Formation DBA4

PostgreSQL Performances

17.12

Dalibo SCOP

https://dalibo.com/formations

PostgreSQL Performances

Formation DBA4

TITRE : PostgreSQL Performances
SOUS-TITRE : Formation DBA4

REVISION: 17.12
DATE: 8 janvier 2018
ISBN: 979-10-97371-03-6

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, Sharon Bonan, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Virginie Jourdan, Guillaume Lelarge, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Flavie Perette, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Cédric Villemain, Thibaud Walkowiak

---

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions

Vous êtes autorisé :

- Partager, copier, distribuer et communiquer le matériel par tous moyens et sous tous formats
- Adapter, remixer, transformer et créer à partir du matériel

Dalibo ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence selon les conditions suivantes :

Attribution : Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que Dalibo vous soutient ou soutient la façon dont vous avez utilisé ce document.

Pas d'Utilisation Commerciale: Vous n'êtes pas autorisé à faire un usage commercial de ce document, tout ou partie du matériel le composant.

Partage dans les Mêmes Conditions : Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant le document original, vous devez diffuser le document modifié dans les même conditions, c'est à dire avec la même licence avec laquelle le document original a été diffusé.

Pas de restrictions complémentaires : Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser le document dans les conditions décrites par la licence.

Note: Ceci est un résumé de la licence.

https://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de plus de 12 ans 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 !

Contents

1 LICENCE CREATIVE COMMONS CC-BY-NC-SA

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

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

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

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

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

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

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

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

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

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

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

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

2 OPTIMISATIONS

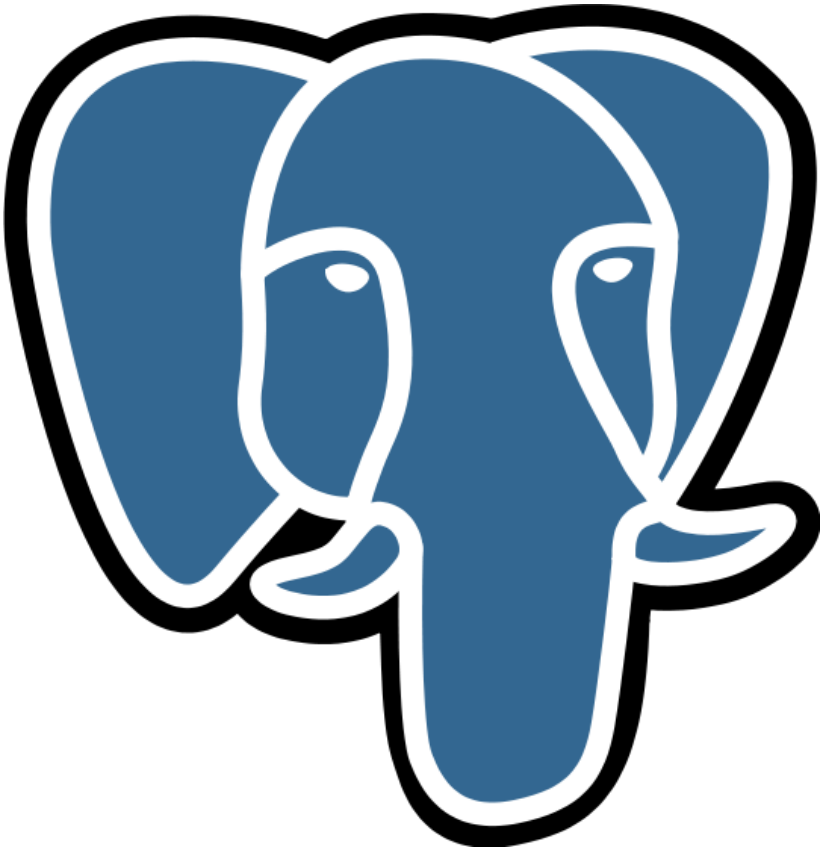


FIGURE 1: POSTGRESQL

2.1 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 appelé `tuning`) doit donc être réalisée sur tous ces éléments **à la fois** pour être optimal !

2.1.1 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

2.1.2 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.

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

2.2 MATÉRIEL

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

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

Au niveau du matériel, les composants essentiels sont :

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

2.2.1 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. Ce processus n'est pas multi-threadé. Par conséquent, chaque requête exécutée est traitée par un cœur de processeur. Plus vous voulez pouvoir exécuter de requêtes en parallèle, plus vous devez avoir de processeurs (ou plus exactement de cœurs). On considère habituellement qu'un cœur peut traiter de 4 à 20 requêtes simultanément. Cela dépend notamment beaucoup des requêtes, de leur complexité, de la quantité de 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'un SGBD pour une application web, il y a de fortes chances que le nombre de requêtes en parallèle soit assez élevé. Dans ce contexte, il faut prévoir un grand nombre de cœurs processeurs. En revanche, sur un entrepôt de données, nous trouvons habituellement peu d'utilisateurs avec des requêtes complexes et gourmandes en ressources. Dans ce cas, beaucoup de processeurs n'apporteront rien. Mieux vaut peu de cœur, mais que ces derniers soient plus puissants afin de répondre plus efficacement aux besoins importants de calculs complexe.

Ainsi, la fréquence (et donc la puissance) des processeurs est un point important à con-

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

À partir de la version 9.6, un processus exécutant une requête peut demander l'aide d'autre processus (appelés workers) pour l'aider à traiter cette requête. Les différents processus utiliseront des CPU différents, permettant ainsi une exécution parallélisée d'une requête. Ceci n'est possible qu'à partir de la version 9.6 et uniquement pour des requêtes en lecture seule. De plus, seules certaines actions sont parallélisables : parcours séquentiel, jointure, calcul d'agrégats. Ceci a un impact important pour les requêtes consommatrices en temps CPU. De ce fait, le facteur principal de choix reste toujours le nombre de CPU disponibles.

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 puissance 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.

2.2.2 RAM

- Essentiel 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.

2.2.3 DISQUES

- Trois grandes technologies : SATA, SAS et SSD

Technologie	Temps d'accès	Débit en lecture
RAM	~ 1 ns	~ 5 Go/s
Fusion IO	~ 0.015 ms	~ 2 Go/s
SSD	~ 0.1 ms	~ 200 Mo/s
SCSI 15ktpm	~ 1 ms	~ 100 Mo/s
SATA	~ 5 ms	~ 100 Mo/s

Il existe actuellement trois types de modèles de disques :

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

Les temps d'accès sont très importants pour un SGBD. Effectivement, ces derniers condi-

tionnent les performances des accès aléatoires, utilisés lors des parcours d'index. Le débit en lecture, lui, influe sur la rapidité de parcours des tables de façon séquentielle (bloc par bloc, de proche en proche) .

Il est immédiatement visible que la mémoire est toujours imbattable, y compris face aux disques SSD avec un facteur 100 000 en performance de temps d'accès entre les deux ! À l'autre bout de l'échelle se trouvent les disques SATA. Leur faible performance en temps d'accès ne doit pas pour autant les disqualifier. Leur prix est là aussi imbattable et il est souvent préférable de prendre un grand nombre de disques pour avoir de bonnes performances. Cependant, la fiabilité des disques SATA impose de les considérer comme des 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 (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 suivant leur utilisation. Par exemple les journaux de transactions sur un système disque, les tables sur un autre et les index sur un dernier.

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

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

Plus d'informations sont disponibles dans cet article[1] sur les disques SSD et Linux.

[1] http://lwn.net/Articles/408428/

17.12

Il existe aussi des supports de stockage moins courant, très onéreux, mais extrêmement rapides : ce sont les cartes 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. Leur utilisation reste cependant très limitée en raison du coût de cette technologie.

2.2.4 RAID

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

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

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

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

2.2.5 SAN

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

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

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

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

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

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

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

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

2.2.6 VIRTUALISATION : NON RECOMMANDÉ

- 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
- Mutualise les disques = problèmes de performances
  - Configurer les disques de PostgreSQL en « `Thick Provisionning` »

L'utilisation de machines virtuelles n'est pas recommandée avec PostgreSQL. En effet, la couche de virtualisation cache totalement les ressources physiques au système, ce qui rend l'investigation et l'optimisation des performances beaucoup plus difficiles qu'avec des serveurs physiques dédiés. Il est néanmoins possible d'utiliser des machines virtuelles avec PostgreSQL. Leur configuration doit alors être orientée vers la stabilité des performances. Cette configuration est complexe et difficile à suivre dans le temps. Les différentes parties de la plate-forme (virtualisation, système et bases de données) sont généralement administrées par des équipes techniques différentes, ce qui rend le 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, il est fortement conseillé de dédier un minimum de ressources aux VM PostgreSQL, et de superviser constamment l'overcommit du côté de l'hyperviseur pour éviter ce « trashing ».

Il est généralement conseillé d'utiliser au moins quatre cœurs physiques. En fonction de

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

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

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

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

2.2.7 VIRTUALISATION : LES BONNES PRATIQUES

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

Il est aussi recommandé d'utiliser la même source NTP sur les OS invité (VM) et l'hôte ESXi afin d'éviter l'effet dit de *time drifts*. Il faut être attentif à ce problème des tops d'horloge. Si une VM manque des tops d'horloges sous une forte charge ou autre raison, elle va percevoir le temps qui passe comme étant plus lent qu'il ne l'est réellement. Par exemple, un OS invité avec un top d'horloge à 1 ms attendra 1000 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 coeur d'un socket dispose de sa propre mémoire. Par exemple, avec un système à 16 cœurs et 128 Go de RAM, chaque cœur ou nœud possède 8 Go de mémoire physique.

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

Pour approfondir : Fiche KB préconisations pour VMWARE[2]

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

2.3 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.

2.3.1 CHOIX DU SYSTÈME D'EXPLOITATION

- PostgreSQL fonctionne sur différents systèmes
  - Linux, BSD, Windows, Solaris, HPUX, etc.
- Principalement développé et testé sous Linux
- Windows intéressant pour les postes des développeurs
  - mais moins performant que Linux
  - 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. Il est donc disponible sur la majorité des systèmes : Linux, BSD, Windows, Solaris, HPUX, etc. Cette portabilité est vérifiée en permanence avec la ferme de construction (BuildFarm, http://buildfarm.postgresql.org/).

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

Un autre système souvent utilisé est Windows. Ce dernier est très intéressant mais beaucoup moins performant avec PostgreSQL que Linux. Cela est principalement dû à sa ges-

tion assez mauvaise de la mémoire partagée. Cela a pour conséquence qu'il est difficile d'avoir un cache disque important pour PostgreSQL sous Windows.

De plus, vous ne pouvez pas démarrer PostgreSQL en tant que service si vous avez besoin de plus de 125 connexions pour des problématiques d'espace mémoire attribuée à un processus non-interactif. Le seul moyen de contourner ce problème sera de le lancer en mode interactif, depuis la ligne de commande. La limite théorique est alors repoussée à 750 connexions (plus d'information sur le wiki PostgreSQL[3]).

Sous Windows, il est fortement recommandé de placer le paramètre `update_process_title` à off pour obtenir de bonnes performances. D'ailleurs, c'est la valeur par défaut depuis la version 9.6 :

- Documentation PostgreSQL[4]
- Change the default of update_process_title to off[5]
- Improve PostgreSQL on Windows performance by 100%[6]

2.3.2 CHOIX DU NOYAU

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

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

[3] http://wiki.postgresql.org/wiki/Running_%26_Installing_PostgreSQL_On_Native_Windows#I_cannot_run_with_more_than_about_125_connections_at_once.2C_despite_having_capable_hardware
[4] https://www.postgresql.org/docs/current/static/runtime-config-logging.html#AEN35082
[5] https://www.postgresql.org/message-id/flat/0A3221C70F24FB45833433255569204D1F5BE3E8%40G01JPEXMBYT05#0A3221C70F24FB45833433255569204D1F5BE3E8@G01JPEXMBYT05
[6] http://www.openscg.com/2016/09/improve-postgresql-windows-performance-by-100/

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.

