sa valeur par défaut `vm.overcommit_ratio` est 50 : sur un système avec 32 Go de mémoire et 2 Go de *swap*, nous n'aurions alors que 32×50/100+2 = 18 Go de mémoire allouable ! Il faut donc monter cette valeur :

```
vm.overcommit_ratio = 75
```

nous obtenons 32×75/100+2 = 26 Go de mémoire utilisable par les applications sur les 32 Go disponibles. 6 Go serviront pour le cache et l'OS (bien sûr, ce pourra être plus quand les processus PostgreSQL utiliseront moins de mémoire).

Les valeurs typiques de `vm.overcommit_ratio`, sur des machines correctement dotées en RAM, et avec un *swap* de 2 Go au plus, vont de 70 à 80 (toujours en vue de réserver 20 % de RAM au cache disque, ce pourrait être un peu moins).

(Alternativement, il existe un paramètre exprimé en kilooctets, `vm.overcommit_kbytes`, mais il faut penser à l'adapter lors d'un ajout de RAM).

La prise en compte des fichiers de configuration modifiés se fait avec :

```
$ sudo sysctl --system
```

Exemple :

Une machine de 16 Go de RAM, 1,6 Go de *swap* possède cette configuration :

```
$ sysctl -a --pattern 'vm.overcommit.*'
vm.overcommit_memory = 2
vm.overcommit_ratio  = 85
```

Extrait de la configuration mémoire résultante :

```
$ free -m
             total       used       free     shared    buffers     cached
Mem:         16087      15914        173          0         65      13194
-/+ buffers/cache:       2653      13433
Swap:         1699          0       1699

dalibo@srv-psql-02:~$ cat /proc/meminfo
MemTotal:       16473548 kB
MemFree:          178432 kB
Buffers:           67260 kB
…
SwapTotal:       1740796 kB
```

```
SwapFree:        1740696 kB
…
CommitLimit:    15743308 kB
Committed_AS:    6436004 kB
…
```

Le `CommitLimit` atteint 15 Go, laissant au cache et l'OS une portion très réduite, mais évitant au moins un crash de l'instance. Ici, `Committed_AS` (valeur totale réservée à ce moment), est très en deça.

Avec des *huge pages* :

Les choses se compliquent si l'on paramètre des *huge pages* (voir plus bas).

Exemple de saturation mémoire :

Avec la désactivation de la sur-allocation, l'instance ne plantera plus par défaut de mémoire. Les requêtes demandant trop de mémoire se verront refuser par le noyau une nouvelle réservation, et elles tomberont simplement en erreur. Cela peut se tester ainsi :

```
SET work_mem = '1000GB' ;
-- DANGEREUX ! Tri de 250 Go en RAM !
EXPLAIN (ANALYZE) SELECT i FROM generate_series (1,3e9) i
ORDER BY i DESC ;
```

Si le paramétrage ci-dessus a été appliqué, on obtiendra ceci dans la session :

```
postgres=# EXPLAIN (ANALYZE) SELECT i FROM generate_series (1,3e9) i
postgres-# ORDER BY i DESC ;
ERROR:  out of memory
DETAIL:  Failed on request of size 23 in memory context "ExecutorState".
```

(les traces seront plus verbeuses). La session et l'instance fonctionnent ensuite normalement.

Sans paramétrage, ce serait plus brutal :

```
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
```

et les traces indiqueraient ceci avant le redémarrage :

```
LOG:  server process (PID 2429) was terminated by signal 9: Killed
```

Noter qu'une requête qui tombe en erreur n'est pas forcément celle qui a consommé le plus de mémoire ; elle est juste celle qui a atteint la première le `CommitLimit`.

Pour plus de détails :

- https://kb.dalibo.com/overcommit_memory
- https://www.kernel.org/doc/html/latest/mm/overcommit-accounting.html
- https://www.kernel.org/doc/html/latest/filesystems/proc.html?highlight=meminfo

1.3.7 Huge pages

Pages mémoire de 2 Mo au lieu de 4 ko :

- Les processus consomment moins de mémoire
- *shared buffers* non swappés
- PostgreSQL :

```
huge_pages = try  # ou: on / off
```

- Noyau :

```
vm.nr_overcommit_hugepages = ?  # selon shared_buffers +10%
vm.overcommit_ratio        = ?  # à baisser
```

- *Transparent Huge Pages* : à désactiver

Principe des *huge pages* :

Les systèmes d'exploitation utilisent un système de mémoire virtuelle : chaque contexte d'exécution (comme un processus) utilise un plan d'adressage virtuel, et c'est le processeur qui s'occupe de réaliser la correspondance entre l'adressage virtuel et l'adressage réel. Chaque processus fournit donc la correspondance entre les deux plans d'adressage, dans ce qu'on appelle une « table de pagination ». Les processeurs modernes permettent d'utiliser plusieurs tailles de page mémoire simultanément. Pour les processeurs Intel/AMD, les tailles de page possibles sont 4 ko, 2 Mo et 1 Go.

Les pages de 4 ko sont les plus souples, car offrant une granularité plus fine. Toutefois, pour des grandes zones mémoire contigües, il est plus économique d'utiliser des tailles de pages plus élevées. Par exemple, il faudra 262 144 entrées pour 1 Go de mémoire avec des pages de 4 ko, contre 512 entrées pour des pages de 2 Mo.

Or, chaque processus PostgreSQL dispose de sa propre table de pagination, qui va gonfler au fur et à mesure que ce processus va accéder à différents blocs des *shared buffers* : pour des *shared buffers* de 8 Go, chaque processus peut gaspiller 16 Mo si les pages font 4 ko, contre une centaine de ko pour des pages de 2 Mo. Une ligne de `/proc/meminfo` indique la mémoire utilisée par les TLB :

```
PageTables:      1193040 kB
```

Sur des petites configurations (quelques Go de RAM et peu de connexions), cela n'a pas beaucoup d'importance. Sinon, cette mémoire pourrait être utilisée à meilleur escient (`work_mem` par exemple, ou tout simplement du cache système).

Paramétrer PostgreSQL pour les *huge pages* :

Dans `postgresql.conf`, le défaut convient :

```
huge_pages = try
```

PostgreSQL se rabattra sur des pages de 4 ko si le système n'arrive pas à fournir les pages de 2 Mo. La valeur `on` permet de refuser le démarrage si les *huge pages* demandées ne sont pas disponibles (mauvais paramétrage, fragmentation mémoire…).

À partir de la version 14, il est possible de surcharger la configuration système de la taille des *huge pages* avec le paramètre `huge_page_size`. Par défaut, PostgreSQL utilisera la valeur du système d'exploitation.

Paramétrer les *huge pages* au niveau noyau :

On se limitera ici aux *huge pages* les plus courantes, celles de 2 Mo (à vérifier sur la ligne `Hugepagesize` de `/proc/meminfo`).

Dans `/etc/sysctl.d/`, définir le nombre de *huge pages* `vm.nr_overcommit_hugepages` : La valeur de ce paramètre est en pages de la taille de *huge page* par défaut. Il doit être suffisamment grand pour contenir les *shared buffers* et les autres zones mémoire partagées (tableau de verrous, etc.). Compter 10 % de plus que ce qui est défini pour `shared_buffers` devrait être suffisant, mais il n'est pas interdit de mettre des valeurs supérieures, puisque Linux créera avec ce système les *huge pages* à la volée (et les détruira à l'extinction de PostgreSQL). Une alternative est le paramètre `vm.nr_hugepages` qui crée des pages statiques (et dont l'utilisation serait obligatoire pour des pages de 1 Go).

En conséquence, si `shared_buffers` vaut 8 Go :

```
# HP dynamiques
vm.nr_overcommit_hugepages=4505    # 8192 / 2 * 1.10
# HP statiques
vm.nr_hugepages=0
```

NB : Sur un système hébergeant plusieurs instances, il faudra additionner toutes les zones mémoire de toutes les instances.

Un outil pratique pour gérer les *huge pages* est `hugeadm`.

Si l'on a paramétré la sur-allocation mémoire comme décrit ci-dessus, le calcul change, car les *huge pages* n'entrent pas dans la `CommitLimit`. On a alors :

```
CommitLimit = ( taille RAM - HugePages_Total*Hugepagesize ) * overcommit_ratio/100
              + taille swap
```

Les valeurs de `vm.overcommit_ratio` sont alors typiquement entre 60 et 72 (pour préserver 20% du cache, si le cache est réduit).

Pour plus de détails : https://kb.dalibo.com/huge_pages

Désactivation des Transparent Huge Pages

Dans `/proc/meminfo`, la ligne `AnonHugePages` indique des *huge pages* allouées par le mécanisme de *Transparent Huge Pages*: le noyau Linux a détecté une allocation contiguë de mémoire et l'a convertie en *huge pages*, indépendamment du mécanisme décrit plus haut. Hélas, les THP ne s'appliquent pas à la mémoire partagée de PostgreSQL.

Les THP sont même contre-productives sur une base de données, à cause de la latence engendrée par la réorganisation par le système d'exploitation. Comme les THP sont activées par défaut, il faut les désactiver au boot via `/etc/crontab` :

```
@reboot  root  echo never > /sys/kernel/mm/transparent_hugepage/enabled
@reboot  root  echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

ou encore dans la configuration de `grub` :

```
transparent_hugepage=never
```

Dans `/proc/meminfo`, la ligne `AnonHugePages` doit donc valoir 0.

1.3.8 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` : passer à 0

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 -i socket
Cœur(s) par socket :                   2
Socket(s) :                            1
```

Sur une architecture NUMA (*Non Uniform Memory Access*), il existe une notion de distance entre les sockets processeurs et les « zones » mémoire (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 mémoire 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 mémoire 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 là 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 on l'a vu, 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 moins important côté système. De plus, elles peuvent provoquer des variations aléatoires des performances en fonction du socket où un processus serveur est exécuté et des zones mémoire 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, durée : 20 h, CPU utilisé par le `COPY` : 3 à 5 %
- avec `zone_reclaim_mode` à 0, durée : 2 h, 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 :

- Linux memory zone reclaim[10]
- MySQL "swap insanity" problem[11]

1.3.9 Configuration de l'ordonnanceur

- Réduire la propension du kernel à migrer les processus
  - `kernel.sched_migration_cost_ns = 5000000` (×10) (si kernel < 5.13)
- Désactiver le regroupement par session TTY
  - `kernel.sched_autogroup_enabled = 0`

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

Depuis le noyau 2.6.23 l'ordonnanceur de tâches est le *CFS (Completely Fair Scheduler)*. Celui-ci est en charge de distribuer les ressources aux différents processus de manière équitable. Lorsqu'un processus est en exécution depuis plus de `kernel.sched_migration_cost_ns`, celui-ci peut être migré afin de laisser la place à un autre processus. Lorsque de nombreux processus demandent des ressources, la gestion de l'ordonnancement et la migration des processus peuvent devenir pénalisantes. Il est donc recommandé d'augmenter significativement cette valeur. Par exemple en la passant de 0,5 à 5 ms (5 000 000 ns). L'option disparaît cependant du noyau Linux 5.13 et suivants (donc de Rocky Linux 9 par exemple) et est remplacée par `/sys/kernel/debug/sched/migration_cost_ns`.

Par ailleurs, l'ordonnanceur regroupe les processus par session (TTY) afin d'avoir un meilleur temps de réponse « perçu ». Dans le cas de PostgreSQL, l'ensemble des processus sont lancés par une seule session TTY. Ces derniers seraient alors dans un même groupe et pourraient être privés de ressources (allouées pour d'autres sessions).

Sans regroupement de processus :

```
[proc PG. 1 | proc PG. 2 | proc PG. 3 | procPG . 4 | proc. 5 | proc. 6]
```

Avec regroupement de processus :

```
[proc PG. 1, 2, 3, 4  |     proc. 5       |     proc. 6      ]
```

Pour désactiver ce comportement, il faut passer le paramètre `kernel.sched_autogroup_enabled` à 0.

1.3.10 Comment les configurer

- Outil
  - `sysctl`
- Fichier de configuration
  - `/etc/sysctl.conf`
  - `/etc/sysctl.d/*conf`

Tous les paramètres expliqués ci-dessus sont à placer dans le fichier `/etc/sysctl.conf` ou dans le répertoire `/etc/sysctl.d/` (où tout fichier ayant l'extension `.conf` est lu et pris en compte). Il est ainsi préconisé d'y créer un ou plusieurs fichiers pour vos configurations spécifiques afin que ces dernières ne soient pas accidentellement écrasées lors d'une mise à jour système par exemple. À chaque redémarrage du serveur, Linux va récupérer le paramétrage et l'appliquer.