2.3.3 CONFIGURATION DU NOYAU

- En plus du choix du noyau, certains paramètres nécessitent une configuration personnalisée
  - gestion du cache disque système
  - gestion de la sur-allocation de mémoire
  - taille et comportement du swap
  - affinité entre les cœurs et les espaces mémoire
  - scheduler processeur
  - *huge pages*

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

2.3.4 CONTRÔLE DU CACHE DISQUE SYSTÈME

- Gestion de l'écriture des « dirty pages »
- Paramètres
  - vm.dirty_ratio
  - vm.dirty_background_ratio
  - vm.dirty_bytes
  - vm.dirty_background_bytes
- Plus nécessaire depuis la version 9.6 (`*_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 `vm.dirty_ratio`, `vm.dirty_background_ratio`, `vm.dirty_bytes` et `vm.dirty_background_bytes`.

`vm.dirty_ratio` exprime le pourcentage de pages mémoires modifiées à atteindre avant que les processus écrivent eux-mêmes les données du cache sur disque afin de les libérer. Ce comportement est à éviter. `vm.dirty_background_ratio` définit le pourcentage de pages mémoires modifiées forçant le noyau à commencer l'écriture les données du cache système en tâche de fond. Ce processus est beaucoup plus léger et à encourager. Ce dernier est alors seul à écrire alors que dans le premier cas, plusieurs processus tentent

de vider le cache système en même temps. Ce comportement provoque alors un encombrement de la bande passante des disques dans les situations de forte charge en écriture, surtout lors des opérations provoquant des synchronisations de données modifiées en cache sur le disque, comme l'appel à fsync. Fsync est utilisé par PostgreSQL lors des `CHECKPOINT`, ce qui peut provoquer des latences supplémentaires à ces moments-là.

Pour réduire les conséquences de ce phénomène, il est conseillé d'abaisser `vm.dirty_ratio` à 10 et `vm.dirty_background_ratio` à 5. Ainsi, lors de fortes charges en écriture, nous demandons au noyau de reporter plus régulièrement son cache disque sur l'espace de stockage mais pour une volumétrie plus faible. Ainsi, l'encombrement de la bande passante vers les disques sera moins 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. Ainsi, avec 32 Go de mémoire, ils représentent 640 Mo et 320 Mo de données à synchroniser, ce qui peut nécessiter plusieurs secondes d'écritures en fonction de la configuration disque utilisée.

Dans les cas plus extrêmes, 1 % de la mémoire représentent des volumétries trop importantes (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 octet et non plus en pourcentage de la mémoire disponible. Notez que ces paramètres ne sont pas complémentaires entre eux. Le dernier paramètre ayant été positionné prend le pas sur le précédent.

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

Depuis la version 9.6, ces options ne sont plus nécessaires grâce à ces paramètres :

- `bgwriter_flush_after` : Lorsque plus de `bgwriter_flush_after` octets sont écrits sur disque par le bgwriter, le moteur tente de forcer la synchronisation sur disque. 512 ko par défaut.
- `backend_flush_after` : 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 (désactive la synchronisation forcée).
- `wal_writer_flush_after` : quantité de donnée à partir de laquelle le WAL writer synchronise les blocs sur disque. 1 Mo par défaut.
- `checkpoint_flush_after` : Lorsque plus de `checkpoint_flush_after` octets sont écrits sur disque lors d'un checkpoint. Le moteur tente de forcer la synchronisation sur disque. 256 ko par défaut.

2.3.5 CONFIGURATION DU OOM

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

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

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

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

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

Attention, la valeur par défaut du paramètre `vm.overcommit_ratio` est 50. Ainsi, sur un système avec 32 Go de mémoire et 2 Go de swap, nous obtenons seulement 18 Go de mémoire allouable ! Ne pas oublier de modifier ce paramètre ; avec `vm.overcommit_ratio` positionné à 75, nous obtenons 26 Go de mémoire utilisable par les applications sur les 32 Go disponibles. Avoir un tel paramétrage permet de garantir qu'il y aura toujours au moins 20 % du total de la RAM disponible pour le cache disque, qui est très bénéfique à PostgreSQL.

2.3.6 CONFIGURATION DU SWAP

- Taille de la swap

    - pas plus de 2 Go
- Contrôler son utilisation
    - vm.swappiness

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

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

2.3.7 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

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

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

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

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

Sur une architecture NUMA (*Non Uniform Memory Access*), il existe une notion de distance entre les sockets processeurs et les "zones" mémoires (bancs de mémoire). La zone mémoire la plus proche d'un socket est alors définie comme sa zone "locale". Il est plus

coûteux pour les cœurs d'un processeur d'accéder aux zones mémoires distantes, ce qui implique des temps d'accès plus importants, et des débits plus faibles.

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

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

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

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

- avec `zone_reclaim_mode` à 1, temps de dump : 20 h, CPU utilisé par le COPY : 3 à 5 %
- avec `zone_reclaim_mode` à 0, temps de dump : 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[7]
- MySQL "swap insanity" problem[8]

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

2.3.8 CONFIGURATION DU SCHEDULER PROCESSEUR

- Réduire la propension du kernel à migrer les processus
  - `kernel.sched_migration_cost_ns = 5000000` (`sched_migration_cost` pour les noyaux <3.6)
- Désactiver le regroupement par session TTY
  - `kernel.sched_autogroup_enabled = 0`

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 à 5 ms (5 000 000 ns).

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.

2.3.9 HUGE PAGES

- Utiliser des pages mémoires de 2 Mo au lieu de 4 ko
- Réduction de la consommation mémoire des processus
- Garantie *Shared Buffers* non swappé
- `vm.nr_overcommit_hugepages=x`
- `huge_pages=on|off|try`

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 mémoires contiguës, il est plus économique d'utiliser des tailles de pages plus élevées : 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.

Permettre à PostgreSQL d'utiliser des *Huge Pages* réduit donc la consommation mémoire de chaque processus : en effet, chaque processus PostgreSQL dispose de sa propre table de pagination. Pour un *Shared Buffers* de 8 Go, chaque processus gaspille 16 Mo de mémoire rien que pour cette table, contre une centaine de ko pour des pages de 2 Mo. Cette mémoire pourra être utilisée à meilleur escient (`work_mem` par exemple, ou tout simplement du cache système).

Pour utiliser les *Huge Pages* :

- `huge_pages` doit être positionné à `try` (essayer, et utiliser des pages de 4 ko si le système n'arrive pas à fournir les pages de 2 Mo) ou `on` : exiger des *Huge Pages* ;
- `vm.nr_overcommit_hugepages` doit être suffisamment grand pour contenir les *Shared Buffers* et les autres zones mémoires 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). Sur un système hébergeant plusieurs instances, il faudra additionner toutes les zones mémoires de toutes les instances. La valeur de ce paramètre est en pages de la taille de *Huge Page* par défaut (valeur de `Hugepagesize` dans `/proc/meminfo`, habituellement 2 Mo).

Si vous souhaitez en apprendre plus sur le sujet des *Huge Pages*, un article détaillé est disponible dans la base de connaissances Dalibo[9] .

[9] https://kb.dalibo.com/huge_pages

2.3.10 COMMENT LES CONFIGURER

- Outil
  - sysctl
- Fichier de configuration
  - /etc/sysctl.conf

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

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

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

```
$ sysctl -p
```

2.3.11 CHOIX DU SYSTÈME DE FICHIERS

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

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. En règle générale, cela voudra dire le système ext4. Les systèmes reiserfs et jfs ne sont pratiquement plus développés et doivent dans tous les cas être évités. btrfs est encore au stade expérimental mais il est très prometteur. Enfin, XFS est un système qui semblait très intéressant pour les performances mais de nouveaux

tests ont montré que ext4 était souvent plus performant (voir notamment un comparatif ext4/xfs[10]).

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

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

NFS peut sembler intéressant, vu ses fonctionnalités. Cependant, ce système de fichiers est source de nombreux problèmes avec PostgreSQL. La documentation[11] l'indique très clairement :

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

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

Par contre, NFS est donc totalement déconseillé dans les environnements critiques avec PostgreSQL. Greg Smith, contributeur très connu, spécialisé dans l'optimisation de Post-

[10] http://blog.pgaddict.com/posts/postgresql-performance-on-ext4-and-xfs
[11] http://www.postgresql.org/docs/current/static/creating-cluster.html

greSQL, parle plus longuement des soucis de NFS avec PostgreSQL[12] . En fait, il y a des dizaines d'exemples de gens ayant eu des problèmes avec NFS. Les problèmes de performance sont quasi-systématiques, et les problèmes de fiabilité fréquents, et compliqués à diagnostiquer (comme illustré dans ce mail[13] , où le problème venait du noyau Linux).

2.3.12 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` :

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

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

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

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

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

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

[12] http://www.postgresql.org/message-id/4D2285CF.3050304@2ndquadrant.com
[13] http://www.postgresql.org/message-id/4D40DDB7.1010000@credativ.com

Les options `data=writeback` et `nobarrier` sont souvent cités comme optimisation potentielle. Le mode `writeback` de journalisation des ext3 et ext4 est à **éviter**. Effectivement, dans certains cas 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.

2.4 SERVEUR DE BASES DE DONNÉES

- Version
- Configuration
- Emplacement des fichiers

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

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

2.4.1 VERSION

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

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

Voici quelques exemples frappants :

- La version 9.0 dispose d'une optimisation du planificateur lui permettant de supprimer une jointure `LEFT JOIN` si elle est inutile pour l'obtention du résultat. C'est une optimisation particulièrement bienvenue pour tous les utilisateurs d'ORM.
- La version 9.1 dispose du SSI, pour `Serializable Snapshot Isolation`. Il s'agit d'une implémentation très performante du mode d'isolation sérialisée. Ce mode permet d'éviter l'utilisation des `SELECT FOR UPDATE`.
- La version 9.2 dispose d'un grand nombre d'améliorations du planificateur et des processus postgres qui en font une version exceptionnelle pour les performances, notamment les parcours d'index seuls.
- La version 9.6 propose la parallélisation de l'exécution de certaines requêtes.

Compiler soi-même PostgreSQL ne permet pas de gagner réellement en performance. Même s'il peut y avoir un gain, ce dernier ne peut être que mineur et difficilement identifiable. La compilation n'a un impact réellement identifié **que** sur les architecture Itanium (`IA-32` et `IA-64`) avec le compilateur propriétaire Intel (appelé `ICC`), *cf* le site d'Intel[14] .

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

- 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.