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

```
# sysctl --system
```

de consulter les valeurs avec :

```
# sysctl -a
```

et de modifier un paramètre précis (jusqu'au prochain redémarrage) :

```
# sysctl -w vm.swappiness=10
```

1.3.11 Choix du système de fichiers

- Windows :
  - NTFS
- Linux :
  - ext4, XFS
  - LVM pour la flexibilité
- Solaris :
  - ZFS
- Utiliser celui préconisé par votre système d'exploitation/distribution

Quel que 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. Les instances de production PostgreSQL utilisent de fait soit ext4, soit XFS, qui sont donc les systèmes de fichiers recommandés.

En 2016, un benchmark sur Linux de Tomas Vondra[12] de différents systèmes de fichiers montrait que ext4 et XFS ont des performantes équivalentes.

Autrefois réservé 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 (*snapshot*). OpenZFS[13], son portage sous Linux/FreeBSD, entre autres, est un système de fichiers proposant un panel impressionnant de fonctionnalités (dont : checksum, compression, gestion de snapshot), les performances en écriture sont cependant bien moins bonnes qu'avec ext4 ou XFS. De plus, il est plus complexe à mettre en place et à administrer. Btrfs[14] est relativement répandu et bien intégré

[12] https://fr.slideshare.net/fuzzycz/postgresql-na-ext4-xfs-btrfs-a-zfs-fosdem-pgday-2016
[13] https://en.wikipedia.org/wiki/OpenZFS
[14] https://en.wikipedia.org/wiki/Btrfs

à Linux, et offre une partie des fonctionnalités de ZFS ; mais il est également peu performant avec PostgreSQL.

LVM[15] permet de rassembler plusieurs partitions dans un même *Volume Group*, puis d'y tailler des partitions (*Logical Volumes*) qui seront autant de points de montage. LVM permet de changer les tailles des LV à volonté, d'ajouter ou supprimer des disques physiques à volonté dans les VG, ce qui simplifie l'administration au niveau PostgreSQL… De nos jours, l'impact en performance est négligeable pour la flexibilité apportée. Si l'on utilise les *snapshots* de LVM, il faudra vérifier l'impact sur les performances. LVM peut même gérer le RAID mais, dans l'idéal, il est préférable qu'une bonne carte RAID s'en charge en dessous.

NFS peut sembler intéressant, vu ses fonctionnalités : facilité de mise en œuvre, administration centralisée du stockage, mutualisation des espaces. Cependant, ce système de fichiers est source de nombreux problèmes avec PostgreSQL. Si la base tient en mémoire et que les latences possibles ne sont pas importantes, on peut éventuellement utiliser NFS. Il faut la garantie que les opérations sont synchrones. Si ce n'est pas le cas, une panne sur la baie peut entraîner une corruption des données. Au minimum, l'option `sync` doit être présente côté serveur et les options `hard`, `proto=tcp`, `noac` et `nointr` doivent être présentes côté client. Si vous souhaitez en apprendre plus sur le sujet des options pour NFS, un article détaillé est disponible dans la base de connaissances Dalibo[16], et la documentation de PostgreSQL à partir de la version 12[17].

Par contre, NFS est 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[18]. 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 ceux de fiabilité fréquents, et compliqués à diagnostiquer (comme illustré dans ce mail[19], où le problème venait du noyau Linux).

Sous Windows, la question ne se pose pas : NTFS est le seul système de fichiers assez stable. L'installeur fourni par EnterpriseDB dispose d'une protection qui empêche l'installation d'une instance PostgreSQL sur une partition VFAT.

[15] https://fr.wikipedia.org/wiki/Gestion_par_volumes_logiques
[16] https://kb.dalibo.com/nfs
[17] https://docs.postgresql.fr/current/creating-cluster.html#creating-cluster-nfs
[18] https://www.postgresql.org/message-id/4D2285CF.3050304@2ndquadrant.com
[19] https://www.postgresql.org/message-id/4D40DDB7.1010000@credativ.com

1.3.12 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` :

```
sudo 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 :

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

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

```
sudo e2fsck -D /dev/sda1
```

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

L'option `nobarrier` peut être utilisée, mais avec précaution. Cette dernière peut apporter une différence significative en termes de performance, mais elle met en péril vos données en cas de coupure

soudaine où les caches disques, RAID ou baies sont alors perdus. Cette option ne peut être utilisée qu'à la seule condition que tous ces différents caches soient sécurisés par une batterie.

1.3.13 Configuration de l'antivirus

Pas d'antivirus

PostgreSQL s'appuie sur le système d'exploitation et, pour l'intégrité des données, s'attend à ce qu'il effectue rigoureusement les opérations qu'il lui demande (écriture de fichiers notamment). Or les antivirus fonctionnent dans des couches très basses du système. L'interaction avec des antivirus a donc régulièrement mené à des problèmes de performance voire de corruption.

Nous déconseillons fortement d'installer un antivirus sur un serveur PostgreSQL.
Si vous devez absolument installer un antivirus, il faut impérativement exclure de son analyse tous les répertoires, fichiers et processus de PostgreSQL.

1.4 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, statistiques).

1.4.1 Version

- Chaque nouvelle version majeure a des améliorations de performance
  - mettre à jour est un bon moyen pour gagner en performances
- Ne pas compiler

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 (*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.

- La version 9.6 propose la parallélisation de l'exécution de certaines requêtes, et le nombre de nœuds concernés augmente à chaque version.
- Le partitionnement déclaratif (introduit en version 10) peut mener à manipuler beaucoup de tables-partitions. Le planificateur gère cela de mieux en mieux dans les dernières versions.

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.

Dans certains 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 » avec `gcc`. Il faut toutefois prendre deux éléments importants en compte avant de remplacer les binaires de PostgreSQL par des binaires recompilés avec `icc` :

- la taille des fichiers recompilés est nettement plus grande ;
- 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.

1.4.2 Configuration - mémoire partagée

- `shared_buffers = ...`
  - 25 % de la RAM en première intention
  - max 40 %
  - complémentaire du cache OS
- `wal_buffers`

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

shared_buffers :

`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 en mémoire partagée, et donc commun à tous les processus PostgreSQL. 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.

Pour dimensionner `shared_buffers` sur un serveur dédié à PostgreSQL, la documentation officielle[20] donne 25 % de la mémoire vive totale comme un bon point de départ et déconseille de dépasser 40 %, car le cache du système d'exploitation est aussi utilisé.

Sur une machine dédiée de 32 Go de RAM, cela donne donc :

```
shared_buffers = 8GB
```

Le défaut de 128 Mo n'est donc pas adapté à un serveur sur une machine récente.

Suivant les cas, une valeur inférieure ou supérieure à 25 % sera encore meilleure pour les performances, mais il faudra tester avec votre charge (en lecture, en écriture, et avec le bon nombre de clients).

Modifier `shared_buffers` impose de redémarrer l'instance.

Attention : une valeur élevée de `shared_buffers` (au-delà de 8 Go) nécessite de paramétrer finement le système d'exploitation (*Huge Pages* notamment) et d'autres paramètres comme `max_wal_size`, et de s'assurer qu'il restera de la mémoire pour le reste des opérations (tri…).

wal_buffers :

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 -1, ce qui correspond à un calcul automatique au démarrage de PostgreSQL. Avec les tailles de `shared_buffers` actuelles, il vaut généralement 16 Mo (la taille par défaut d'un segment du journal de transaction).

1.4.3 Configuration - mémoire des processus

- `work_mem`
  - par processus, voire nœud
  - valeur très dépendante de la charge et des requêtes
  - fichiers temporaires vs saturation RAM
- × `hash_mem_multiplier`
- `maintenance_work_mem`

Les processus de PostgreSQL ont accès à la mémoire partagée, définie principalement par `shared_buffers`, mais ils ont aussi leur mémoire propre. Cette mémoire n'est utilisable que par le processus l'ayant allouée.

Le paramètre le plus important est `work_mem`, qui définit la taille de la mémoire de travail d'un processus lors d'une requête, principalement lors d'opérations de tri : `ORDER BY`, certaines jointures, déduplication… Autre paramètre capital, `maintenance_work_mem` est la mémoire pour les opérations de maintenance lourdes : `VACUUM`, `CREATE INDEX`, ajouts de clé étrangère…

Cette mémoire est rendue immédiatement après la fin de l'ordre concerné.

Opérations de maintenance & maintenance_work_mem :

`maintenance_work_mem` peut être monté à 256 Mo à 1 Go sur les machines récentes, car il concerne des opérations lourdes rarement exécutées plusieurs fois simultanément. Monter au-delà est rare, mais peut avoir un intérêt dans les créations de très gros index.

Paramétrage de work_mem :

Pour `work_mem`, c'est beaucoup plus compliqué.

Si `work_mem` est trop bas, beaucoup d'opérations de tri, y compris nombre de jointures, ne s'effectueront pas en RAM. Par exemple, si une jointure par hachage impose d'utiliser 100 Mo en mémoire, mais que `work_mem` vaut 10 Mo, PostgreSQL écrira des dizaines de Mo sur disque à chaque appel de la jointure. Si, par contre, le paramètre `work_mem` vaut 120 Mo, aucune écriture n'aura lieu sur disque, ce qui accélérera forcément la requête.

Trop de fichiers temporaires peuvent ralentir les opérations, voire saturer le disque. Un `work_mem` trop bas peut aussi contraindre le planificateur à choisir des plans d'exécution moins optimaux.

Par contre, si `work_mem` est trop haut, et que trop de requêtes le consomment simultanément, le danger est de saturer la RAM. Il n'existe en effet pas de limite à la consommation des sessions de PostgreSQL, ni globalement ni par session !
Or le paramétrage de l'overcommit sous Linux est par défaut très permissif, le noyau ne bloquera rien. La première conséquence de la saturation de mémoire est l'assèchement du cache système (complémentaire de celui de PostgreSQL), et la dégradation des performances. Puis le système va se mettre à swapper, avec à la clé un ralentissement général et durable. Enfin le noyau, à court de mémoire, peut être amené à tuer un processus de PostgreSQL. Cela mène à l'arrêt de l'instance, ou plus fréquemment à son redémarrage brutal avec coupure de toutes les connexions et requêtes en cours.

Toutefois, si l'administrateur paramètre correctement l'overcommit[21], Linux refusera d'allouer la RAM et la requête tombera en erreur, mais le cache système sera préservé, et PostgreSQL ne tombera pas.

Suivant la complexité des requêtes, il est possible qu'un processus utilise plusieurs fois `work_mem` (par exemple si une requête fait une jointure et un tri, ou qu'un nœud est parallélisé). À l'inverse, beaucoup de requêtes ne nécessitent aucune mémoire de travail.

La valeur de `work_mem` dépend donc beaucoup de la mémoire disponible, des requêtes et du nombre de connexions actives.

Si le nombre de requêtes simultanées est important, `work_mem` devra être faible. Avec peu de requêtes simultanées, `work_mem` pourra être augmenté sans risque.

Il n'y a pas de formule de calcul miracle. Une première estimation courante, bien que très conservatrice, peut être :

> `work_mem` = mémoire / `max_connections`

On obtient alors, sur un serveur dédié avec 16 Go de RAM et 200 connexions autorisées :

```
work_mem = 80MB
```

Mais `max_connections` est fréquemment surdimensionné, et beaucoup de sessions sont inactives. `work_mem` est alors sous-dimensionné.

Plus finement, Christophe Pettus propose en première intention[22] :

> `work_mem` = 4 × mémoire libre / `max_connections`

Soit, pour une machine dédiée avec 16 Go de RAM, donc 4 Go de *shared buffers*, et 200 connections :

```
work_mem = 240MB
```

[21] https://dali.bo/j1_html#configuration-du-oom
[22] https://thebuild.com/blog/2023/03/13/everything-you-know-about-setting-work_mem-is-wrong/

Dans l'idéal, si l'on a le temps pour une étude, on montera `work_mem` jusqu'à voir disparaître l'essentiel des fichiers temporaires dans les traces, tout en restant loin de saturer la RAM lors des pics de charge.

En pratique, le défaut de 4 Mo est très conservateur, souvent insuffisant. Généralement, la valeur varie entre 10 et 100 Mo. Au-delà de 100 Mo, il y a souvent un problème ailleurs : des tris sur de trop gros volumes de données, une mémoire insuffisante, un manque d'index (utilisés pour les tris), etc. Des valeurs vraiment grandes ne sont valables que sur des systèmes d'infocentre.

Augmenter globalement la valeur du `work_mem` peut parfois mener à une consommation excessive de mémoire. Il est possible de ne la modifier que le temps d'une session pour les besoins d'une requête ou d'un traitement particulier :

```
SET work_mem TO '30MB' ;
```

hash_mem_multiplier :

À partir de PostgreSQL 13, un paramètre multiplicateur peut s'appliquer à certaines opérations particulières (le hachage, lors de jointures ou agrégations). Nommé `hash_mem_multiplier`, il vaut 1 par défaut en versions 13 et 14, et 2 à partir de la 15. `hash_mem_multiplier` permet de donner plus de RAM à ces opérations sans augmenter globalement `work_mem`.

Ajoutons qu'avant PostgreSQL 13, il y a parfois des problèmes dans les calculs d'agrégats : lorsque l'optimiseur sélectionne les nœuds d'exécution, il estime la mémoire à utiliser par la table de hachage Si l'estimation est supérieure à `work_mem`, il choisira plutôt un agrégat par tri. Si elle est inférieure, il passera par un agrégat par hachage. Il peut arriver que l'estimation soit mauvaise, et qu'il faille plus de mémoire : l'exécuteur continuera d'en allouer au-delà de `work_mem`. Selon la quantité et le paramétrage du serveur, cela peut passer inaperçu, mener à l'échec de la requête, interdire aux autres processus d'en allouer, voire provoquer un swap ou l'arrêt de l'instance. La version 13 corrige cela : lors d'un hachage, l'exécuteur ne se permet de consommer la mémoire qu'à hauteur de `work_mem` × `hash_mem_multiplier` (2 par défaut dès la version 15, 1 auparavant), puis se rabat sur le disque si cela reste insuffisant.

1.4.4 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 totale du cache disque disponible pour une requête. Pour le configurer, il faut prendre en compte le cache de PostgreSQL (`shared_buffers`) et celui du système

d'exploitation. Ce n'est donc pas une mémoire que PostgreSQL va allouer, mais plutôt une simple indication de ce qui est disponible. 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 789 116 Kio, résultat de l'addition de 190 580 (colonne `buffers`) et 598 536 (colonne `cached`). Il faut ensuite ajouter `shared_buffers` à cette valeur.

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 4 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 disques rapides, il ne faut pas hésiter à descendre un peu cette valeur (entre 2 et 3 par exemple). Si les données tiennent entièrement en cache ou sont stockées sur des disques SSD, il est même possible de descendre encore plus cette valeur.

1.4.5 Configuration - parallélisation : principe

- Par défaut : 1 requête = 1 processus
- Grosses tables : *parallel workers* en complément
- Nombreux nœuds parallélisables

Par défaut, une requête possède un seul processus dédié sur le serveur, qui par défaut n'utilise qu'un seul cœur. Pour répartir la charge des grosses requêtes sur plusieurs cœurs, un processus PostgreSQL peut se faire aider d'autres processus durant l'exécution de certains nœuds.

Les *parallel workers* se répartissent les lignes issues, par exemple, d'un parcours. Un nœud est dédié à la récupération des résultats (*gather*). Il est opéré par le processus principal qui peut, s'il n'a rien à faire, participer au traitement réalisé par les *parallel workers*.

La parallélisation peut se faire sur différentes parties d'une requête, comme un parcours de table ou d'index, une jointure ou un calcul d'agrégat.

La mise en place de la parallélisation a un coût. En conséquence, la parallélisation n'est possible sur un parcours que si la table ou l'index est suffisamment volumineux.

Le coût du transfert des lignes est aussi pris en compte. En conséquence, ce même parcours de table ou d'index ne sera pas forcément parallélisé s'il n'y a pas une clause de filtrage, par exemple.

En pratique, cette parallélisation n'a d'intérêt que si les performances sont contraintes d'abord par le CPU, et non par les disques.

1.4.6 Configuration - parallélisation : paramètres

- Nombre de *parallel workers* :
  - `max_parallel_workers_per_gather` (défaut : 2)
  - `max_parallel_workers` (8)
  - `max_worker_processes` (8)
- Taille minimale des tables/index :
  - `min_parallel_table_scan_size` (8 Mo)
  - `min_parallel_index_scan_size` (512 ko)
- Indexation et `VACUUM` :
  - `max_parallel_maintenance_workers` (2)

Paramètre principaux :

Le nombre maximum de processus utilisables pour un nœud d'exécution dépend de la valeur du paramètre `max_parallel_workers_per_gather` (à 2 par défaut). Ils ne seront lancés que si la requête le nécessite.

Si plusieurs processus veulent paralléliser l'exécution de leur requête au même moment, le nombre total de *workers* parallèles simultanés ne pourra pas dépasser la valeur du paramètre `max_parallel_workers` (8 par défaut).

Ce nombre ne peut lui-même dépasser la valeur du paramètre `max_worker_processes`, nombre de processus d'arrière-plan. (Avant PostgreSQL 10, le paramètre `max_parallel_workers` n'existait pas

et la limite se basait sur `max_worker_processes`.)

Impact de la volumétrie :

Le volume déclencheur dépend pour les tables de la valeur du paramètre `min_parallel_table_scan_size` (8 Mo par défaut) et de celle de `min_parallel_index_scan_size` (512 ko par défaut) pour les index. Ces paramètres sont rarement modifiés. Le moteur détermine ensuite ainsi le nombre de *workers* à lancer :

- Taille de la relation = T
- `min_parallel_table_scan_size` = S (dans le cas d'une table)
  - si T < S : pas de *worker*
  - si T ≥ S : on utilise 1 worker en plus du processus principal
  - si T ≥ S×3 : 2 workers en plus du processus principal
  - si T ≥ S×3×3 : 3 workers en plus du processus principal
  - si T ≥ S×3×3×3 : 4 workers en plus du processus principal
  - etc.

Si le processus ne peut lancer tous les *workers* qu'il a prévu, il poursuit sans message d'erreur avec ceux qu'il peut lancer.

Le coût induit par la mise en place du parallélisme est défini par `parallel_setup_cost` (1000 par défaut, rarement modifié).

Il faut se rappeler que le processus principal traite lui aussi des lignes, comme ses *parallel workers*. Il pourrait donc devenir un goulet d'étranglement. Le paramètre `parallel_leader_participation` peut alors être passé à `off` afin qu'il ne s'occupe plus que de récupérer et traiter le résultat des *workers*.

Exemple :

Le parcours suivant sur une table de 13 Go demande 7 *parallel workers* (mention *Planned*). Cela est possible car on a monté `max_parallel_workers_per_gather` au moins à 7. Seuls 4 *parallel workers* ont été accordés : le seuil de `max_parallel_workers` a dû être dépassé à cause d'autres requêtes. On note 5 boucles (*loops*) car le processus principal participe aussi au parcours.

```
EXPLAIN (ANALYZE, COSTS OFF)
SELECT * FROM pgbench_accounts
WHERE bid = 55  ;
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Gather (actual time=1837.490..2019.284 rows=100000 loops=1)
   Workers Planned: 7
   Workers Launched: 4
   ->  Parallel Seq Scan on pgbench_accounts (actual time=1849.780..1904.057
↪  rows=20000 loops=5)
         Filter: (bid = 55)
         Rows Removed by Filter: 19980000
 Planning Time: 0.038 ms
 Execution Time: 2024.043 ms
```

L'option `VERBOSE` donne plus de détails :

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Gather (actual time=1983.902..2124.019 rows=100000 loops=1)
   Output: aid, bid, abalance, filler
   Workers Planned: 7
   Workers Launched: 4
   ->  Parallel Seq Scan on public.pgbench_accounts (actual time=2001.592..2052.496
↪  rows=20000 loops=5)
         Output: aid, bid, abalance, filler
         Filter: (pgbench_accounts.bid = 55)
         Rows Removed by Filter: 19980000
         Worker 0:  actual time=1956.263..2047.370 rows=16893 loops=1
         Worker 1:  actual time=1957.269..2043.763 rows=62464 loops=1
         Worker 2:  actual time=2055.270..2055.271 rows=0 loops=1
         Worker 3:  actual time=2055.577..2055.577 rows=0 loops=1
 Query Identifier: 7891460439412068106
 Planning Time: 0.067 ms
 Execution Time: 2130.117 ms
```

Mémoire :

Les processus parallélisés sont susceptibles d'utiliser chacun l'équivalent des ressources mémoire d'un processus.

Concrètement, chaque *parallel worker* d'un nœud consommant de la mémoire est susceptible d'utiliser la quantité définie par `work_mem`. La parallélisation peut donc augmenter le besoin en mémoire.

Paramétrage :

Le défaut de `max_worker_processes` est 8. Ne descendez pas plus bas, car les *background workers* sont de plus en plus utilisés par les nouvelles fonctionnalités de PostgreSQL et les extensions tierces, et modifier ce paramètre implique de redémarrer. Sur les machines modernes, il peut être monté assez haut (bien au-delà du nombre de cœurs).

Les autres paramètres peuvent être modifiés avec `SET` au sein d'une session.

Le choix de `max_parallel_workers` dépend du nombre de cœurs. Le défaut de 8 est trop bas pour la plupart des machines récentes. La valeur de `max_parallel_workers_per_gather` dépend du type des grosses requêtes et du nombre de requêtes tournant simultanément. Les petites requêtes (OLTP) profiteront rarement du parallélisme. Des configurations assez agressives sur de grosses configurations vont bien au-delà du nombre de cœurs[23].

Cependant, il ne sert à rien de trop paralléliser si les disques ne suivent pas, ou si le nombre de cœurs est réduit. Et si ces paramètres sont trop hauts, il y a un risque que les grosses requêtes saturent les CPU au détriment des plus petites et d'autres processus.

Par contre, si le paramétrage est trop prudent, les CPU seront sous-utilisés. De nombreuses requêtes candidates au parallélisme peuvent se retrouver privées de *worker*, ce qui mènera à des plans suboptimaux. Là encore, la supervision et l'expérimentation prudente sont nécessaires.

[23] https://thebuild.com/blog/2023/02/08/xtreme-postgresql/

Création et maintenance d'index :

La création d'index B-tree peut être aussi être parallélisée depuis la version 11. Le paramètre `max_parallel_maintenance_workers`, par défaut à 2, indique le nombre de *workers* utilisables lors de la création d'un index, mais aussi de son nettoyage lors d'un `VACUUM`. Le gain de temps peut être appréciable. Cette opération étant assez rare, le paramètre peut être monté assez haut. Les limites de `max_worker_processes` et `max_parallel_workers` s'appliquent là encore.

Contrairement au cas de `work_mem` qui peut être alloué par chaque *worker* lors d'un tri, `maintenance_work_mem` est allouée une seule fois et partagé entre les différents workers.

Référence :

La documentation a un chapitre entier sur le sujet du parallélisme[24].

1.4.7 Configuration - WAL

- `fsync` (`on` !)
- `min_wal_size` (80 Mo) / `max_wal_size` (1 Go)
- `checkpoint_timeout` (5 min, ou plus)
- `checkpoint_completion_target` (passer à 0.9)

`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 système pour le noyau qui, pour des raisons de performances, conserve dans un premier temps les données dans un cache. En cas de coupure de 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. Cela entraînera 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`, et est activé par défaut (`on`). C'est essentiel pour la fiabilité, même si cela impacte très négativement les performances en écriture en cas de nombreuses transactions. Il est donc obligatoire en production de conserver ce paramètre activé. Pour accélérer de très gros imports ou restaurations, on peut le passer exceptionnellement à `off`, et l'on n'oubliera pas de revenir à `on`.

Chaque bloc modifié dans le cache disque de PostgreSQL doit être écrit sur disque au bout d'un certain temps. Le premier paramètre concerné est `checkpoint_timeout`, qui permet de déclencher un

[24] https://docs.postgresql.fr/current/parallel-query.html

`CHECKPOINT` au moins toutes les X minutes (5 par défaut). Pour lisser des grosses écritures, on le monte fréquemment à 15 minutes voire plus. Cela peut aussi avoir l'intérêt de réduire un peu la taille des journaux générés : en effet, PostgreSQL écrit un bloc modifié intégralement dans les journaux lors de sa première modification après un checkpoint et une fois que ce bloc intégral est enregistré, il n'écrit plus que des deltas du bloc correspondant aux modifications réalisées.

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 quelle quantité de journaux générés il faut lancer un `CHECKPOINT` :

- `min_wal_size` : quantité de WAL conservés pour le recyclage (par défaut 80 Mo) ;
- `max_wal_size` : quantité maximale de WAL avant un checkpoint (par défaut 1 Go, mais on peut monter beaucoup plus haut).