2.4.2 CONFIGURATION - MÉMOIRE

- shared_buffers
- wal_buffers
- work_mem
- maintenance_work_mem

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

`shared_buffers` permet de configurer la taille du cache disque de PostgreSQL. Chaque fois qu'un utilisateur veut extraire des données d'une table (par une requête `SELECT`) ou

[14] http://www.intel.com/cd/software/products/asmo-na/eng/compilers/clin/277618.htm

modifier les données d'une table (par exemple avec une requête `UPDATE`), PostgreSQL doit d'abord lire les lignes impliquées et les mettre dans son cache disque. Cette lecture prend du temps. Si ces lignes sont déjà dans le cache, l'opération de lecture n'est plus utile, ce qui permet de renvoyer plus rapidement les données à l'utilisateur. Ce cache est commun à tous les processus PostgreSQL, il n'existe donc qu'en un exemplaire. Généralement, il faut lui donner une grande taille, tout en conservant malgré tout la majorité de la mémoire pour le cache disque du système, à priori plus efficace pour de grosses quantités de données. Le pourcentage généralement préconisé est de 25 % de la mémoire totale pour un serveur dédié. Donc, par exemple, pour un serveur contenant 8 Go de mémoire, nous configurerons le paramètre `shared_buffers` à 2 Go. Néanmoins, on veillera à ne pas dépasser 8 Go. Des études ont montré que les performances décroissaient avec plus de mémoire.

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

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

probablement un problème ailleurs : des tris sur de trop gros volumes de données, une mémoire insuffisante, un manque d'index (utilisé pour les tris), etc. Des valeurs vraiment grandes ne sont valables que sur des systèmes d'infocentre.

Quant à `maintenance_work_mem`, il est aussi utilisé par chaque processus PostgreSQL réalisant une opération particulière : un VACUUM, une création d'index ou l'ajout d'une clé étrangère. Comme il est peu fréquent que ces opérations soient effectuées en simultané, la valeur de ce paramètre est très souvent bien supérieure à celle du paramètre `work_mem`. Sa valeur se situe fréquemment entre 128 Mo et 1 Go, voire plus.

2.4.3 CONFIGURATION - PLANIFICATEUR

- effective_cache_size
- random_page_cost

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

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

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

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

Soit 789 166 ko, résultat de l'addition de 190 580 (colonne `buffers`) et 598 536 (colonne `cached`).

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

2.4.4 CONFIGURATION - PARALLÉLISATION

- max_worker_processes
- max_parallel_workers
- max_parallel_workers_per_gather
- min_parallel_table_scan_size
- min_parallel_index_scan_size

À partir de la version 9.6, un processus PostgreSQL peut se faire aider d'autres processus pour exécuter une seule et même requête. Le nombre de processus utilisables pour une requête dépend de la valeur du paramètre `max_parallel_workers_per_gather` (à 2 par défaut). Si plusieurs processus veulent paralléliser l'exécution de leur requête, le nombre de processus d'aide ne pourra pas dépasser la valeur du paramètre `max_parallel_workers` (8 par défaut).

Il est à noter que ce nombre ne peut pas dépasser la valeur du paramètre `max_worker_processes` (par défaut à 16). De plus, avant la version 10, le paramètre `max_parallel_workers` n'existait pas et tout se basait sur le paramètre `max_worker_processes`.

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. Dans le cas d'un parcours, la parallélisation n'est possible que si la table ou l'index est suffisamment volumineux pour qu'une telle action soit intéressante au niveau des performances. Le volume déclencheur dépend de la valeur du paramètre `min_parallel_table_scan_size`, dont la valeur par défaut est de 8 Mo, pour une table et de la valeur du paramètre `min_parallel_index_scan_size`

pour un index (valeur par défaut, 512 Ko).

Voici comment le moteur détermine le nombre de worker à exécuter :

- 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 un worker
  - si T > Sx3 => On utilise un worker supplémentaire (2)
  - si T > Sx3x3 => On utilise deux workers supplémentaires (3)
  - si T > Sx3^3 => On utilise trois workers supplémentaires (4)
  - etc ...

2.4.5 CONFIGURATION - WAL

- fsync
- checkpoint_segments
  - remplacé par `min_wal_size` et `max_wal_size` à partir de la version 9.5
- checkpoint_timeout
- checkpoint_completion_target

`fsync` est le paramètre qui assure que les données sont non seulement écrites mais aussi forcées sur disque. En fait, quand PostgreSQL écrit dans des fichiers, cela passe par des appels 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, ce qui entraînera dans ce cas des incohérences dans les fichiers de données au redémarrage. Il est donc essentiel que les données enregistrées dans les journaux de transactions soient non seulement écrites mais que le noyau soit forcé de les écrire réellement sur disque. Cette opération s'appelle `fsync`. Par défaut, ce comportement est activé. Évidemment, cela coûte en performance mais ce que ça apporte en termes de fiabilité est essentiel. Il est donc obligatoire en production de conserver ce paramètre activé.

Chaque bloc modifié dans le cache disque de PostgreSQL doit être écrit sur disque au bout d'un certain temps. Ce temps dépend de deux paramètres `checkpoint_segments` et `checkpoint_timeout` . Le deuxième permet de s'assurer d'avoir au minimum un `CHECKPOINT` toutes les X minutes (5 par défaut). Tout surplus d'activité doit aussi être

géré. Un surplus d'activité engendrera des journaux de transactions supplémentaires. Le meilleur moyen dans ce cas est de préciser au bout de combien de journaux traités il faut lancer un `CHECKPOINT`. Cela se fait via le paramètre `checkpoint_segments`.

À partir de la version 9.5, le paramètre `checkpoint_segments` a été remplacé par deux paramètres :

- `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.

Le nom du paramètre `max_wal_size` peut porter à confusion. Le volume de WAL peut dépasser `max_wal_size` en cas de forte activité, ce n'est pas une valeur plafond.

2.4.6 CONFIGURATION - STATISTIQUES

- track_activities
- track_counts
- track_functions et track_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.

D'autres paramètres, désactivés par défaut, permettent d'aller plus loin. `track_functions` permet de récupérer des informations sur l'utilisation des procédures 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 `pg_stat_database` et `pg_stat_statements` et les plans d'exécutions appelés avec `EXPLAIN (ANALYZE,BUFFERS)`. Avant de l'activer sur une machine peu performante, vérifiez l'impact avec l'outil `pg_test_timing`.

2.4.7 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.

2.4.8 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[15]
- Il existe également pgconfig[16]

Le site du projet en ruby[17] 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, DW (pour DataWarehouse), mixed, etc.

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

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

[15] http://pgtune.leopard.in.ua/
[16] https://www.pgconfig.org/
[17] https://github.com/le0pard/pgtune

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.

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

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

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 (paramètre `wal_buffers`) est augmentée. Il faudra essayer de tout faire passer par les `CHECKPOINT`, d'où la valeur maximale pour `checkpoint_completion_target`. Quant à `shared_buffers` et `effective_cache_size`, ils restent aux valeurs définies ci-dessus (respectivement un quart et deux tiers de la mémoire).

Et enfin avec un profil entrepôt de données (*datawarehouse*) :

```
max_connections = 20
shared_buffers = 2GB
effective_cache_size = 6GB
work_mem = 52428kB
```

```
maintenance_work_mem = 1GB
min_wal_size = 4GB
max_wal_size = 8GB
checkpoint_completion_target = 0.9
wal_buffers = 16MB
default_statistics_target = 500
```

Pour un entrepôt de données, il y a généralement peu d'utilisateurs à un instant `t`. Par contre, ils exécutent des requêtes complexes sur une grosse volumétrie. Du coup, la configuration change en profondeur cette fois. Le paramètre `max_connections` est diminué très fortement. Cela permet d'allouer beaucoup de mémoire aux tris et hachages (paramètre `work_mem` à 50 Mo). Les entrepôts de données ont souvent des scripts d'import de données (batchs). Cela nécessite de pouvoir écrire rapidement de grosses quantités de données, autrement dit une augmentation conséquente du paramètre `wal_buffers` et 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.

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

2.4.9 OUTIL PGBENCH

- Outil pour réaliser rapidement des tests de performance
- Fourni dans les modules de "contrib" de PostgreSQL
- Travail 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.

pgbench est en fort développement ces derniers temps. La version 9.5 apporte de nombreuses nouvelles fonctionnalités pour cet outil.

2.4.10 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.

2.4.11 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

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

```
$ 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)
```

2.4.12 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. Chaque requête doit être écrite sur **UNE** seule ligne, un script peut contenir plusieurs requêtes. Toutes les requêtes du fichier seront exécutées dans leur ordre d'apparition. Si plusieurs scripts SQL sont indiqués, chaque transaction sélectionne le fichier à exécuter de façon aléatoire. Enfin, il est possible d'utiliser des variables dans vos scripts SQL afin de faire varier le 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;
```

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

2.4.13 EMPLACEMENT DES FICHIERS DE DONNÉES

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

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

```
$ mkdir /opt/tablespace1
$ chown postgres:postgres /opt/tablespace1
$ chmod 700 /opt/tablespace1
```

```
$ psql postgres
postgres =# CREATE TABLESPACE grosdisque LOCATION '/opt/tablespace1';
postgres =# ALTER TABLE t1 SET TABLESPACE grosdisque;
```

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

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

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

2.4.14 EMPLACEMENT DES JOURNAUX DE TRANSACTIONS

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

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

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

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

Il est aussi possible de faire en sorte que la commande initdb le fasse elle-même. Pour cela, il faut utiliser l'option -X :

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

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

2.4.15 EMPLACEMENT DES FICHIERS STATISTIQUES

- Placer les fichiers statistiques sur un autre disque
  - et de préférence sur un montage en RAM
- Option stats_temp_directory

PostgreSQL met à disposition différents compteurs statistiques via des vues. Ces vues utilisent des métriques stockées dans des fichiers de statistiques, mis à jour par le processus `stats collector`. Ces fichiers 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 entraîner une forte activité disque. Dans de tels cas, le processus `stats collector` apparaît parmi les processus les plus consommateurs d'I/O avec `iotop`.

Lorsque le problème se pose, il est recommandé de déplacer ces fichiers dans un RAM-disk. Cette opération peut être réalisée à chaud en suivant la procédure suivante. Attention cependant, depuis PostgreSQL 9.4, le module `pg_stat_statements` 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.

Voici la procédure à suivre pour mettre en place un RAM-disk pour le répertoire des fichiers statistiques :

- création du point de montage

```
mkdir /var/lib/postgresql/10/pg_stat_tmpfs
```

- création et montage du système de fichiers

```
mount -o auto,nodev,nosuid,noexec,noatime,mode=0700,size=64M,
      uid=<postgres-uid>,gid=<postgres-gid>
      -t tmpfs tmps /var/lib/postgresql/10/pg_stat_tmpfs
```

- modification de la configuration PostgreSQL

```
stats_temp_directory = '/var/lib/postgresql/10/data/pg_stat_tmpfs'
```

- recharger la configuration de PostgreSQL
- ajouter au fichier /etc/fstab la ligne suivante

```
tmpfs /var/lib/postgresql/10/data/pg_stat_tmpfs tmpfs auto,nodev,nosuid,noexec,
      noatime,uid=<postgres-uid>,gid=<postgres-gid>,mode=0700,size=64M  0 0
```

Les informations <postgres-uid> et <postgres-gid> doivent être remplacées suivant les UID et GID du système.

Le point de montage employé doit être placé à l'extérieur du PGDATA, tout comme les répertoires des tablespaces.

2.4.16 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

Voici un exemple de sortie :

```
Connecting to localhost:5432 database template1 with user postgres...
[OK]      User used for report have super rights
=====  OS information  =====
[INFO]    OS: Debian GNU/Linux 6.0
[INFO]    OS total memory: 7.81 GB
[BAD]     Memory overcommitment is allowed on the system.
          This can lead to OOM Killer killing some PostgreSQL process,
          which will cause a PostgreSQL server restart (crash recovery)
[INFO]    sysctl vm.overcommit_ratio=50
[BAD]     vm.overcommit_memory is too small, you will not be able to use more
          than 50*RAM+SWAP for applications
[INFO]    Running on physical machine
[INFO]    Currently used I/O scheduler(s) : cfq
=====  General instance informations  =====
-----  Version  -----
[WARN]    You are using version 9.3.11 which is not the latest version
-----  Uptime  -----
[INFO]    Service uptime : 290d 04h 56m 12s
-----  Databases  -----
[INFO]    Database count (except templates): 5
[INFO]    Database list (except templates): postgres dotclear maizeweeds
```