Le nom du paramètre `max_wal_size` peut porter à confusion : le volume de WAL peut dépasser largement `max_wal_size` en cas de forte activité ou de retard de l'archivage, ce n'est en aucun cas une valeur plafond.

(Avant la version 9.5, le paramètre équivalent à `min_wal_size` et `max_wal_size` se nommait `checkpoint_segments` et était exprimé en nombre de journaux de 16 Mo.)

Un checkpoint déclenché par atteinte des seuils `max_wal_size` ou `checkpoint_segments` apparaît dans `postgresql.log`. Si cela arrive trop fréquemment, il est conseillé d'augmenter ces paramètres.

`checkpoint_completion_target` permet de lisser les écritures du checkpoint pour éviter de saturer les disques par de grosses écritures au détriment des requêtes des utilisateurs. On le monte généralement à `0.9` (soit 90 % de `checkpoint_timeout`, donc 4 minutes et demie par défaut). D'ailleurs, à partir de la version 14, il s'agit de la valeur par défaut.

1.4.8 Configuration - statistiques

- `track_activities`
- `track_counts`
- `track_functions`, `track_io_timing` et `track_wal_io_timing`

Ces quatre 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).

Les deux premiers paramètres sont activés par défaut. Les désactiver peut vous faire un peu gagner en performance mais les informations que vous perdrez vous empêcheront d'aller très loin en matière d'optimisation. De plus, `track_counts` est requis pour que l'autovacuum puisse fonctionner.

D'autres paramètres, désactivés par défaut, permettent d'aller plus loin :

`track_functions` à `pl` ou `all` permet de récupérer des informations sur l'utilisation des routines stockées.

`track_io_timing` réalise un chronométrage des opérations de lecture et écriture disque. Il complète les champs `blk_read_time` et `blk_write_time` dans les tables `pg_stat_database` et `pg_stat_statements` (si cette extension[25] est installée). Il ajoute des traces suite à un `VACUUM` ou un `ANALYZE` exécutés par le processus `autovacuum` Dans les plans d'exécutions (avec `EXPLAIN (ANALYZE,BUFFERS)`), il permet l'affichage du temps passé à lire hors du cache de PostgreSQL (sur disque ou dans le cache de l'OS) :

```
I/O Timings: read=2.062
```

Avant d'activer `track_io_timing` sur une machine peu performante, vérifiez avec l'outil `pg_test_timing`[26] que la quasi-totalité des appels dure moins d'une nanoseconde.

La version 14 a ajouté le paramètre `track_wal_io_timing` qui permet de suivre les performances des opérations de lecture et écriture dans les WAL dans la vue `pg_stat_wal`. Par défaut, le paramètre est désactivé.

1.4.9 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.

Sa configuration est généralement trop basse pour être suffisamment efficace.

[25] https://dali.bo/x2_html#pg_stat_statements
[26] https://docs.postgresql.fr/current/pgtesttiming.html

1.4.10 Tablespaces : principe

- Espace de stockage physique d'objets
  - et non logique !
- Simple répertoire (**hors de PGDATA**)
  - lien symbolique depuis `pg_tblspc`
- Pour :
  - répartir I/O et volumétrie
  - données froides/chaudes
  - tri sur disque séparé

Dans PGDATA, le sous-répertoire `pg_tblspc` contient les *tablespaces*, c'est-à-dire des espaces de stockage.

Sous Linux, ce sont des liens symboliques vers un simple répertoire extérieur à PGDATA. Chaque lien symbolique a comme nom l'OID du tablespace (table système `pg_tablespace`). PostgreSQL y crée un répertoire lié aux versions de PostgreSQL et du catalogue, et y place les fichiers de données.

```
postgres=# \db+
                          Liste des tablespaces
    Nom     | Propriétaire |      Emplacement      | … | Taille  |…
------------+--------------+-----------------------+---+---------+-
 froid      | postgres     | /HDD/tbl/froid        |   | 3576 kB |
 pg_default | postgres     |                       |   | 6536 MB |
 pg_global  | postgres     |                       |   | 587 kB  |
```

```
sudo ls -R /HDD/tbl/froid
```

```
/HDD/tbl/froid:
PG_15_202209061

/HDD/tbl/froid/PG_15_202209061:
5

/HDD/tbl/froid/PG_15_202209061/5:
142532  142532_fsm  142532_vm
```

Sous Windows, les liens sont à proprement parler des *Reparse Points* (ou *Junction Points*) :

```
postgres=# \db
          Liste des tablespaces
    Nom     | Propriétaire | Emplacement
------------+--------------+-------------
 pg_default | postgres     |
```

```
 pg_global  | postgres     |
 tbl1       | postgres     | T:\TBL1
```

```
PS P:\PGDATA13> dir 'pg_tblspc/*' | ?{$_.LinkType} | select FullName,LinkType,Target

FullName                     LinkType Target
--------                     -------- ------
P:\PGDATA13\pg_tblspc\105921 Junction {T:\TBL1}
```

Par défaut, `pg_tblspc/` est vide. N'existent alors que les tablespaces `pg_global` (sous-répertoire `global/` des objets globaux à l'instance) et `pg_default` (soit `base/`).

1.4.10.1 Utilité des tablespaces

Un *tablespace*, vu de PostgreSQL, est un espace de stockage des objets (tables et index principalement). Son rôle est purement physique, il n'a pas à être utilisé pour une séparation *logique* des tables (c'est le rôle des bases et des schémas), encore moins pour gérer des droits.

Pour le système d'exploitation, il s'agit juste d'un répertoire, déclaré ainsi :

```
CREATE TABLESPACE ssd LOCATION '/var/lib/postgresql/tbl_ssd';
```

L'idée est de séparer physiquement les objets suivant leur utilisation. Les cas d'utilisation des tablespaces dans PostgreSQL sont :

- l'ajout d'un disque après saturation de la partition du PGDATA sans possibilité de l'étendre au niveau du système (par LVM ou dans la baie de stockage, par exemple) ;
- la répartition des entrées-sorties… si le SAN ou la virtualisation permet encore d'agir à ce niveau ;
- et notamment la séparation des index et des tables, pour répartir les écritures ;
- le déport des fichiers temporaires vers un tablespace dédié, pour la performance ou éviter qu'ils saturent le PGDATA ;
- la séparation entre données froides et chaudes sur des disques de performances différentes, ou encore des index et des tables ;
- les quotas : PostgreSQL ne disposant pas d'un système de quotas, dédier une partition entière d'une taille précise à un tablespace est un contournement ; une transaction voulant étendre un fichier sera alors annulée avec l'erreur `cannot extend file`.

Sans un réel besoin physique, il n'y a pas besoin de créer des tablespaces, et de complexifier l'administration.

Un tablespace n'est pas adapté à une séparation logique des objets. Si vous tenez à distinguer les fichiers de chaque base sans besoin physique, rappelez-vous que PostgreSQL crée déjà un sous-répertoire par base de données dans `PGDATA/base/`.

PostgreSQL ne connaît pas de notion de tablespace en lecture seule, ni de tablespace transportable entre deux bases ou deux instances.

1.4.10.2 Emplacement des tablespaces

Il y a quelques pièges à éviter à la définition d'un tablespace :

Pour des raisons de sécurité et de fiabilité, le répertoire choisi **ne doit pas** être à la racine d'un point de montage. (Cela vaut aussi pour les répertoires PGDATA ou `pg_wal`).

Positionnez toujours les données dans un sous-répertoire, voire deux niveaux en-dessous du point de montage.

Par exemple, déclarez votre PGDATA dans `/<point de montage>/<version majeure>/<nom instance>` plutôt que directement dans `/<point de montage>`.

Un tablespace ira dans `/<autre point de montage>/<nom répertoire>/` plutôt que directement dans `/<autre point de montage>/`.

(Voir *Utilisation de systèmes de fichiers secondaires*[27] dans la documentation officielle, ou le bug à l'origine de ce conseil[28].)

Surtout, le tablespace doit **impérativement être placé hors de PGDATA**. Certains outils poseraient problème sinon.

Si ce conseil n'est pas suivi, PostgreSQL crée le tablespace mais renvoie un avertissement :

```
WARNING:  tablespace location should not be inside the data directory
CREATE TABLESPACE
```

Il est aussi déconseillé de mettre le numéro de version de PostgreSQL dans le chemin du tablespace. PostgreSQL le gère à l'intérieur du tablespace, et en tient notamment compte dans les migrations avec `pg_upgrade`.

[27] https://doc.postgresql.fr/current/creating-cluster.html#CREATING-CLUSTER-MOUNT-POINTS
[28] https://bugzilla.redhat.com/show_bug.cgi?id=1247477#c1

1.4.11 Tablespaces : mise en place

```
CREATE TABLESPACE chaud LOCATION '/SSD/tbl/chaud';

CREATE DATABASE nom TABLESPACE 'chaud';

ALTER DATABASE nom SET default_tablespace TO 'chaud';

GRANT CREATE ON TABLESPACE chaud TO un_utilisateur ;

CREATE TABLE une_table (…) TABLESPACE chaud ;

ALTER TABLE une_table SET TABLESPACE chaud ;  -- verrou !

ALTER INDEX une_table_i_idx SET TABLESPACE chaud ; -- pas automatique
```

Le répertoire du tablespace doit exister et les accès ouverts et restreints à l'utilisateur système sous lequel tourne l'instance (en général **postgres** sous Linux, **Network Service** sous Windows) :

```
# mkdir /SSD/tbl/chaud
# chown postgres:postgres /SSD/tbl/chaud
# chmod 700 /SSD/tbl/chaud
```

Les ordres SQL plus haut permettent de :

- créer un tablespace simplement en indiquant son emplacement dans le système de fichiers du serveur ;
- créer une base de données dont le tablespace par défaut sera celui indiqué ;
- modifier le tablespace par défaut d'une base ;
- donner le droit de créer des tables dans un tablespace à un utilisateur (c'est nécessaire avant de l'utiliser) ;
- créer une table dans un tablespace ;
- déplacer une table dans un tablespace ;
- déplacer un index dans un tablespace.

Quelques choses à savoir :

- La table ou l'index est totalement verrouillé le temps du déplacement.
- Les index existants ne « suivent » pas automatiquement une table déplacée, il faut les déplacer séparément.
- Par défaut, les nouveaux index ne sont **pas** créés automatiquement dans le même tablespace que la table, mais en fonction de `default_tablespace`.

Les tablespaces des tables sont visibles dans la vue système `pg_tables`, dans `\d+` sous psql, et dans `pg_indexes` pour les index :

```
SELECT schemaname, indexname, tablespace
FROM   pg_indexes
WHERE  tablename = 'ma_table';
```

```
 schemaname |  indexname   | tablespace
------------+--------------+------------
 public     | matable_idx  | chaud
 public     | matable_pkey |
```

1.4.12 Tablespaces : configuration

- `default_tablespace`
- `temp_tablespaces`
- Droits à ouvrir :

```
GRANT CREATE ON TABLESPACE  ssd_tri  TO  dupont ;
```

- Performances :
  - `seq_page_cost`, `random_page_cost`
  - `effective_io_concurrency`, `maintenance_io_concurrency`

```
ALTER TABLESPACE chaud SET ( random_page_cost = 1 );
ALTER TABLESPACE chaud SET ( effective_io_concurrency    = 500,
                             maintenance_io_concurrency = 500 ) ;
```

1.4.12.1 Tablespaces de données

Le paramètre `default_tablespace` permet d'utiliser un autre tablespace que celui par défaut dans PGDATA. En plus du `postgresql.conf`, il peut être défini au niveau rôle, base, ou le temps d'une session :

```
ALTER DATABASE critique SET default_tablespace TO 'chaud' ; -- base
ALTER ROLE etl SET default_tablespace TO 'chaud' ; -- rôle
SET default_tablespace TO 'chaud' ; -- session
```

1.4.12.2 Tablespaces de tri

Les opérations de tri et les tables temporaires peuvent être déplacées vers un ou plusieurs tablespaces dédiés grâce au paramètre `temp_tablespaces`. Le premier intérêt est de dédier aux tris une partition rapide (SSD, disque local…). Un autre est de ne plus risquer de saturer la partition du PGDATA en cas de fichiers temporaires énormes dans `base/pgsql_tmp/`.

Ne jamais utiliser de ramdisk (comme tmpfs) pour des tablespaces de tri : la mémoire de la machine ne doit servir qu'aux applications et outils, au cache de l'OS, et aux tris en RAM. Favorisez ces derniers en jouant sur `work_mem`.
En cas de redémarrage, ce tablespace ne serait d'ailleurs plus utilisable. Un ramdisk est encore plus dangereux pour les tablespaces de données, bien sûr.

Il faudra ouvrir les droits aux utilisateurs ainsi :

```
GRANT CREATE ON TABLESPACE  ssd_tri  TO  dupont ;
```

Si plusieurs tablespaces de tri sont paramétrés, chaque transaction en choisira un de façon aléatoire à la création d'un objet temporaire, puis utilisera alternativement chaque tablespace. Un gros tri sera donc étalé sur plusieurs de ces tablespaces. afin de répartir la charge.

Paramètres de performances :

Dans le cas de disques de performances différentes, il faut adapter les paramètres concernés aux caractéristiques du tablespace si la valeur par défaut ne convient pas. Ce sont des paramètres classiques qui ne seront pas décrits en détail ici :

- `seq_page_cost` (coût d'accès à un bloc pendant un parcours) ;
- `random_page_cost` (coût d'accès à un bloc isolé) ;
- `effective_io_concurrency` (nombre d'I/O simultanées) et `maintenance_io_concurrency` (idem, pour une opération de maintenance).

Notamment : `effective_io_concurrency` a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Seuls les parcours *Bitmap Scan* sont impactés par ce paramètre. Selon la documentation[29], pour un système disque utilisant un RAID matériel, il faut

[29] https://docs.postgresql.fr/current/runtime-config-resource.html#GUC-EFFECTIVE-IO-CONCURRENCY

le configurer en fonction du nombre de disques utiles dans le RAID (n s'il s'agit d'un RAID 1, n-1 s'il s'agit d'un RAID 5 ou 6, n/2 s'il s'agit d'un RAID 10). Avec du SSD, il est possible de monter à plusieurs centaines, étant donné la rapidité de ce type de disque. À l'inverse, il faut tenir compte du nombre de requêtes simultanées qui utiliseront ce nœud. Le défaut est seulement de 1, et la valeur maximale est 1000. Attention, à partir de la version 13, le principe reste le même, mais la valeur exacte de ce paramètre doit être 2 à 5 fois plus élevée qu'auparavant, selon la formule des notes de version[30].

Toujours en version 13 apparaît `maintenance_io_concurrency`. similaire à `effective_io_concurrency`, mais pour les opérations de maintenance. Celles-ci peuvent ainsi se voir accorder plus de ressources qu'une simple requête. Le défaut est de 10, et il faut penser à le monter aussi si on adapte `effective_io_concurrency`.

Par exemple, sur un système paramétré pour des disques classiques, un tablespace sur un SSD peut porter ces paramètres :

```
ALTER TABLESPACE chaud SET ( random_page_cost = 1 );
ALTER TABLESPACE chaud SET ( effective_io_concurrency   = 500,
                             maintenance_io_concurrency = 500 ) ;
```

1.4.13 Emplacement des journaux de transactions

- Placer les journaux sur un autre disque
- Option `--wal-dir` 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 par défaut), 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.

La commande `initdb` permet de créer une instance en positionnant le répertoire dédié aux journaux de transaction en dehors du répertoire de données. Pour cela, il faut utiliser l'option `-X` ou `--waldir` :

```
$ initdb -X /montage/14/pgwal
```

Un lien symbolique est créé dans le répertoire de données pour que PostgreSQL retrouve le répertoire des journaux de transactions.

[30] https://docs.postgresql.fr/13/release.html

Si l'on souhaite modifier une instance existante, 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 `/montage/14/pgwal`.

```
# systemctl stop postgresql-14
# cd /var/lib/pgsql/14/data
# mv pg_wal /montage/14/pgwal
# ln -s /montage/14/pgwal pg_wal
# ls -l pg_wal
lrwxrwxrwx. 1 root root 6 Sep 18 16:07 pg_wal -> /montage/14/pgwal
# systemctl start postgresql-14
```

1.4.14 Emplacement des fichiers statistiques

- Avant la v15
  - placer les fichiers statistiques sur un autre disque
  - option `stats_temp_directory`
- À partir de la v15
  - cela peut être intéressant pour `pg_stat_statements`

PostgreSQL met à disposition différents compteurs statistiques via des vues.

Avant la version 15, les vues système utilisent des métriques stockées dans des fichiers de statistiques. Ces fichiers sont mis à jour par le processus `stats collector`. Ils sont localisés dans un répertoire pointé par le paramètre `stats_temp_directory`. Par défaut, les fichiers sont stockés dans le sous-répertoire `pg_stat_tmp` du répertoire principal des données. Habituellement, cela ne pose pas de difficultés, mais sous une forte charge, il peut apparaître une forte activité disque à cause du processus `stats collector`.

Lorsque le problème se pose, il est recommandé de déplacer ces fichiers en RAM avec la procédure suivante. Attention, le module `pg_stat_statements`, s'il est installé, sauvegarde le texte des requêtes également à cet emplacement, sans limite de taille pour la taille des requêtes. L'espace occupé par ces statistiques peut donc être très important. Le RAM-disk fera donc au moins 64, voire 128 Mo, pour tenir compte de ce changement.

Le point de montage employé doit être placé à l'extérieur du `PGDATA`.

Sur Debian, Ubuntu et dérivés, cela est fait par défaut : `stats_temp_directory` pointe vers un répertoire dans `/run`, qui est un `tmpfs`.

Sur Rocky Linux 8, la procédure est :

- création du point de montage (en tant qu'utilisateur **postgres**) :

```
$ mkdir /var/lib/pgsql/14/pg_stat_tmpfs
```

- création et montage du système de fichiers :

```
# mount -o \
    auto,nodev,nosuid,noexec,noatime,mode=0700,size=256M,uid=postgres,gid=postgres \
    -t tmpfs tmpfs /var/lib/pgsql/14/pg_stat_tmpfs
# mount
...
tmpfs on /var/lib/pgsql/14/pg_stat_tmpfs type tmpfs
       (rw,nosuid,nodev,noexec,noatime,seclabel,size=262144k,mode=700,uid=26,gid=26)
```

- modification de la configuration PostgreSQL dans `postgresql.conf` :

```
stats_temp_directory = '/var/lib/pgsql/14/pg_stat_tmpfs'
```

- recharger la configuration de PostgreSQL :

```
SELECT pg_reload_conf() ;

SHOW stats_temp_directory ;

 stats_temp_directory
----------------------
 /var/lib/pgsql/14/pg_stat_tmpfs
```

- vérifier dans `postgresql.conf` que le changement de paramètre a bien été pris en compte et qu'il n'y a pas d'erreur :

```
LOG:  received SIGHUP, reloading configuration files
LOG:  parameter "stats_temp_directory" changed to "/var/lib/pgsql/14/pg_stat_tmpfs"
```

- ajouter au fichier `/etc/fstab` la ligne suivante pour que la modification survive au prochain redémarrage :

```
tmpfs /var/lib/pgsql/14/pg_stat_tmpfs tmpfs auto,nodev,nosuid,noexec,
 ↪  noatime,uid=postgres,gid=postgres,mode=0700,size=256M  0 0
```

Depuis la version 15, le paramètre `stats_temp_directory` n'existe plus, car les informations statistiques sont conservées en mémoire partagée. À l'arrêt de l'instance, elles sont enregistrées sur disque dans le répertoire `pg_stat`. Comme `pg_stat` et `pg_stat_tmp/` restent utilisés par `pg_stat_statements` et d'autres extensions, le principe du lien symbolique vers `tmpfs` reste valable en version 15.

1.5 OUTILS

- pgtune
- pgbench
- postgresqltuner.pl

Nous allons discuter de trois outils.

Le premier, `pgtune`, est un petit script permettant d'obtenir rapidement et simplement une configuration un peu plus optimisée que celle proposée par la commande `initdb`.

Le second est livré avec les sources de PostgreSQL. `pgbench` a pour but de permettre la réalisation de benchmarks simples pour un serveur PostgreSQL.

Enfin, le dernier, `postgresqltuner.pl`, est un script permettant de vérifier la configuration matérielle, système et PostgreSQL d'un serveur.

1.5.1 Outil pgtune

- Outil écrit en Python, par Greg Smith
  - repris en Ruby par Alexey Vasiliev
- Propose quelques meilleures valeurs pour certains paramètres
- Quelques options pour indiquer des informations système
- Version web : https://pgtune.leopard.in.ua/
- Il existe également pgconfig[31]

Le site du projet en ruby[32] se trouve sur github.

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, *datawarehouse*, mixed, etc.

Commençons par une configuration pour une utilisation par une application web sur une machine avec 8 Go de RAM, 2 CPU, un stockage mécanique (HDD) :

[32] https://github.com/le0pard/pgtune

```
max_connections = 200
shared_buffers = 2GB
effective_cache_size = 6GB
maintenance_work_mem = 512MB
checkpoint_completion_target = 0.7
wal_buffers = 16MB
default_statistics_target = 100
random_page_cost = 4
effective_io_concurrency = 2
work_mem = 10485kB
min_wal_size = 1GB
max_wal_size = 2GB
max_worker_processes = 2
max_parallel_workers_per_gather = 1
max_parallel_workers = 2
```

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 au quart de la mémoire, alors que le paramètre `effective_cache_size` est au deux tiers évoqué précédemment. Le paramètre `wal_buffers` est aussi augmenté, il arrive à 16 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 `min_wal_size`, `max_wal_size` et `checkpoint_completion_target` sont augmentés mais là aussi très raisonnablement. La parallélisation est activée, sans excès.

Voyons maintenant avec un profil OLTP (*OnLine Transaction Processing*) :

```
max_connections = 300
shared_buffers = 2GB
effective_cache_size = 6GB
maintenance_work_mem = 512MB
checkpoint_completion_target = 0.9
wal_buffers = 16MB
default_statistics_target = 100
random_page_cost = 4
effective_io_concurrency = 2
work_mem = 6990kB
min_wal_size = 2GB
max_wal_size = 4GB
max_worker_processes = 2
max_parallel_workers_per_gather = 1
max_parallel_workers = 2
```

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. Due au fait qu'il y aura plus d'écritures, la taille du cache des journaux de transactions (`wal_buffers`) est augmentée. Il faudra essayer de tout faire passer par les *checkpoints*, d'où la valeur maximale pour `checkpoint_completion_target`,

et des valeurs encore augmentées pour `min_wal_size` et `max_wal_size`. 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 (*datawarehouse*) sur une machine moins modeste avec 32 Go de RAM, 8 cœurs, et un disque SSD :

```
max_connections = 40
shared_buffers = 8GB
effective_cache_size = 24GB
maintenance_work_mem = 2GB
checkpoint_completion_target = 0.9
wal_buffers = 16MB
default_statistics_target = 500
random_page_cost = 1.1
effective_io_concurrency = 200
work_mem = 26214kB
min_wal_size = 4GB
max_wal_size = 8GB
max_worker_processes = 8
max_parallel_workers_per_gather = 4
max_parallel_workers = 8
```

Pour un entrepôt de données, il y a généralement peu d'utilisateurs à un instant t, mais qui 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 26 Mo). `shared_buffers` et `effective_cache_size` suivent les règles habituelles. 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 des `min_wal_size` / `max_wal_size`. Du fait de la grosse volumétrie des bases dans ce contexte, une valeur importante pour le `maintenance_work_mem` est essentielle 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. `random_page_cost`, auparavant à sa valeur par défaut, descend à 1.1 pour tenir compte des performances d'un SSD. C'est le cas aussi pour `effective_io_concurrency`. Enfin, la configuration de la machine autorise une parallélisation importante, généralement bienvenue pour du décisionnel.

Évidemment, tout ceci n'est qu'une recommandation générale, et ne doit servir que de point de départ. Chaque paramètre peut être affiné. L'expérimentation permettra de se diriger vers une configuration plus personnalisée.

1.5.2 Outil pgbench

- Outil pour réaliser rapidement des tests de performance
- Fourni dans les modules de contrib de PostgreSQL
- Travaille sur une base de test créée par l'outil…
  - … ou sur une vraie base de données

pgbench est un outil disponible avec les modules contrib de PostgreSQL depuis de nombreuses années. Son but est de faciliter la mise en place de benchmarks simples et rapides. Des solutions plus complètes sont disponibles, mais elles sont aussi bien plus complexes.

pgbench travaille soit à partir d'un schéma de base qu'il crée et alimente lui-même, soit à partir d'une base déjà existante. Dans ce dernier cas, les requêtes SQL à exécuter sont à fournir à pgbench.