```
          roundcubemail ttrss
-----  Extensions  -----
[INFO]    Number of activated extensions : 1
[INFO]    Activated extensions : plpgsql
[WARN]    Extensions pg_stat_statements is disabled
-----  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
[INFO]    current used connections: 1 (1.00%)
[INFO]    3 are reserved for super user (3.00%)
[INFO]    Average connection age :  00s
[BAD]     Average connection age is less than 1 minute.
          Use a connection pooler to limit new connection/seconds
-----  Memory usage  -----
[INFO]    configured work_mem: 1024.00 KB
[INFO]    Using an average ratio of work_mem buffers by connection of 150%
          (use --wmp to change it)
[INFO]    total work_mem (per connection): 1.50 MB
[INFO]    shared_buffers: 128.00 MB
[INFO]    Track activity reserved size : 103.00 KB
[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 * 1024.00 KB *
        + autovacuum_max_workers * maintenance_work_mem (3 * 16.00 MB = 48.00 MB)
        + track activity size (103.00 KB)
        = 326.10 MB
[INFO]    effective_cache_size: 128.00 MB
[INFO]    Size of all databases : 444.07 MB
[INFO]    PostgreSQL maximum memory usage: 4.08% 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 5.68% of total RAM
[WARN]    Increase shared_buffers and/or effective_cache_size to use more memory
-----  Logs  -----
[OK]      log_hostname is off : no reverse DNS lookup latency
[WARN]    log of long queries is desactivated. 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  -----
[BAD]     The wal_level minimal does not allow PITR backup and recovery
-----  Planner  -----
[OK]      costs settings are defaults
[OK]      all plan features are enabled
=====  Database information for database template1  =====
-----  Database size  -----
[INFO]    Database template1 total size : 6.32 MB
[INFO]    Database template1 tables size : 3.93 MB (62.18%)
[INFO]    Database template1 indexes size : 2.39 MB (37.82%)
-----  Shared buffer hit rate  -----
[INFO]    shared_buffer_heap_hit_rate: 99.70%
[INFO]    shared_buffer_toast_hit_rate: 0.00%
[INFO]    shared_buffer_tidx_hit_rate: 0.00%
[INFO]    shared_buffer_idx_hit_rate: 99.81%
[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 advices  =====
-----  backup  -----
[URGENT] Configure your wal_level to a level which allow PITR backup and recovery
-----  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 to collect statistics on all queries
      (not only queries longer than log_min_duration_statement in logs)
-----  sysctl  -----
[URGENT] set vm.overcommit_memory=2 in /etc/sysctl.conf and run sysctl -p to reload it.
   This will disable memory overcommitment and avoid postgresql killed by OOM killer.
-----  version  -----
[LOW] Upgrade to last version
```

2.5 CONCLUSION

PostgreSQL propose de nombreuses voies d'optimisation.

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

2.5.1 QUESTIONS

N'hésitez pas, c'est le moment !

2.6 TRAVAUX PRATIQUES

2.6.1 SCHÉMA DE LA BASE CAVE

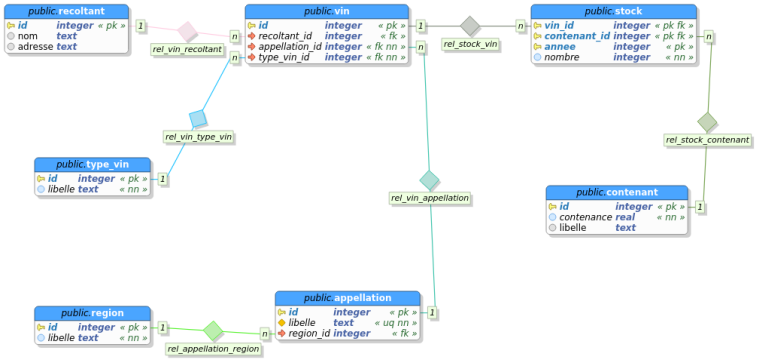


FIGURE 2: SCHÉMA DE LA BASE CAVE

2.6.2 ÉNONCÉS

Utilisation de pgbench

- Vérifier que le paquet correspondant aux modules de contrib de PostgreSQL est installé.

- Créer une base `bench` sur laquelle nous effectuerons nos premiers tests. L'initialiser avec 1 million de lignes dans la table `pgbench_accounts`.
- Simuler l'utilisation de la base `bench` par 3 clients simultanés, chacun effectuant 20 transactions.
- Simuler l'utilisation de la base `bench` par 3 clients simultanés, utilisant une connexion à chaque fois, et effectuant chacun 20 transactions.
- Simuler l'utilisation de la base `cave` par 3 utilisateurs effectuant 50 fois la sélection des vins de la région « Bourgogne », en un seul ordre.
- Positionner le paramètre `fsync` à « off » dans le fichier `postgresql.conf` et relancer le serveur. Simuler l'utilisation de la base `bench` par 3 clients simultanés, chacun effectuant 20 transactions.
- Remettre `fsync` à « on » et passer `synchronous_commit` à « off », et relancer le serveur. Refaire le même test.

2.6.3 SOLUTIONS

Utilisation de pgbench

- Créer une base pour `pgbench` :

```
$ createdb bench
```

Initialiser la base `bench` avec les données de test :

```
$ pgbench -i -s 10 bench
```

- Simuler l'utilisation de la base `bench` par 3 clients simultanés, chacun effectuant 20 transactions :

```
$ pgbench -c 3 -t 20 bench
```

- Simuler l'utilisation de la base `bench` par 3 clients simultanés, utilisant une connexion à chaque fois, et effectuant chacun 20 transactions :

```
$ pgbench -c 3 -t 20 -C bench
```

- Simuler l'utilisation de la base `cave` par 3 utilisateurs effectuant 50 fois la sélection des vins de la région « Bourgogne », en un seul ordre.

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

```
SELECT vin.id, appellation.libelle, type_vin.libelle
FROM vin, appellation, type_vin, region
WHERE vin.type_vin_id = type_vin.id
  AND vin.appellation_id = appellation.id
```

```
  AND appellation.region_id = region.id
  AND region.libelle = 'Bourgogne';
```

Puis lancer la commande :

```
pgbench -c 3 -t 50 -N -n -f query.sql cave
```

L'option `-N` sert à désactiver la mise à jour des tables `branches` et `tellers` puisqu'elles n'existent pas dans notre base `cave`. L'option `-n` est utilisée pour ne pas tenter de nettoyer la table `history`.

`pgbench` peut servir à regarder l'impact d'un paramètre de configuration sur les performances du système. Positionner le paramètre `fsync` à « off » dans le fichier `postgresql.conf` et relancer le serveur.

Puis, simuler l'utilisation de la base `bench` par 3 clients simultanés, chacun effectuant 20 transactions :

```
$ pgbench -c 3 -t 20 bench
```

- Repositionner `fsync` à « on », et passer `synchronous_commit` à « off ». Refaire le test.

3 COMPRENDRE EXPLAIN

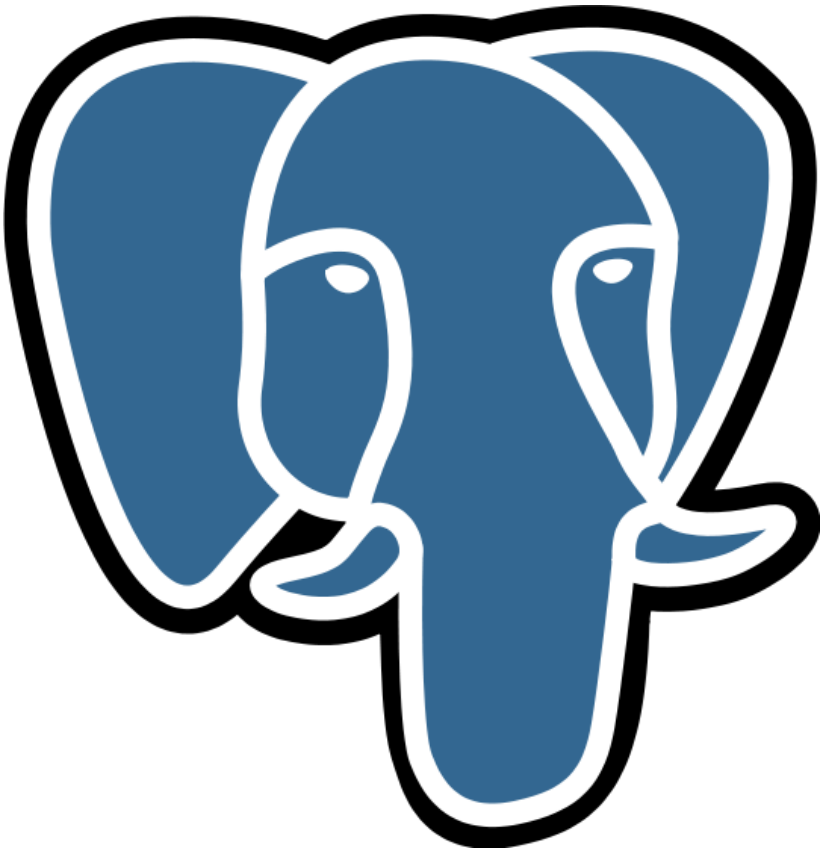


FIGURE 3: POSTGRESQL

3.1 INTRODUCTION

- Le matériel, le système et la configuration sont importants pour les performances
- Mais il est aussi essentiel de se préoccuper des requêtes et de leurs performances

Face à un problème de performances, l'administrateur se retrouve assez rapidement face à une (ou plusieurs) requête(s). Une requête en soi représente très peu d'informations.

17.12

Suivant la requête, des dizaines de plans peuvent être sélectionnés pour l'exécuter. Il est donc nécessaire de pouvoir trouver le plan d'exécution et de comprendre ce plan. Cela permet de mieux appréhender la requête et de mieux comprendre les pistes envisageables pour la corriger.

3.1.1 AU MENU

- Exécution globale d'une requête
- Planificateur : utilité, statistiques et configuration
- EXPLAIN
- Nœuds d'un plan
- Outils

Avant de détailler le fonctionnement du planificateur, nous allons regarder la façon dont une requête s'exécute globalement. Ensuite, nous aborderons le planificateur : en quoi est-il utile, comment fonctionne-t-il, et comment le configurer. Nous verrons aussi l'ensemble des opérations utilisables par le planificateur. Enfin, nous expliquerons comment utiliser `EXPLAIN` ainsi que les outils essentiels pour faciliter la compréhension d'un plan de requête.

Tous les exemples proposés ici viennent d'une version 9.1.

3.2 EXÉCUTION GLOBALE D'UNE REQUÊTE

- L'exécution peut se voir sur deux niveaux
  - Niveau système
  - Niveau SGBD
- De toute façon, composée de plusieurs étapes

L'exécution d'une requête peut se voir sur deux niveaux :

- ce que le système perçoit ;
- ce que le SGBD fait.

Dans les deux cas, cela va nous permettre de trouver les possibilités de lenteurs dans l'exécution d'une requête par un utilisateur.

3.2.1 NIVEAU SYSTÈME

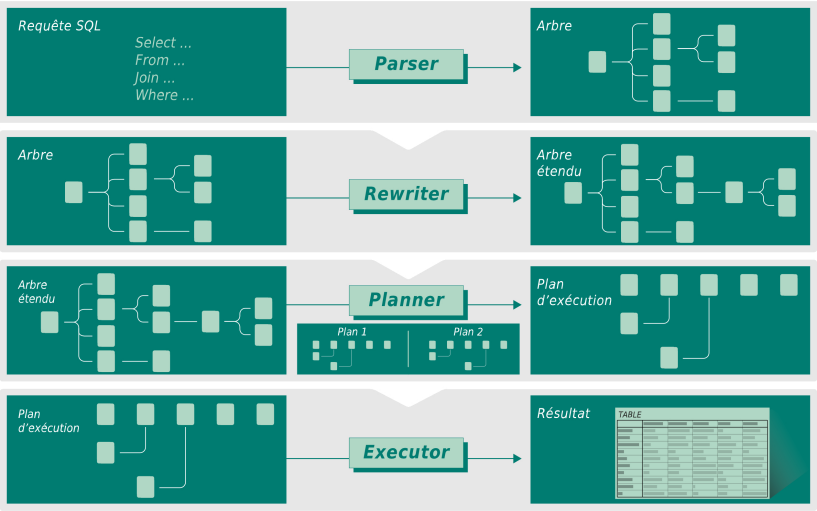
- Le client envoie une requête au serveur de bases de données
- Le serveur l'exécute
- Puis il renvoie le résultat au client

PostgreSQL est un système client-serveur. L'utilisateur se connecte via un outil (le client) à une base d'une instance PostgreSQL (le serveur). L'outil peut envoyer une requête au serveur, celui-ci l'exécute et finit par renvoyer les données résultant de la requête ou le statut de la requête.

Généralement, l'envoi de la requête est rapide. Par contre, la récupération des données peut poser problème si une grosse volumétrie est demandée sur un réseau à faible débit.

3.2.2 NIVEAU SGBD

TRAITEMENT D'UNE REQUÊTE SQL



Lorsque le serveur récupère la requête, un ensemble de traitements est réalisé :

- le `parser` va réaliser une analyse syntaxique de la requête
- le `rewriter` va réécrire, si nécessaire la requête

- pour cela, il prend en compte les règles et vues
- si une règle demande de changer la requête, la requête envoyée est remplacée par la nouvelle
- si une vue est utilisée, la requête qu'elle contient est intégrée dans la requête envoyée
- le `planner` va générer l'ensemble des plans d'exécutions
- il calcule le coût de chaque plan
- puis il choisit le plan le moins coûteux, donc le plus intéressant
- l' `executer` exécute la requête
- pour cela, il doit commencer par récupérer les verrous nécessaires sur les objets ciblés
- une fois les verrous récupérés, il exécute la requête
- une fois la requête exécutée, il envoie les résultats à l'utilisateur

Plusieurs goulets d'étranglement sont visibles ici. Les plus importants sont :

- la planification (à tel point qu'il est parfois préférable de ne générer qu'un sous-ensemble de plans, pour passer plus rapidement à la phase d'exécution) ;
- la récupération des verrous (une requête peut attendre plusieurs secondes, minutes, voire heures avant de récupérer les verrous et exécuter réellement la requête) ;
- l'exécution de la requête ;
- l'envoi des résultats à l'utilisateur.

Il est possible de tracer l'exécution des différentes étapes grâce aux options `log_parser_stats`, `log_planner_stats` et `log_executor_stats`. Voici un exemple complet :

- Mise en place de la configuration sur la session :

```
b1=# SET log_parser_stats TO on;
b1=# SET log_planner_stats TO on;
b1=# SET log_executor_stats TO on;
b1=# SET client_min_messages TO log;
```

- Exécution de la requête :

```
b1=# SELECT * FROM t1 WHERE id=10;
```

- Trace du `parser`

```
LOG:  PARSER STATISTICS
DETAIL:  ! system usage stats:
! 0.000051 elapsed 0.000000 user 0.000000 system sec
! [0.017997 user 0.021996 sys total]
! 0/0 [13040/248] filesystem blocks in/out
! 0/0 [40/1589] page faults/reclaims, 0 [0] swaps
```

```
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 0/0 [167/6] voluntary/involuntary context switches
LOG:  PARSE ANALYSIS STATISTICS
DETAIL:  ! system usage stats:
! 0.000197 elapsed 0.001000 user 0.000000 system sec
! [0.018997 user 0.021996 sys total]
! 0/0 [13040/248] filesystem blocks in/out
! 0/1 [40/1590] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 0/0 [167/6] voluntary/involuntary context switches
```

- Trace du `rewriter`

```
LOG:  REWRITER STATISTICS
DETAIL:  ! system usage stats:
! 0.000007 elapsed 0.000000 user 0.000000 system sec
! [0.018997 user 0.021996 sys total]
! 0/0 [13040/248] filesystem blocks in/out
! 0/0 [40/1590] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 0/0 [167/6] voluntary/involuntary context switches
```

- Trace du `planner`

```
LOG:  PLANNER STATISTICS
DETAIL:  ! system usage stats:
! 0.000703 elapsed 0.000000 user 0.000000 system sec
! [0.018997 user 0.021996 sys total]
! 0/0 [13040/248] filesystem blocks in/out
! 0/6 [40/1596] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 0/0 [167/6] voluntary/involuntary context switches
```

- Trace du `executer`

```
LOG:  EXECUTOR STATISTICS
DETAIL:  ! system usage stats:
! 0.078548 elapsed 0.000000 user 0.000000 system sec
! [0.019996 user 0.021996 sys total]
! 16/0 [13056/248] filesystem blocks in/out
! 0/2 [40/1599] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 1/0 [168/6] voluntary/involuntary context switches
```

3.2.3 EXCEPTIONS

- Requêtes DDL
- Instructions TRUNCATE et COPY
- Pas de réécriture, pas de plans d'exécution... une exécution directe

Il existe quelques requêtes qui échappent à la séquence d'opérations présentées précédemment. Toutes les opérations DDL (modification de la structure de la base), les instructions `TRUNCATE` et `COPY` (en partie) sont vérifiées syntaxiquement, puis directement exécutées. Les étapes de réécriture et de planification ne sont pas réalisées.

Le principal souci pour les performances sur ce type d'instructions est donc l'obtention des verrous et l'exécution réelle.

3.3 QUELQUES DÉFINITIONS

- Prédicat
  - filtre de la clause `WHERE`
- Sélectivité
  - pourcentage de lignes retournées après application d'un prédicat
- Cardinalité
  - nombre de lignes d'une table
  - nombre de lignes retournées après filtrage

Un prédicat est une condition de filtrage présente dans la clause `WHERE` d'une requête. Par exemple `colonne = valeur`.

La sélectivité est liée à l'application d'un prédicat sur une table. Elle détermine le nombre de lignes remontées par la lecture d'une relation suite à l'application d'une clause de filtrage, ou prédicat. Elle peut être vue comme un coefficient de filtrage d'un prédicat. La sélectivité est exprimée sous la forme d'un pourcentage. Pour une table de 1000 lignes, si la sélectivité d'un prédicat est de 10%, la lecture de la table en appliquant le prédicat devrait retourner 100 lignes.

La cardinalité représente le nombre de lignes d'une relation. En d'autres termes, la cardinalité représente le nombre de lignes d'une table ou du résultat d'une fonction. Elle représente aussi le nombre de lignes retourné par la lecture d'une table après application d'un ou plusieurs prédicats.

3.3.1 REQUÊTE ÉTUDIÉE

Cette requête d'exemple :

```
SELECT matricule, nom, prenom, nom_service, fonction, localisation
  FROM employes emp
  JOIN services ser ON (emp.num_service = ser.num_service)
 WHERE ser.localisation = 'Nantes';
```

Cette requête permet de déterminer quels sont les employés basés à Nantes.

Le script suivant permet de recréer le jeu d'essai :

```
CREATE TABLE services (
    num_service integer primary key,
    nom_service character varying(20),
    localisation character varying(20)
);

CREATE TABLE employes (
    matricule     integer primary key,
    nom           varchar(15) not null,
    prenom        varchar(15) not null,
    fonction      varchar(20) not null,
    manager       integer,
    date_embauche date,
    num_service   integer not null references services (num_service)
);

INSERT INTO services VALUES (1, 'Comptabilité', 'Paris');
INSERT INTO services VALUES (2, 'R&D', 'Rennes');
INSERT INTO services VALUES (3, 'Commerciaux', 'Limoges');
INSERT INTO services VALUES (4, 'Consultants', 'Nantes');

INSERT INTO employes VALUES
    (33, 'Roy', 'Arthur', 'Consultant', 105, '2000-06-01', 4);
INSERT INTO employes VALUES
    (81, 'Prunelle', 'Léon', 'Commercial', 97, '2000-06-01', 3);
INSERT INTO employes VALUES
    (97, 'Lebowski', 'Dude', 'Responsable', 104, '2003-01-01', 3);
INSERT INTO employes VALUES
    (104, 'Cruchot', 'Ludovic', 'Directeur Général', NULL, '2005-03-06', 3);
INSERT INTO employes VALUES
    (105, 'Vacuum', 'Anne-Lise', 'Responsable', 104, '2005-03-06', 4);
INSERT INTO employes VALUES
    (119, 'Thierrie', 'Armand', 'Consultant', 105, '2006-01-01', 4);
INSERT INTO employes VALUES
    (120, 'Tricard', 'Gaston', 'Développeur', 125, '2006-01-01', 2);
```

```
INSERT INTO employes VALUES
    (125, 'Berlicot', 'Jules', 'Responsable', 104, '2006-03-01', 2);
INSERT INTO employes VALUES
    (126, 'Fougasse', 'Lucien', 'Comptable', 128, '2006-03-01', 1);
INSERT INTO employes VALUES
    (128, 'Cruchot', 'Joséphа', 'Responsable', 105, '2006-03-01', 1);
INSERT INTO employes VALUES
    (131, 'Lareine-Leroy', 'Émilie', 'Développeur', 125, '2006-06-01', 2);
INSERT INTO employes VALUES
    (135, 'Brisebard', 'Sylvie', 'Commercial', 97, '2006-09-01', 3);
INSERT INTO employes VALUES
    (136, 'Barnier', 'Germaine', 'Consultant', 105, '2006-09-01', 4);
INSERT INTO employes VALUES
    (137, 'Pivert', 'Victor', 'Consultant', 105, '2006-09-01', 4);
```

3.3.2 PLAN DE LA REQUÊTE ÉTUDIÉE

L'objet de ce module est de comprendre son plan d'exécution :