Il existe donc principalement deux modes d'utilisation de pgbench : le mode initialisation quand on veut utiliser le schéma et le scénario par défaut, et le mode benchmarks.

1.5.3 Types de tests avec pgbench

- 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.

1.5.4 Environnement de test avec pgbench

- pgbench est capable de créer son propre environnement de test
- Environnement adapté pour des tests de type TPC-B
- Permet de rapidement tester une configuration PostgreSQL
  - en termes de performance
  - en termes de charge
- Ou pour expérimenter/tester

La documentation complète est sur https://docs.postgresql.fr/current/pgbench.html. L'auteur principal, Fabien Coelho, a fait une présentation complète, en français, à la PG Session #9 de 2017[33].

L'option `-i` demande à pgbench de créer un schéma et de le peupler de données 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 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`. Pour que le test soit significatif, il est important que la taille de la base dépasse fortement la quantité de mémoire disponible.

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

Quelques exemples. Le plus simple :

- création de la base et peuplement par pgbench :

```
$ createdb benchs
$ pgbench -i -s 2 benchs
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
creating tables...
100000 of 200000 tuples (50%) done (elapsed 0.08 s, remaining 0.08 s)
200000 of 200000 tuples (100%) done (elapsed 0.26 s, remaining 0.00 s)
vacuum...
set primary keys...
done.
```

- benchmarks sur cette base :

[33] https://youtu.be/aTwh_CgRaE0

```
$ pgbench benchs
starting vacuum...end.
transaction type: <builtin: 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
latency average = 2.732 ms
tps = 366.049857 (including connections establishing)
tps = 396.322853 (excluding connections establishing)
```

- nouveau test avec 10 clients et 200 transactions pour chacun :

```
$ pgbench -c 10 -t 200 benchs
starting vacuum...end.
transaction type: <builtin: 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
latency average = 19.716 ms
tps = 507.204902 (including connections establishing)
tps = 507.425131 (excluding connections establishing)
```

- changement de la configuration avec `fsync=off`, et nouveau test avec les mêmes options que précédemment :

```
$ pgbench -c 10 -t 200 benchs
starting vacuum...end.
transaction type: <builtin: 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
latency average = 2.361 ms
tps = 4234.926931 (including connections establishing)
tps = 4272.412154 (excluding connections establishing)
```

- toujours avec les mêmes options, mais en effectuant le test durant 10 secondes :

```
$ pgbench -c 10 -T 10 benchs
starting vacuum...end.
transaction type: <builtin: 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: 45349
latency average = 2.207 ms
```

```
tps = 4531.835068 (including connections establishing)
tps = 4534.070449 (excluding connections establishing)
```

1.5.5 Environnement réel avec pgbench

- pgbench est capable de travailler avec une base existante
- Lecture des requêtes depuis un ou plusieurs fichiers
- Utilisation possible de variables et commandes

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

Pour effectuer de tels tests, il faut créer un ou plusieurs scripts SQL contenant les requêtes à exécuter sur la base de donnée. Avant la 9.6, chaque requête doit être écrite sur **UNE** seule ligne, sinon le point-virgule `;` habituel convient. 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 groupe de données manipulé dans vos tests. Ce dernier point est essentiel afin d'éviter les effets de cache ou encore pour simuler la charge lorsqu'un sous-ensemble des données de la base est utilisé en comparaison avec la totalité de la base (en utilisant un champ 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 aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-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;
```

Si la base de donnée utilisée est celle fournie avec pgbench, la variable `:scale` sera valorisée à partir du nombre de lignes présentes dans la relation `pgbench_branches`. Si vous utilisez une autre base de donnée, il faut indiquer l'échelle avec l'option `-s` lorsque l'on exécute le test.

1.5.6 Outil postgresqltuner.pl

- Outil écrit en `Perl` par Julien Francoz
- Propose quelques meilleures valeurs pour certains paramètres
  - système
  - PostgreSQL
- Site du projet : https://github.com/jfcoz/postgresqltuner

Ci-dessous figure un exemple de sortie en version 11 sur une machine virtuelle avec un SSD. Les conseils sont généralement pertinents. Il conseille notamment d'installer l'extension `pg_stat_statements`[34] et d'attendre un certain temps avant d'utiliser l'outil.

```
-bash-4.2$ ./postgresqltuner.pl --ssd

postgresqltuner.pl version 1.0.1
Checking if OS commands are available on /var/run/postgresql...
[OK]      OS command OK
Connecting to /var/run/postgresql:5432 database template1 with user postgres...
[OK]      User used for report has superuser rights
=====  OS information  =====
[INFO]    OS: linux Version: 3.10.0-693.17.1.el7.x86_64
          Arch: x86_64-linux-thread-multi
[INFO]    OS total memory: 3.70 GB
[OK]      vm.overcommit_memory is good: no memory overcommitment
[INFO]    Running in kvm hypervisor
[INFO]    Currently used I/O scheduler(s): mq-deadline
=====  General instance informations  =====
-----  Version  -----
[OK]      You are using latest major 11.5
-----  Uptime  -----
[INFO]    Service uptime:  44s
[WARN]    Uptime is less than 1 day. postgresqltuner.pl result may not be accurate
-----  Databases  -----
[INFO]    Database count (except templates): 2
[INFO]    Database list (except templates): postgres postgis
-----  Extensions  -----
[INFO]    Number of activated extensions: 1
[INFO]    Activated extensions: plpgsql
[WARN]    Extensions pg_stat_statements is disabled in database template1
-----  Users  -----
[OK]      No user account will expire in less than 7 days
[OK]      No user with password=username
[OK]      Password encryption is enabled
-----  Connection information  -----
[INFO]    max_connections: 100
```

[34] https://dali.bo/x2_html#pg_stat_statements

```
[INFO]    current used connections: 6 (6.00%)
[INFO]    3 connections are reserved for super user (3.00%)
[INFO]    Average connection age:  36s
[BAD]     Average connection age is less than 1 minute.
          Use a connection pooler to limit new connection/seconds
-----  Memory usage  -----
[INFO]    configured work_mem: 4.00 MB
[INFO]    Using an average ratio of work_mem buffers by connection of 150%
          (use --wmp to change it)
[INFO]    total work_mem (per connection): 6.00 MB
[INFO]    shared_buffers: 128.00 MB
[INFO]    Track activity reserved size: 0.00 B
[WARN]    maintenance_work_mem is less or equal default value.
          Increase it to reduce maintenance tasks time
[INFO]    Max memory usage:
                  shared_buffers (128.00 MB)
                + max_connections * work_mem *
                  average_work_mem_buffers_per_connection
                  (100 * 4.00 MB * 150 / 100 = 600.00 MB)
                + autovacuum_max_workers * maintenance_work_mem
                  (3 * 64.00 MB = 192.00 MB)
                + track activity size (0.00 B)
                = 920.00 MB
[INFO]    effective_cache_size: 4.00 GB
[INFO]    Size of all databases: 29.55 MB
[WARN]    shared_buffer is too big for the total databases size, memory is lost
[INFO]    PostgreSQL maximum memory usage: 24.28% of system RAM
[WARN]    Max possible memory usage for PostgreSQL is less than 60% of
          system total RAM. On a dedicated host you can increase PostgreSQL
          buffers to optimize performances.
[INFO]    max memory+effective_cache_size is 132.37% of total RAM
[WARN]    the sum of max_memory and effective_cache_size is too high,
          the planner can find bad plans if system cache is smaller than expected
-----  Huge pages  -----
[BAD]     No Huge Pages available on the system
[BAD]     huge_pages disabled in PostgreSQL
[INFO]    Hugepagesize is 2048 kB
[INFO]    HugePages_Total 0 pages
[INFO]    HugePages_Free 0 pages
[INFO]    Suggested number of Huge Pages: 195
          (Consumption peak: 398868 / Huge Page size: 2048)
-----  Logs  -----
[OK]      log_hostname is off: no reverse DNS lookup latency
[WARN]    log of long queries is deactivated. It will be more
          difficult to optimize query performances
[OK]      log_statement=none
-----  Two phase commit  -----
[OK]      Currently no two phase commit transactions
-----  Autovacuum  -----
[OK]      autovacuum is activated.
[INFO]    autovacuum_max_workers: 3
-----  Checkpoint  -----
[WARN]    checkpoint_completion_target(0.5) is low
-----  Disk access  -----
[OK]      fsync is on
[OK]      synchronize_seqscans is on
```

```
-----  WAL  -----
-----  Planner  -----
[OK]      cost settings are defaults
[WARN]    With SSD storage, set random_page_cost=seq_page_cost
          to help planner use more index scan
[BAD]     some plan features are disabled: enable_partitionwise_aggregate,
          enable_partitionwise_join
=====  Database information for database template1  =====
-----  Database size  -----
[INFO]    Database template1 total size: 7.88 MB
[INFO]    Database template1 tables size: 4.85 MB (61.55%)
[INFO]    Database template1 indexes size: 3.03 MB (38.45%)
-----  Tablespace location  -----
[OK]      No tablespace in PGDATA
-----  Shared buffer hit rate  -----
[INFO]    shared_buffer_heap_hit_rate: 99.31%
[INFO]    shared_buffer_toast_hit_rate: 0.00%
[INFO]    shared_buffer_tidx_hit_rate: 58.82%
[INFO]    shared_buffer_idx_hit_rate: 99.63%
[OK]      Shared buffer idx hit rate is very good
-----  Indexes  -----
[OK]      No invalid indexes
[OK]      No unused indexes
-----  Procedures  -----
[OK]      No procedures with default costs

=====  Configuration advice  =====
-----  checkpoint  -----
[MEDIUM] Your checkpoint completion target is too low.
        Put something nearest from 0.8/0.9 to balance your writes better
        during the checkpoint interval
-----  extension  -----
[LOW] Enable pg_stat_statements in database template1 to collect statistics
      on all queries (not only queries longer than log_min_duration_statement
      in logs)
-----  hugepages  -----
[LOW] Change Huge Pages size from 2MB to 1GB
[MEDIUM] Enable huge_pages in PostgreSQL to consume system Huge Pages
[MEDIUM] set vm.nr_hugepages=195 in /etc/sysctl.conf and run sysctl -p
         to reload it. This will allocate huge pages (may require system reboot).
-----  planner  -----
[MEDIUM] Set random_page_cost=seq_page_cost on SSD disks
```

1.6 CONCLUSION

- PostgreSQL propose de nombreuses voies d'optimisation.
- Cela passe en priorité par un bon choix des composants matériels et par une configuration pointilleuse.
- Mais ceci ne peut se faire qu'en connaissance de l'ensemble du système, et notamment des applications utilisant les bases de l'instance.

1.6.1 Questions

N'hésitez pas, c'est le moment !

1.7 QUIZ

https://dali.bo/j1_quiz

1.8 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

L'installation est détaillée ici pour Rocky Linux 8 et 9 (similaire à Red Hat et à d'autres variantes comem Oracle Linux et Fedora), et Debian/Ubuntu.

Elle ne dure que quelques minutes.

1.8.1 Sur Rocky Linux 8 ou 9

ATTENTION : Red Hat, CentOS, Rocky Linux fournissent souvent par défaut des versions de PostgreSQL qui ne sont plus supportées. Ne jamais installer les packages `postgresql`, `postgresql-client` et `postgresql-server` ! L'utilisation des dépôts du PGDG est fortement conseillée.

Installation du dépôt communautaire :

Les dépôts de la communauté sont sur https://yum.postgresql.org/. Les commandes qui suivent sont inspirées de celles générées par l'assistant sur https://www.postgresql.org/download/linux/redhat/, en précisant :

- la version majeure de PostgreSQL (ici la 16) ;
- la distribution (ici Rocky Linux 8) ;
- l'architecture (ici x86_64, la plus courante).

Les commandes sont à lancer sous **root** :

```
# dnf install -y https://download.postgresql.org/pub/repos/yum/reporpms\
/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm

# dnf -qy module disable postgresql
```

Installation de PostgreSQL 16 (client, serveur, librairies, extensions) :

```
# dnf install -y postgresql16-server postgresql16-contrib
```

Les outils clients et les librairies nécessaires seront automatiquement installés.

Une fonctionnalité avancée optionnelle, le JIT (*Just In Time compilation*), nécessite un paquet séparé.