```
Hash Join  (cost=1.06..2.29 rows=4 width=48)
  Hash Cond: (emp.num_service = ser.num_service)
  ->  Seq Scan on employes emp  (cost=0.00..1.14 rows=14 width=35)
  ->  Hash  (cost=1.05..1.05 rows=1 width=21)
        ->  Seq Scan on services ser  (cost=0.00..1.05 rows=1 width=21)
              Filter: ((localisation)::text = 'Nantes'::text)
```

La directive `EXPLAIN` permet de connaître le plan d'exécution d'une requête. Elle permet de savoir par quelles étapes va passer le SGBD pour répondre à la requête.

3.4 PLANIFICATEUR

- Chargé de sélectionner le meilleur plan d'exécution
- Énumère tous les plans d'exécution
  - Tous ou presque...
- Calcule leur coût suivant des statistiques, un peu de configuration et beaucoup de règles
- Sélectionne le meilleur (le moins coûteux)

Le but du planificateur est assez simple. Pour une requête, il existe de nombreux plans d'exécution possibles. Il va donc énumérer tous les plans d'exécution possibles (sauf

si cela représente vraiment trop de plans auquel cas, il ne prendra en compte qu'une partie des plans possibles). Il calcule ensuite le coût de chaque plan. Pour cela, il dispose d'informations sur les données (des statistiques), d'une configuration (réalisée par l'administrateur de bases de données) et d'un ensemble de règles inscrites en dur. Une fois tous les coûts calculés, il ne lui reste plus qu'à sélectionner le plan qui a le plus petit coût.

3.4.1 UTILITÉ

- SQL est un langage déclaratif
- Une requête décrit le résultat à obtenir
  - Mais pas la façon de l'obtenir
- C'est au planificateur de déduire le moyen de parvenir au résultat demandé

Le planificateur est un composant essentiel d'un moteur de bases de données. Les moteurs utilisent un langage SQL qui permet à l'utilisateur de décrire le résultat qu'il souhaite obtenir. Par exemple, s'il veut récupérer des informations sur tous les clients dont le nom commence par la lettre A en triant les clients par leur département, il pourrait utiliser une requête du type :

```
SELECT * FROM clients WHERE nom LIKE 'A%' ORDER BY departement;
```

Un moteur de bases de données peut récupérer les données de plusieurs façons :

- faire un parcours séquentiel de la table `clients` en filtrant les enregistrements d'après leur nom, puis trier les données grâce à un algorithme ;
- faire un parcours d'index sur la colonne nom pour trouver plus rapidement les enregistrements de la table `clients` satisfaisant le filtre 'A%', puis trier les données grâce à un algorithme ;
- faire un parcours d'index sur la colonne département pour récupérer les enregistrements déjà triés, et ne retourner que ceux vérifiant nom like 'A%'

Et ce ne sont que quelques exemples car il serait possible d'avoir un index utilisable pour le tri et le filtre par exemple.

Donc la requête décrit le résultat à obtenir, et le planificateur va chercher le meilleur moyen pour parvenir à ce résultat.

Pour ce travail, il dispose d'un certain nombre d'opérateurs. Ces opérateurs travaillent sur des ensembles de lignes, généralement un ou deux. Chaque opérateur renvoie un seul ensemble de lignes. Le planificateur peut combiner ces opérations suivant certaines règles. Un opérateur peut renvoyer l'ensemble de résultats de deux façons : d'un coup

(par exemple le tri) ou petit à petit (par exemple un parcours séquentiel). Le premier cas utilise plus de mémoire, et peut nécessiter d'écrire des données temporaires sur disque. Le deuxième cas aide à accélérer des opérations comme les curseurs, les sous-requêtes `IN` et `EXISTS`, la clause `LIMIT`, etc.

3.4.2 RÈGLES

- 1ère règle : Récupérer le bon résultat
- 2è règle : Le plus rapidement possible
  - En minimisant les opérations disques
  - En préférant les lectures séquentielles
  - En minimisant la charge CPU
  - En minimisant l'utilisation de la mémoire

Le planificateur suit deux règles :

- il doit récupérer le bon résultat ;
- il doit le récupérer le plus rapidement possible.

Cette deuxième règle lui impose de minimiser l'utilisation des ressources : en tout premier lieu les opérations disques vu qu'elles sont les plus coûteuses, mais aussi la charge CPU et l'utilisation de la mémoire. Dans le cas des opérations disques, s'il doit en faire, il doit absolument privilégier les opérations séquentielles aux opérations aléatoires (qui demandent un déplacement de la tête de disque, ce qui est l'opération la plus coûteuse sur les disques magnétiques).

3.4.3 OUTILS DE L'OPTIMISEUR

- L'optimiseur s'appuie sur :
  - un mécanisme de calcul de coûts
  - des statistiques sur les données
  - le schéma de la base de données

Pour déterminer le chemin d'exécution le moins coûteux, l'optimiseur devrait connaître précisément les données mises en œuvre dans la requête, les particularités du matériel et la charge en cours sur ce matériel. Cela est impossible. Ce problème est contourné en utilisant deux mécanismes liés l'un à l'autre :

- un mécanisme de calcul de coût de chaque opération,

- des statistiques sur les données.

Pour quantifier la charge nécessaire pour répondre à une requête, PostgreSQL utilise un mécanisme de coût. Il part du principe que chaque opération a un coût plus ou moins important. Les statistiques sur les données permettent à l'optimiseur de requêtes de déterminer assez précisément la répartition des valeurs d'une colonne d'une table, sous la forme d'histogramme. Il dispose encore d'autres informations comme la répartition des valeurs les plus fréquentes, le pourcentage de `NULL`, le nombre de valeurs distinctes, etc… Toutes ces informations aideront l'optimiseur à déterminer la sélectivité d'un filtre (prédicat de la clause `WHERE`, condition de jointure) et donc quelle est la quantité de données récupérées par la lecture d'une table en utilisant le filtre évalué. Enfin, l'optimiseur s'appuie sur le schéma de la base de données afin de déterminer différents paramètres qui entrent dans le calcul du plan d'exécution : contrainte d'unicité sur une colonne, présence d'une contrainte `NOT NULL`, etc.

3.4.4 OPTIMISATIONS

- À partir du modèle de données
  - suppression de jointures externes inutiles
- Transformation des sous-requêtes
  - certaines sous-requêtes transformées en jointures
- Appliquer les prédicats le plus tôt possible
  - réduit le jeu de données manipulé
- Intègre le code des fonctions SQL simples (inline)
  - évite un appel de fonction coûteux

À partir du modèle de données et de la requête soumise, l'optimiseur de PostgreSQL va pouvoir déterminer si une jointure externe n'est pas utile à la production du résultat.

Suppression des jointures externes inutiles

Sous certaines conditions, PostgreSQL peut supprimer des jointures externes, à condition que le résultat ne soit pas modifié :

```
EXPLAIN SELECT e.matricule, e.nom, e.prenom
  FROM employes e
  LEFT JOIN services s
    ON (e.num_service = s.num_service)
 WHERE e.num_service = 4;
                        QUERY PLAN
-----------------------------------------------------------
```

```
 Seq Scan on employes e  (cost=0.00..1.18 rows=5 width=23)
   Filter: (num_service = 4)
```

Toutefois, si le prédicat de la requête est modifié pour s'appliquer sur la table `services`, la jointure est tout de même réalisée, puisqu'on réalise un test d'existence sur cette table `services` :

```
EXPLAIN SELECT e.matricule, e.nom, e.prenom
  FROM employes e
  LEFT JOIN services s
    ON (e.num_service = s.num_service)
 WHERE s.num_service = 4;
                                QUERY PLAN
--------------------------------------------------------------------------
 Nested Loop  (cost=0.15..9.39 rows=5 width=19)
   ->  Index Only Scan using services_pkey on services s  (cost=0.15..8.17...)
         Index Cond: (num_service = 4)
   ->  Seq Scan on employes e  (cost=0.00..1.18 rows=5 width=23)
         Filter: (num_service = 4)
```

Transformation des sous-requêtes

Certaines sous-requêtes sont transformées en jointure :

```
EXPLAIN SELECT *
  FROM employes emp
  JOIN (SELECT * FROM services WHERE num_service = 1) ser
    ON (emp.num_service = ser.num_service);
                              QUERY PLAN
----------------------------------------------------------------------
 Nested Loop  (cost=0.15..9.36 rows=2 width=163)
   ->  Index Scan using services_pkey on services  (cost=0.15..8.17...)
         Index Cond: (num_service = 1)
   ->  Seq Scan on employes emp  (cost=0.00..1.18 rows=2 width=43)
         Filter: (num_service = 1)
(5 lignes)
```

La sous-requête `ser` a été remonté dans l'arbre de requête pour être intégré en jointure.

Application des prédicats au plus tôt

Lorsque cela est possible, PostgreSQL essaye d'appliquer les prédicats au plus tôt :

```
EXPLAIN SELECT MAX(date_embauche)
  FROM (SELECT * FROM employes WHERE num_service = 4) e
 WHERE e.date_embauche < '2006-01-01';
                                QUERY PLAN
--------------------------------------------------------------------------
 Aggregate  (cost=1.21..1.22 rows=1 width=4)
   ->  Seq Scan on employes  (cost=0.00..1.21 rows=2 width=4)
```

```
        Filter: ((date_embauche < '2006-01-01'::date) AND (num_service = 4))
(3 lignes)
```

Les deux prédicats `num_service = 4` et `date_embauche < '2006-01-01'` ont été appliqués en même temps, réduisant ainsi le jeu de données à considéré dès le départ.

En cas de problème, il est possible d'utiliser une CTE (clause *WITH*) pour bloquer cette optimisation :

```
EXPLAIN WITH e AS (SELECT * FROM employes WHERE num_service = 4)
SELECT MAX(date_embauche)
 FROM e
WHERE e.date_embauche < '2006-01-01';
                          QUERY PLAN
---------------------------------------------------------------
 Aggregate  (cost=1.29..1.30 rows=1 width=4)
   CTE e
     ->  Seq Scan on employes  (cost=0.00..1.18 rows=5 width=43)
           Filter: (num_service = 4)
   ->  CTE Scan on e  (cost=0.00..0.11 rows=2 width=4)
         Filter: (date_embauche < '2006-01-01'::date)
```

Function inlining

```
CREATE TABLE inline (id serial, tdate date);
INSERT INTO inline (tdate)
 SELECT generate_series('1800-01-01', '2015-12-01', interval '15 days');

CREATE OR REPLACE FUNCTION add_months_sql(mydate date, nbrmonth integer)
  RETURNS date AS
$BODY$
SELECT ( mydate + interval '1 month' * nbrmonth )::date;
$BODY$
  LANGUAGE SQL;

CREATE OR REPLACE FUNCTION add_months_plpgsql(mydate date, nbrmonth integer)
  RETURNS date AS
$BODY$
 BEGIN RETURN ( mydate + interval '1 month' * nbrmonth ); END;
$BODY$
  LANGUAGE plpgsql;
```

Si l'on utilise la fonction écrite en PL/pgsql, on retrouve l'appel de la fonction dans la clause `Filter` du plan d'exécution de la requête :

```
mabase=#EXPLAIN (ANALYZE, BUFFERS) SELECT *
FROM inline WHERE tdate = add_months_plpgsql(now()::date, -1);
                                          QUERY PLAN
-----------------------------------------------------------------------------
```

```
 Seq Scan on inline  (cost=0.00..1430.52...) (actual time=42.102..42.102...)
   Filter: (tdate = add_months_plpgsql((now())::date, (-1)))
   Rows Removed by Filter: 5258
   Buffers: shared hit=24
 Total runtime: 42.139 ms
(5 lignes)
```

PostgreSQL ne sait pas intégrer le code des fonctions PL/pgsql dans ses plans d'exécution.

En revanche, en utilisant la fonction écrite en langage SQL, la définition de la fonction a été intégrée dans la clause de filtrage de la requête :

```
mabase=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM inline
WHERE tdate = add_months_sql(now()::date, -1);
                                              QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on inline  (cost=0.00..142.31...) (actual time=6.647..6.647...)
   Filter: (tdate = (((now())::date + '-1 mons'::interval))::date)
   Rows Removed by Filter: 5258
   Buffers: shared hit=24
 Total runtime: 6.699 ms
(5 lignes)
```

Le code de la fonction SQL a été correctement intégré dans le plan d'exécution. Le temps d'exécution a été divisé par 6 sur le jeu de donnée réduit, montrant l'impact de l'appel d'une fonction dans une clause de filtrage.

3.4.5 DÉCISIONS

- Stratégie d'accès aux lignes
  - Par parcours d'une table, d'un index, de TID, etc
- Stratégie d'utilisation des jointures
  - Ordre des jointures
  - Type de jointure (Nested Loop, Merge/Sort Join, Hash Join)
  - Ordre des tables jointes dans une même jointure
- Stratégie d'agrégation
  - Brut, trié, haché

Pour exécuter une requête, le planificateur va utiliser des opérations. Pour lire des lignes, il peut utiliser un parcours de table, un parcours d'index ou encore d'autres types de parcours. Ce sont généralement les premières opérations utilisées. Ensuite, d'autres opérations permettent différentes actions :

- joindre deux ensembles de lignes avec des opérations de jointures (trois au total) ;

- agréger un ensemble de lignes avec une opération d'agrégation (trois là- aussi) ;
- trier un ensemble de lignes ;
- etc.

3.5 MÉCANISME DE COÛTS

- Modèle basé sur les coûts
  - quantifier la charge pour répondre à une requête
- Chaque opération a un coût :
  - lire un bloc selon sa position sur le disque
  - manipuler une ligne issue d'une lecture de table ou d'index
  - appliquer un opérateur

L'optimiseur statistique de PostgreSQL utilise un modèle de calcul de coût. Les coûts calculés sont des indications arbitraires sur la charge nécessaire pour répondre à une requête. Chaque facteur de coût représente une unité de travail : lecture d'un bloc, manipulation des lignes en mémoire, application d'un opérateur sur des données.

3.5.1 COÛTS UNITAIRES

- L'optimiseur a besoin de connaître :
  - le coût relatif d'un accès séquentiel au disque.
  - le coût relatif d'un accès aléatoire au disque.
  - le coût relatif de la manipulation d'une ligne en mémoire.
  - le coût de traitement d'une donnée issue d'un index.
  - le coût d'application d'un opérateur.
  - le coût de la manipulation d'une ligne en mémoire pour un parcours parallèle parallélisé.
  - le coût de mise en place d'un parcours parallélisé.

Pour quantifier la charge nécessaire pour répondre à une requête, PostgreSQL utilise un mécanisme de coût. Il part du principe que chaque opération a un coût plus ou moins important.

Sept paramètres permettent d'ajuster les coûts relatifs :

- `seq_page_cost` représente le coût relatif d'un accès séquentiel au disque. Ce paramètre vaut 1 et ne devrait pas être modifié.

- `random_page_cost` représente le coût relatif d'un accès aléatoire au disque. Ce paramètre vaut 4 par défaut, cela signifie que le temps de déplacement de la tête de lecture de façon aléatoire est estimé quatre fois plus important que le temps d'accès d'un bloc à un autre.
- `cpu_tuple_cost` représente le coût relatif de la manipulation d'une ligne en mémoire. Ce paramètre vaut par défaut 0,01.
- `cpu_index_tuple_cost` répercute le coût de traitement d'une donnée issue d'un index. Ce paramètre vaut par défaut 0,005.
- `cpu_operator_cost` indique le coût d'application d'un opérateur sur une donnée. Ce paramètre vaut par défaut 0,0025.
- `parallel_tuple_cost` indique le coût de traitement d'une ligne lors d'un parcours parallélisé. Ce paramètre vaut par défaut 0.1.
- `parallel_setup_cost` indique le coût de mise en place d'un parcours parallélisé. Ce paramètre vaut par défaut 1000.0.

En général, on ne modifie pas ces paramètres sans justification sérieuse. On peut être amené à diminuer `random_page_cost` si le serveur dispose de disques rapides et d'une carte RAID équipée d'un cache important. Mais en faisant cela, il faut veiller à ne pas déstabiliser des plans optimaux qui obtiennent des temps de réponse constant. À trop diminuer `random_page_cost`, on peut obtenir de meilleurs temps de réponse si les données sont en cache, mais aussi des temps de réponse dégradés si les données ne sont pas en cache. Il n'est pas recommandé de modifier les paramètres `cpu_tuple_cost`, `cpu_index_tuple_cost` et `cpu_operator_cost` sans réelle justification.

Pour des besoins particuliers, ces paramètres sont des paramètres de sessions. Ils peuvent être modifiés dynamiquement avec l'ordre `SET` au niveau de l'application en vue d'exécuter des requêtes bien particulières.

3.6 STATISTIQUES

- Toutes les décisions du planificateur se basent sur les statistiques
  - Le choix du parcours
  - Comme le choix des jointures
- Statistiques mises à jour avec ANALYZE
- Sans bonnes statistiques, pas de bons plans

Le planificateur se base principalement sur les statistiques pour ses décisions. Le choix du parcours, le choix des jointures, le choix de l'ordre des jointures, tout cela dépend des statistiques (et un peu de la configuration). Sans statistiques à jour, le choix du planifi-

cateur a un fort risque d'être mauvais. Il est donc important que les statistiques soient mises à jour fréquemment. La mise à jour se fait avec l'instruction `ANALYZE` qui peut être exécuté manuellement ou automatiquement (via un cron ou l'autovacuum par exemple).

3.6.1 UTILISATION DES STATISTIQUES

- L'optimiseur utilise les statistiques pour déterminer :
  - la cardinalité d'un filtre -> quelle stratégie d'accès
  - la cardinalité d'une jointure -> quel algorithme de jointure
  - la cardinalité d'un regroupement -> quel algorithme de regroupement

Les statistiques sur les données permettent à l'optimiseur de requêtes de déterminer assez précisément la répartition des valeurs d'une colonne d'une table, sous la forme d'un histogramme de répartition des valeurs. Il dispose encore d'autres informations comme la répartition des valeurs les plus fréquentes, le pourcentage de `NULL`, le nombre de valeurs distinctes, etc... Toutes ces informations aideront l'optimiseur à déterminer la sélectivité d'un filtre (prédicat de la clause `WHERE`, condition de jointure) et donc quelle sera la quantité de données récupérées par la lecture d'une table en utilisant le filtre évalué.

Par exemple, pour une table simple, nommée `test`, de 1 million de lignes dont 250000 lignes ont des valeurs uniques et les autres portent la même valeur :

```
CREATE TABLE test (i integer not null, t text);
INSERT INTO test SELECT CASE WHEN i > 250000 THEN 250000 ELSE i END,
md5(i::text) FROM generate_series(1, 1000000) i;
CREATE INDEX ON test (i);
```

Après un chargement massif de données, il est nécessaire de collecter les statistiques :

```
ANALYZE test;
```

Ensuite, grâce aux statistiques connues par PostgreSQL (voir la vue `pg_stats`), l'optimiseur est capable de déterminer le chemin le plus intéressant selon les valeurs recherchées.

Ainsi, avec un filtre peu sélectif, `i = 250000`, la requête va ramener les 3/ 4 de la table. PostgreSQL choisira donc une lecture séquentielle de la table, ou `Seq Scan` :

```
base=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM test WHERE i = 250000;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on test  (cost=[...] rows=752400) (actual [...] rows=750001 loops=1)
   Filter: (i = 250000)
   Rows Removed by Filter: 249999
```

```
   Buffers: shared hit=8334
 Total runtime: 244.605 ms
(5 lignes)
```

La partie `cost` montre que l'optimiseur estime que la lecture va ramener 752400 lignes. En réalité, ce sont 750001 lignes qui sont ramenées. L'optimiseur se base donc sur une estimation obtenue selon la répartition des données.

Avec un filtre plus sélectif, la requête ne ramènera qu'une seule ligne. L'optimiseur préférera donc passer par l'index que l'on a créé :

```
base=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM test WHERE i = 250;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on test  ([...] rows=25 width=37) ([...] rows=1 loops=1)
   Recheck Cond: (i = 250)
   Buffers: shared hit=4
   ->  Bitmap Index Scan on test_i_idx  ([...] rows=25) ([...] rows=1 loops=1)
         Index Cond: (i = 250)
         Buffers: shared hit=3
 Total runtime: 0.134 ms
(7 lignes)
```

Dans ce deuxième essai, l'optimiseur estime ramener 25 lignes. En réalité, il n'en ramène qu'une seule. L'estimation reste relativement précise étant donné le volume de données.

Dans le premier cas, l'optimiseur estime qu'il est moins coûteux de passer par une lecture séquentielle de la table plutôt qu'une lecture d'index. Dans le second cas, où le filtre est très sélectif, une lecture par index est plus appropriée.

3.6.2 STATISTIQUES : TABLE ET INDEX

- Taille
- Cardinalité
- Stocké dans `pg_class`
  - `relpages` et `reltuples`

L'optimiseur a besoin de deux données statistiques pour une table ou un index : sa taille physique et le nombre de lignes portées par l'objet.

Ces deux données statistiques sont stockées dans la table `pg_class`. La taille de la table ou de l'index est exprimée en nombre de blocs de 8 Ko et stockée dans la colonne `relpages`. La cardinalité de la table ou de l'index, c'est-à-dire le nombre de lignes, est stockée dans la colonne `reltuples`.

L'optimiseur utilisera ces deux informations pour apprécier la cardinalité de la table en fonction de sa volumétrie courante en calculant sa densité estimée puis en utilisant cette densité multipliée par le nombre de blocs actuel de la table pour estimer le nombre de lignes réel de la table :

```
density = reltuples / relpages;
tuples = density * curpages;
```

3.6.3 STATISTIQUES : MONO-COLONNE

- Nombre de valeurs distinctes
- Nombre d'éléments qui n'ont pas de valeur (`NULL`)
- Largeur d'une colonne
- Distribution des données
  - tableau des valeurs les plus fréquentes
  - histogramme de répartition des valeurs

Au niveau d'une colonne, plusieurs données statistiques sont stockées :

- le nombre de valeurs distinctes,
- le nombre d'éléments qui n'ont pas de valeur (`NULL`),
- la largeur moyenne des données portées par la colonne,
- le facteur de corrélation entre l'ordre des données triées et la répartition physique des valeurs dans la table,
- la distribution des données.

La distribution des données est représentée sous deux formes qui peuvent être complémentaires. Tout d'abord, un tableau de répartition permet de connaître les valeurs les plus fréquemment rencontrées et la fréquence d'apparition de ces valeurs. Un histogramme de distribution des valeurs rencontrées permet également de connaître la répartition des valeurs pour la colonne considérée.

3.6.4 STOCKAGE DES STATISTIQUES MONO-COLONNE

- Les informations statistiques vont dans la table `pg_statistic`
  - mais elle est difficile à comprendre
  - mieux vaut utiliser la vue `pg_stats`
  - une table vide n'a pas de statistiques
- Taille et cardinalité dans `pg_class`

  - colonnes `relpages` et `reltuples`

Le stockage des statistiques se fait dans le catalogue système `pg_statistic` mais les colonnes de cette table sont difficiles à interpréter. Il est préférable de passer par la vue `pg_stats` qui est plus facilement compréhensible par un être humain.

La collecte des statistiques va également mettre à jour la table `pg_class` avec deux informations importantes pour l'optimiseur. Il s'agit de la taille d'une table, exprimée en nombre de blocs de 8 Ko et stockée dans la colonne `relpages`. La cardinalité de la table, c'est-à-dire le nombre de lignes de la table, est stockée dans la colonne `reltuples`. L'optimiseur utilisera ces deux informations pour apprécier la cardinalité de la table en fonction de sa volumétrie courante.

3.6.5 VUE PG_STATS

- Une ligne par colonne de chaque table
- 3 colonnes d'identification
  - schemaname, tablename, attname
- 8 colonnes d'informations statistiques
  - inherited, null_frac, avg_width, n_distinct
  - most_common_vals, most_common_freqs, histogram_bounds
  - most_common_elems, most_common_elem_freqs, elem_count_histogram
  - correlation

La vue `pg_stats` a été créée pour faciliter la compréhension des statistiques récupérées par la commande `ANALYZE`.

Elle est composée de trois colonnes qui permettent d'identifier la colonne :

- schemaname : nom du schéma (jointure possible avec pg_namespace)
- tablename : nom de la table (jointure possible avec pg_class, intéressant pour récupérer reltuples et relpages)
- attname : nom de la colonne (jointure possible avec pg_attribute, intéressant pour récupérer attstatstarget, valeur d'échantillon)

Suivent ensuite les colonnes de statistiques.

inherited

Si true, les statistiques incluent les valeurs de cette colonne dans les tables filles.

Exemple