```
# dnf install postgresql16-llvmjit
```

Création d'une première instance :

Il est conseillé de déclarer `PG_SETUP_INITDB_OPTIONS`, notamment pour mettre en place les sommes de contrôle et forcer les traces en anglais :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-16/bin/postgresql-16-setup initdb
# cat /var/lib/pgsql/16/initdb.log
```

Ce dernier fichier permet de vérifier que tout s'est bien passé et doit finir par :

```
Success. You can now start the database server using:

    /usr/pgsql-16/bin/pg_ctl -D /var/lib/pgsql/16/data/ -l logfile start
```

Chemins :

Objet	Chemin
Binaires	`/usr/pgsql-16/bin`
Répertoire de l'utilisateur **postgres**	`/var/lib/pgsql`
`PGDATA` par défaut	`/var/lib/pgsql/16/data`
Fichiers de configuration	dans `PGDATA/`
Traces	dans `PGDATA/log`

Configuration :

Modifier `postgresql.conf` est facultatif pour un premier lancement.

Commandes d'administration habituelles :

Démarrage, arrêt, statut, rechargement à chaud de la configuration, redémarrage :

```
# systemctl start postgresql-16
# systemctl stop postgresql-16
# systemctl status postgresql-16
# systemctl reload postgresql-16
# systemctl restart postgresql-16
```

Test rapide de bon fonctionnement et connexion à psql :

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Démarrage de l'instance au lancement du système d'exploitation :

```
# systemctl enable postgresql-16
```

Ouverture du *firewall* pour le port 5432 :

Voir si le *firewall* est actif :

```
# systemctl status firewalld
```

Si c'est le cas, autoriser un accès extérieur :

```
# firewall-cmd --zone=public --add-port=5432/tcp --permanent
# firewall-cmd --reload
# firewall-cmd --list-all
```

(Rappelons que `listen_addresses` doit être également modifié dans `postgresql.conf`.)

Création d'autres instances :

Si des instances de *versions majeures différentes* doivent être installées, il faut d'abord installer les binaires pour chacune (adapter le numéro dans `dnf install …`) et appeler le script d'installation de chaque version. l'instance par défaut de chaque version vivra dans un sous-répertoire numéroté de `/var/lib/pgsql` automatiquement créé à l'installation. Il faudra juste modifier les ports dans les `postgresql.conf` pour que les instances puissent tourner simultanément.

Si plusieurs instances d'une *même version majeure* (forcément de la même version mineure) doivent cohabiter sur le même serveur, il faut les installer dans des `PGDATA` différents.

- Ne pas utiliser de tiret dans le nom d'une instance (problèmes potentiels avec systemd).
- Respecter les normes et conventions de l'OS : placer les instances dans un nouveau sous-répertoire de `/var/lib/pgsqsl/16/` (ou l'équivalent pour d'autres versions majeures).

Pour créer une seconde instance, nommée par exemple **infocentre** :

- Création du fichier service de la deuxième instance :

```
# cp /lib/systemd/system/postgresql-16.service \
        /etc/systemd/system/postgresql-16-infocentre.service
```

- Modification de ce dernier fichier avec le nouveau chemin :

```
Environment=PGDATA=/var/lib/pgsql/16/infocentre
```

- Option 1 : création d'une nouvelle instance vierge :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-16/bin/postgresql-16-setup initdb postgresql-16-infocentre
```

- Option 2 : restauration d'une sauvegarde : la procédure dépend de votre outil.
- Adaptation de `/var/lib/pgsql/16/infocentre/postgresql.conf` (port surtout).
- Commandes de maintenance de cette instance :

```
# systemctl [start|stop|reload|status] postgresql-16-infocentre
# systemctl [enable|disable] postgresql-16-infocentre
```

- Ouvrir le nouveau port dans le firewall au besoin.

1.8.2 Sur Debian / Ubuntu

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Référence : https://apt.postgresql.org/

Installation du dépôt communautaire :

L'installation des dépôts du PGDG est prévue dans le paquet Debian :

```
# apt update
# apt install -y  gnupg2  postgresql-common
# /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

Ce dernier ordre créera le fichier du dépôt `/etc/apt/sources.list.d/pgdg.list` adapté à la distribution en place.

Installation de PostgreSQL 16 :

La méthode la plus propre consiste à modifier la configuration par défaut avant l'installation :

Dans `/etc/postgresql-common/createcluster.conf`, paramétrer au moins les sommes de contrôle et les traces en anglais :

```
initdb_options = '--data-checksums --lc-messages=C'
```

Puis installer les paquets serveur et clients et leurs dépendances :

```
# apt install postgresql-16 postgresql-client-16
```

La première instance est automatiquement créée, démarrée et déclarée comme service à lancer au démarrage du système. Elle porte un nom (par défaut `main`).

Elle est immédiatement accessible par l'utilisateur système **postgres**.

Chemins :

Objet	Chemin
Binaires	`/usr/lib/postgresql/16/bin/`
Répertoire de l'utilisateur **postgres**	`/var/lib/postgresql`
PGDATA de l'instance par défaut	`/var/lib/postgresql/16/main`
Fichiers de configuration	dans `/etc/postgresql/16/main/`
Traces	dans `/var/log/postgresql/`

Configuration

Modifier `postgresql.conf` est facultatif pour un premier essai.

Démarrage/arrêt de l'instance, rechargement de configuration :

Debian fournit ses propres outils, qui demandent en paramètre la version et le nom de l'instance :

```
# pg_ctlcluster 16 main [start|stop|reload|status|restart]
```

Démarrage de l'instance avec le serveur :

C'est en place par défaut, et modifiable dans `/etc/postgresql/16/main/start.conf`.

Ouverture du firewall :

Debian et Ubuntu n'installent pas de firewall par défaut.

Statut des instances du serveur :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement et connexion à psql :

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Destruction d'une instance :

```
# pg_dropcluster 16 main
```

Création d'autres instances :

Ce qui suit est valable pour remplacer l'instance par défaut par une autre, par exemple pour mettre les *checksums* en place :

- optionnellement, `/etc/postgresql-common/createcluster.conf` permet de mettre en place tout d'entrée les *checksums*, les messages en anglais, le format des traces ou un emplacement séparé pour les journaux :

```
initdb_options = '--data-checksums --lc-messages=C'
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
waldir = '/var/lib/postgresql/wal/%v/%c/pg_wal'
```

- créer une instance :

```
# pg_createcluster 16 infocentre
```

Il est également possible de préciser certains paramètres du fichier `postgresql.conf`, voire les chemins des fichiers (il est conseillé de conserver les chemins par défaut) :

```
# pg_createcluster 16 infocentre \
  --port=12345 \
  --datadir=/PGDATA/16/infocentre \
  --pgoption shared_buffers='8GB' --pgoption work_mem='50MB' \
  --  --data-checksums --waldir=/ssd/postgresql/16/infocentre/journaux
```

- adapter au besoin `/etc/postgresql/16/infocentre/postgresql.conf` ;
- démarrage :

```
# pg_ctlcluster 16 infocentre start
```

1.8.3 Accès à l'instance depuis le serveur même (toutes distributions)

Par défaut, l'instance n'est accessible que par l'utilisateur système **postgres**, qui n'a pas de mot de passe. Un détour par `sudo` est nécessaire :

```
$ sudo -iu postgres psql
psql (16.0)
Type "help" for help.
postgres=#
```

Ce qui suit permet la connexion directement depuis un utilisateur du système :

Pour des tests (pas en production !), il suffit de passer à `trust` le type de la connexion en local dans le `pg_hba.conf` :

```
local   all             postgres                                trust
```

La connexion en tant qu'utilisateur `postgres` (ou tout autre) n'est alors plus sécurisée :

```
dalibo:~$ psql -U postgres
psql (16.0)
Type "help" for help.
postgres=#
```

Une authentification par mot de passe est plus sécurisée :

- dans `pg_hba.conf`, paramétrer une authentification par mot de passe pour les accès depuis `localhost` (déjà en place sous Debian) :

```
# IPv4 local connections:
host    all             all             127.0.0.1/32            scram-sha-256
# IPv6 local connections:
host    all             all             ::1/128                 scram-sha-256
```

(Ne pas oublier de recharger la configuration en cas de modification.)

- ajouter un mot de passe à l'utilisateur `postgres` de l'instance :

```
dalibo:~$ sudo -iu postgres psql
psql (16.0)
Type "help" for help.
postgres=# \password
Enter new password for user "postgres":
Enter it again:
postgres=# quit

dalibo:~$ psql -h localhost -U postgres
Password for user postgres:
psql (16.0)
Type "help" for help.
postgres=#
```

- Pour se connecter sans taper le mot de passe à une instance, un fichier `.pgpass` dans le répertoire personnel doit contenir les informations sur cette connexion :

```
localhost:5432:*:postgres:motdepassetrèslong
```

Ce fichier doit être protégé des autres utilisateurs :

```
$ chmod 600 ~/.pgpass
```

- Pour n'avoir à taper que `psql`, on peut définir ces variables d'environnement dans la session voire dans `~/.bashrc` :

```
export PGUSER=postgres
export PGDATABASE=postgres
export PGHOST=localhost
```

Rappels :

- en cas de problème, consulter les traces (dans `/var/lib/pgsql/16/data/log` ou `/var/log/postgresql/`) ;
- toute modification de `pg_hba.conf` ou `postgresql.conf` impliquant de recharger la configuration peut être réalisée par une de ces trois méthodes en fonction du système :

```
root:~# systemctl reload postgresql-16

root:~# pg_ctlcluster 16 main reload

postgres:~$ psql -c 'SELECT pg_reload_conf()'
```

1.9 INTRODUCTION À PGBENCH

pgbench est un outil de test livré avec PostgreSQL. Son but est de faciliter la mise en place de benchmarks simples et rapides. Par défaut, il installe une base assez simple, génère une activité plus ou moins intense et calcule le nombre de transactions par seconde et la latence. C'est ce qui sera fait ici dans cette introduction. On peut aussi lui fournir ses propres scripts.

La documentation complète est sur https://docs.postgresql.fr/current/pgbench.html. L'auteur principal, Fabien Coelho, a fait une présentation complète, en français, à la PG Session #9 de 2017[35].

1.9.1 Installation

L'outil est installé avec les paquets habituels de PostgreSQL, client ou serveur suivant la distribution.

Dans le cas des paquets RPM du PGDG, l'outil n'est pas dans le PATH par défaut ; il faudra donc fournir le chemin complet :

```
/usr/pgsql-16/bin/pgbench
```

Il est préférable de créer un rôle non privilégié dédié, qui possédera la base de données :

```
CREATE ROLE pgbench LOGIN PASSWORD 'unmotdepassebienc0mplexe';
CREATE DATABASE pgbench OWNER pgbench ;
```

Le `pg_hba.conf` doit éventuellement être adapté.

La base par défaut s'installe ainsi (indiquer la base de données en dernier ; ajouter `-p` et `-h` au besoin) :

```
pgbench -U pgbench --initialize --scale=100 pgbench
```

`--scale` permet de faire varier proportionnellement la taille de la base. À 100, la base pèsera 1,5 Go, avec 10 millions de lignes dans la table principale `pgbench_accounts` :

```
pgbench@pgbench=# \d+
                          Liste des relations
 Schéma |       Nom        | Type  | Propriétaire | Taille  | Description
--------+------------------+-------+--------------+---------+-------------
 public | pg_buffercache   | vue   | postgres     | 0 bytes |
 public | pgbench_accounts | table | pgbench      | 1281 MB |
 public | pgbench_branches | table | pgbench      | 40 kB   |
 public | pgbench_history  | table | pgbench      | 0 bytes |
 public | pgbench_tellers  | table | pgbench      | 80 kB   |
```

1.9.2 Générer de l'activité

Pour simuler une activité de 20 clients simultanés, répartis sur 4 processeurs, pendant 100 secondes :

[35] https://youtu.be/aTwh_CgRaE0

```
pgbench -U pgbench -c 20 -j 4 -T100 pgbench
```

NB : ne **pas** utiliser `-d` pour indiquer la base, qui signifie `--debug` pour pgbench, qui noiera alors l'affichage avec ses requêtes :

```
UPDATE pgbench_accounts SET abalance = abalance + -3455 WHERE aid = 3789437;
SELECT abalance FROM pgbench_accounts WHERE aid = 3789437;
UPDATE pgbench_tellers SET tbalance = tbalance + -3455 WHERE tid = 134;
UPDATE pgbench_branches SET bbalance = bbalance + -3455 WHERE bid = 78;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
 VALUES (134, 78, 3789437, -3455, CURRENT_TIMESTAMP);
```

À la fin, s'affichent notamment le nombre de transactions (avec et sans le temps de connexion) et la durée moyenne d'exécution du point de vue du client (`latency`) :

```
scaling factor: 100
query mode: simple
number of clients: 20
number of threads: 4
duration: 10 s
number of transactions actually processed: 20433
latency average = 9.826 ms
tps = 2035.338395 (including connections establishing)
tps = 2037.198912 (excluding connections establishing)
```

Modifier le paramétrage est facile grâce à la variable d'environnement `PGOPTIONS` :

```
PGOPTIONS='-c synchronous_commit=off -c commit_siblings=20' \
            pgbench -d pgbench -U pgbench -c 20 -j 4 -T100   2>/dev/null

latency average = 6.992 ms
tps = 2860.465176 (including connections establishing)
tps = 2862.964803 (excluding connections establishing)
```

Des tests rigoureux doivent durer bien sûr beaucoup plus longtemps que 100 s, par exemple pour tenir compte des effets de cache, des checkpoints périodiques, etc.

1.10 TRAVAUX PRATIQUES

1.10.1 Utilisation de pgbench

But : Manipuler pgbench

Créer une base **pgbench** sur laquelle seront effectués les premiers tests, et l'initialiser avec 13,5 millions de lignes dans la table `pgbench_accounts` :

```
# en tant qu'utilisateur postgres
createdb pgbench
/usr/pgsql-16/bin/pgbench -i -s 135 pgbench
```

Simuler l'utilisation de la base **pgbench** par 3 clients simultanés, chacun effectuant par exemple 1000 transactions :

```
# en tant que postgres
/usr/pgsql-16/bin/pgbench -c 3 -t 1000 -n -P1 pgbench
```

Simuler la même chose, en ajoutant le paramètre `-C` pour forcer une connexion à chaque transaction. Comparer la latence et le nombre de transactions par seconde.

On veut simuler l'utilisation de la base **pgbench** par 3 utilisateurs effectuant 10 fois la sélection des comptes dont le solde est positif dans la table `pgbench_accounts`.

Créer un fichier `query.sql` contenant la requête suivante :

```
SELECT aid,bid,abalance
FROM pgbench_accounts
WHERE abalance > 0;
```

Lancer la commande :

```
/usr/pgsql-16/bin/pgbench -c 3 -t 10 -n -P1 -f query.sql pgbench
```

1.10.2 Influence de fsync et synchronous_commit

But : Mesurer l'impact de fsync

Positionner le paramètre `fsync` à `off` dans le fichier `postgresql.conf` et redémarrer PostgreSQL.

Simuler l'utilisation de la base **pgbench** par 3 clients simultanés, chacun effectuant par exemple 1000 transactions :

```
/usr/pgsql-16/bin/pgbench -c 3 -t 1000 -n -P1 pgbench
```

Comparer avec le premier test précédent.

Rétablir `fsync` à `on`, puis relancer PostgreSQL.

Refaire le même test avec `synchronous_commit` à `off` dans la session :

```
PGOPTIONS='-c synchronous_commit=off' \
        /usr/pgsql-16/bin/pgbench -c 3 -t 1000 pgbench
```

1.11 TRAVAUX PRATIQUES (SOLUTIONS)

1.11.1 Utilisation de pgbench

> Créer une base **pgbench** sur laquelle seront effectués les premiers tests, et l'initialiser avec 13,5 millions de lignes dans la table `pgbench_accounts` :
>
> ```
> # en tant qu'utilisateur postgres
> createdb pgbench
> /usr/pgsql-16/bin/pgbench -i -s 135 pgbench
> ```

Noter que le nom de la base (ici en dernier paramètre) ne doit pas être précédé de `-d`.

La base obtenue fait 2 Go.

> Simuler l'utilisation de la base **pgbench** par 3 clients simultanés, chacun effectuant par exemple 1000 transactions :
>
> ```
> # en tant que postgres
> /usr/pgsql-16/bin/pgbench -c 3 -t 1000 -n -P1 pgbench
> ```

```
…
progress: 1.0 s, 1081.9 tps, lat 2.743 ms stddev 3.450, 0 failed
progress: 2.0 s, 1127.0 tps, lat 2.653 ms stddev 1.112, 0 failed
…
number of transactions per client: 1000
number of transactions actually processed: 3000/3000
number of failed transactions: 0 (0.000%)
latency average = 2.691 ms
latency stddev = 2.933 ms
initial connection time = 8.860 ms
tps = 1111.524022 (without initial connection time)
```

(Les chiffres peuvent bien sûr varier fortement selon la machine et la configuration.)

De vrais tests devraient être beaucoup plus longs !

> Simuler la même chose, en ajoutant le paramètre `-C` pour forcer une connexion à chaque transaction. Comparer la latence et le nombre de transactions par seconde.

L'exécution est ici beaucoup plus longue :

```
/usr/pgsql-16/bin/pgbench -c 3 -t 1000 -n -P1 -C pgbench

…
progress: 7.0 s, 360.8 tps, lat 6.083 ms stddev 1.569, 0 failed
progress: 8.0 s, 342.4 tps, lat 6.392 ms stddev 1.943, 0 failed
…
```

```
number of transactions actually processed: 3000/3000
number of failed transactions: 0 (0.000%)
latency average = 6.021 ms
latency stddev = 1.876 ms
average connection time = 1.855 ms
tps = 358.071199 (including reconnection times)
```

La connexion systématique a diminué les performances des deux tiers. Ce serait encore pire s'il y avait un réseau entre PostgreSQL et client `pgbench`.

> On veut simuler l'utilisation de la base **pgbench** par 3 utilisateurs effectuant 10 fois la sélection des comptes dont le solde est positif dans la table `pgbench_accounts`.
>
> Créer un fichier `query.sql` contenant la requête suivante :
>
> ```
> SELECT aid,bid,abalance
> FROM pgbench_accounts
> WHERE abalance > 0;
> ```
>
> Lancer la commande :
>
> ```
> /usr/pgsql-16/bin/pgbench -c 3 -t 10 -n -P1 -f query.sql pgbench
> ```

```
…
latency average = 1578.937 ms
latency stddev = 13.462 ms
initial connection time = 5.789 ms
tps = 1.899930 (without initial connection time)
```

On voit ainsi que `pgbench` permet de créer ses propres requêtes de test.

1.11.2 Influence de fsync et synchronous_commit

`pgbench` peut servir à mesurer l'impact d'un paramètre de configuration sur les performances du système.

> Positionner le paramètre `fsync` à `off` dans le fichier `postgresql.conf` et redémarrer PostgreSQL.

```
fsync = off
```

(Ne jamais faire cela en production, bien sûr.)

Relancer :

```
systemctl restart postgresql-16
```

> Simuler l'utilisation de la base **pgbench** par 3 clients simultanés, chacun effectuant par exemple 1000 transactions :
>
> ```
> /usr/pgsql-16/bin/pgbench -c 3 -t 1000 -n -P1 pgbench
> ```

> Comparer avec le premier test précédent.

```
…
latency average = 0.424 ms
latency stddev = 0.173 ms
initial connection time = 6.861 ms
tps = 6837.809439 (without initial connection time)
```

On est sur des valeurs beaucoup plus élevés que le premier test. La synchronisation à chaque `COMMIT` est très coûteuse et on l'a désactivée. Mais `fsync` à `off` provoquerait une corruption en cas d'arrêt brutal et ne peut être conservé.

> Rétablir `fsync` à `on`, puis relancer PostgreSQL.

```
fsync = on
```

```
sudo systemctl restart postgresql-16
```

> Refaire le même test avec `synchronous_commit` à `off` dans la session :
>
> ```
> PGOPTIONS='-c synchronous_commit=off' \
>         /usr/pgsql-16/bin/pgbench -c 3 -t 1000 pgbench
> ```

La variable d'environnement est un moyen de changer la configuration depuis le shell quand on ne peut changer la configuration de la session avec `SET`.

```
…
latency average = 0.473 ms
initial connection time = 13.312 ms
tps = 6344.453045 (without initial connection time)
```

Le paramètre `synchronous_commit` à `off` « regroupe » les synchronisations, et permet de récupérer une bonne partie des performances perdues avec `fsync` à `on`, si l'application le permet : en cas d'arrêt brutal, il reste le risque de perdre jusqu'à 600 ms de données pourtant committées (mais sans risque de corruption). Cette perte de données est parfois acceptable, parfois non. Ce choix peut être opéré transaction par transaction en changeant `synchronous_commit` dans la session.

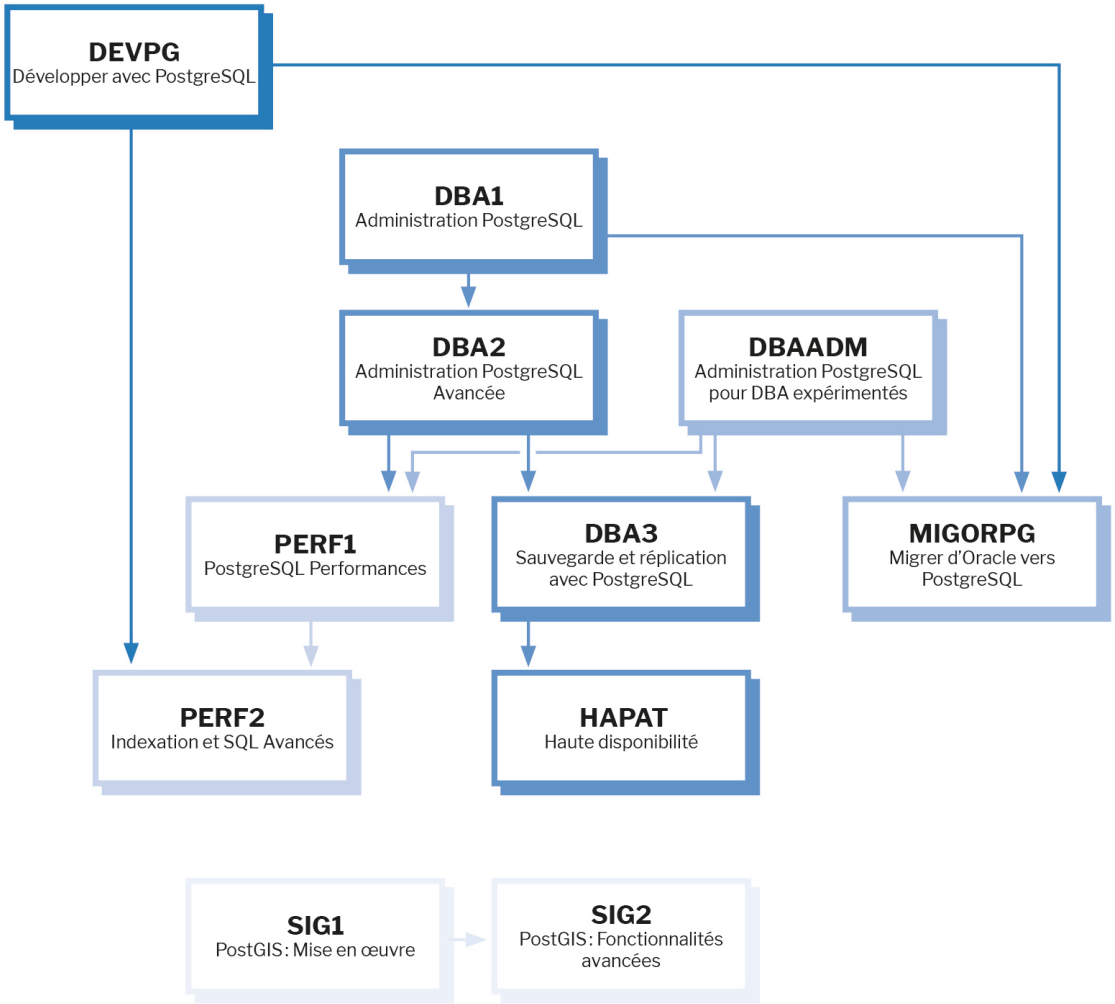
Noter à nouveau qu'un test rigoureux devrait durer beaucoup plus longtemps, sur plus de transactions, et tenir compte des effets de cache.

Les formations Dalibo

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

- DBA1 : Administration PostgreSQL
  https://dali.bo/dba1
- DBA2 : Administration PostgreSQL avancé
  https://dali.bo/dba2
- DBA3 : Sauvegarde et réplication avec PostgreSQL
  https://dali.bo/dba3
- DEVPG : Développer avec PostgreSQL
  https://dali.bo/devpg
- PERF1 : PostgreSQL Performances
  https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés
  https://dali.bo/perf2
- MIGORPG : Migrer d'Oracle à PostgreSQL
  https://dali.bo/migorpg
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

Les livres blancs

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.