```
b1=# SELECT count(*) FROM ONLY parent;
-[ RECORD 1 ]
count | 0
b1=# SELECT * FROM pg_stats WHERE tablename='parent';
-[ RECORD 1 ]-----+-----------------------------------------------------
schemaname        | public
tablename         | parent
attname           | id
inherited         | t
null_frac         | 0
avg_width         | 4
n_distinct        | -0.285714
most_common_vals  | {1,2,17,18,19,20,3,4,5,15,16,6,7,8,9,10}
[...]
histogram_bounds  | {11,12,13,14}
correlation       | 0.762715
```

null_frac

Cette statistique correspond au pourcentage de valeurs NULL dans l'échantillon considéré. Elle est toujours calculée.

avg_width

Il s'agit de la largeur moyenne en octets des éléments de cette colonne. Elle est constante pour les colonnes dont le type est à taille fixe (integer, booléen, char, etc.). Dans le cas du type `char(n)`, il s'agit du nombre de caractères saisissables + 1. Il est variable pour les autres (principalement text, varchar, bytea).

n_distinct

Si cette colonne contient un nombre positif, il s'agit du nombre de valeurs distinctes dans l'échantillon. Cela arrive uniquement quand le nombre de valeurs distinctes possibles semble fixe.

Si cette colonne contient un nombre négatif, il s'agit du nombre de valeurs distinctes dans l'échantillon divisé par le nombre de lignes. Cela survient uniquement quand le nombre de valeurs distinctes possibles semble variable. -1 indique donc que toutes les valeurs sont distinctes, -0,5 que chaque valeur apparaît deux fois.

Cette colonne peut être `NULL` si le type de données n'a pas d'opérateur =.

Il est possible de forcer cette colonne a une valeur constante en utilisant l'ordre `ALTER TABLE nom_table ALTER COLUMN nom_colonne SET (parametre = valeur);` où parametre vaut soit `n_distinct` (pour une table standard) soit `n_distinct_inherited` (pour une table comprenant des partitions). Pour les grosses tables contenant des valeurs distinctes, indiquer une grosse valeur ou la valeur -1 permet de favoriser l'utilisation de

parcours d'index à la place de parcours de bitmap. C'est aussi utile pour des tables où les données ne sont pas réparties de façon homogène, et où la collecte de cette statistique est alors faussée.

most_common_vals

Cette colonne contient une liste triée des valeurs les plus communes. Elle peut être `NULL` si les valeurs semblent toujours aussi communes ou si le type de données n'a pas d'opérateur =.

most_common_freqs

Cette colonne contient une liste triée des fréquences pour les valeurs les plus communes. Cette fréquence est en fait le nombre d'occurrences de la valeur divisé par le nombre de lignes. Elle est `NULL` si `most_common_vals` est `NULL`.

histogram_bounds

PostgreSQL prend l'échantillon récupéré par `ANALYZE`. Il trie ces valeurs. Ces données triées sont partagées en x tranches, appelées classes, égales, où x dépend de la valeur du paramètre `default_statistics_target` ou de la configuration spécifique de la colonne. Il construit ensuite un tableau dont chaque valeur correspond à la valeur de début d'une tranche.

most_common_elems, most_common_elem_freqs, elem_count_histogram

Ces trois colonnes sont équivalentes aux trois précédentes, mais uniquement pour les données de type tableau.

correlation

Cette colonne est la corrélation statistique entre l'ordre physique et l'ordre logique des valeurs de la colonne. Si sa valeur est proche de -1 ou 1, un parcours d'index est privilégié. Si elle est proche de 0, un parcours séquentiel est mieux considéré.

Cette colonne peut être `NULL` si le type de données n'a pas d'opérateur `<`.

3.6.6 STATISTIQUES : MULTI-COLONNES

- Pas par défaut
- `CREATE STATISTICS`
- Deux types de statistique
  - nombre de valeurs distinctes
  - dépendances fonctionnelles

- À partir de la version 10

Par défaut, la commande ANALYZE de PostgreSQL calcule des statistiques mono-colonnes uniquement. Depuis la version 10, elle peut aussi calculer certaines statistiques multi-colonnes.

Pour cela, il est nécessaire de créer un objet statistique avec l'ordre SQL CREATE STATISTICS. Cet objet indique les colonnes concernées ainsi que le type de statistique souhaité.

Actuellement, PostgreSQL supporte deux types de statistiques pour ces objets :

- ndistinct pour le nombre de valeurs distinctes sur ces colonnes ;
- dependencies pour les dépendances fonctionnelles.

Dans les deux cas, cela peut permettre d'améliorer fortement les estimations de nombre de lignes, ce qui ne peut qu'amener de meilleurs plans d'exécution.

3.6.7 CATALOGUE PG_STATISTIC_EXT

- Une ligne par objet statistique
- 4 colonnes d'identification
  - stxrelid, stxname, stxnamespace, stxkeys
- 1 colonne pour connaître le type de statistiques géré
  - stxkind
- 2 colonnes d'informations statistiques
  - stxndistinct
  - stxdependencies

stxname est le nom de l'objet statistique, et stxnamespace l'OID de son schéma.

stxrelid précise l'OID de la table concernée par cette statistique. stxkeys est un tableau d'entiers correspondant aux numéros des colonnes.

stxkind peut avoir une ou plusieurs valeurs parmi d pour le nombre de valeurs distinctes et f pour les dépendances fonctionnelles.

Créons une table avec deux colonnes et peuplons-la avec les mêmes données :

```
postgres=# CREATE TABLE t (a INT, b INT);
CREATE TABLE
postgres=# INSERT INTO t SELECT i % 100, i % 100 FROM generate_series(1, 10000) s(i);
INSERT 0 10000
postgres=# ANALYZE t;
```

`ANALYZE`

Après une analyse des données de la table, les statistiques sont à jour comme le montrent ces deux requêtes :

```
postgres=# EXPLAIN (ANALYZE) SELECT * FROM t WHERE a = 1;
                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on t  (cost=0.00..170.00 rows=100 width=8)
         (actual time=0.037..1.704 rows=100 loops=1)
   Filter: (a = 1)
   Rows Removed by Filter: 9900
 Planning time: 0.097 ms
 Execution time: 1.731 ms
(5 rows)

postgres=# EXPLAIN (ANALYZE) SELECT * FROM t WHERE b = 1;
                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on t  (cost=0.00..170.00 rows=100 width=8)
         (actual time=0.010..1.590 rows=100 loops=1)
   Filter: (b = 1)
   Rows Removed by Filter: 9900
 Planning time: 0.029 ms
 Execution time: 1.609 ms
(5 rows)
```

Cela fonctionne bien (*i.e.* l'estimation du nombre de lignes est très proche de la réalité) dans le cas spécifique où le filtre se fait sur une seule colonne. Par contre, si le filtre se fait sur les deux colonnes, l'estimation diffère d'un facteur d'échelle :

```
postgres=# EXPLAIN (ANALYZE) SELECT * FROM t WHERE a = 1 AND b = 1;
                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8)
       (actual time=0.009..1.554 rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
 Planning time: 0.044 ms
 Execution time: 1.573 ms
(5 rows)
```

En fait, il y a une dépendance fonctionnelle entre ces deux colonnes mais PostgreSQL ne le sait pas car ses statistiques sont mono-colonnes par défaut. Pour avoir des statistiques sur les deux colonnes, il faut créer un objet statistique pour ces deux colonnes :

```
postgres=# CREATE STATISTICS stts (dependencies) ON a, b FROM t;
CREATE STATISTICS
postgres=# ANALYZE t;
ANALYZE
postgres=# EXPLAIN (ANALYZE) SELECT * FROM t WHERE a = 1 AND b = 1;
                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=100 width=8)
         (actual time=0.007..0.668 rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
 Planning time: 0.093 ms
 Execution time: 0.683 ms
(5 rows)
```

Cette fois, l'estimation est beaucoup plus proche de la réalité.

Maintenant, prenons le cas d'un regroupement :

```
postgres=# EXPLAIN (ANALYZE) SELECT COUNT(*) FROM t GROUP BY a;
                           QUERY PLAN
-----------------------------------------------------------------
 HashAggregate  (cost=195.00..196.00 rows=100 width=12)
            (actual time=2.346..2.358 rows=100 loops=1)
   Group Key: a
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=4)
               (actual time=0.006..0.640 rows=10000 loops=1)
 Planning time: 0.024 ms
 Execution time: 2.381 ms
(5 rows)
```

L'estimation du nombre de lignes pour un regroupement sur une colonne est très bonne. Par contre, sur deux colonnes :

```
postgres=# EXPLAIN (ANALYZE) SELECT COUNT(*) FROM t GROUP BY a, b;
                           QUERY PLAN
-----------------------------------------------------------------
 HashAggregate  (cost=220.00..230.00 rows=1000 width=16)
             (actual time=2.321..2.339 rows=100 loops=1)
```

```
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8)
               (actual time=0.004..0.596 rows=10000 loops=1)
 Planning time: 0.025 ms
 Execution time: 2.359 ms
(5 rows)
```

Là-aussi, on constate un facteur d'échelle important entre l'estimation et la réalité. Et là-aussi, c'est un cas où un objet statistique peut fortement aider :

```
postgres=# DROP STATISTICS stts;
DROP STATISTICS
postgres=# CREATE STATISTICS stts (dependencies, ndistinct) ON a, b FROM t;
CREATE STATISTICS
postgres=# ANALYZE t;
ANALYZE
postgres=# EXPLAIN (ANALYZE) SELECT COUNT(*) FROM t GROUP BY a, b;
                      QUERY PLAN
-------------------------------------------------------------
 HashAggregate  (cost=220.00..221.00 rows=100 width=16)
             (actual time=3.310..3.324 rows=100 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8)
               (actual time=0.007..0.807 rows=10000 loops=1)
 Planning time: 0.087 ms
 Execution time: 3.356 ms
(5 rows)
```

L'estimation est bien meilleure grâce aux statistiques spécifiques aux deux colonnes.

3.6.8 ANALYZE

- Ordre SQL de calcul de statistiques
  - ANALYZE [VERBOSE] [table [(colonne [, ...])]]
- Sans argument : base entière
- Avec argument : la table complète ou certaines colonnes seulement
- Prend un échantillon de chaque table
- Et calcule des statistiques sur cet échantillon
- Si table vide, conservation des anciennes statistiques

ANALYZE est l'ordre SQL permettant de mettre à jour les statistiques sur les données. Sans argument, l'analyse se fait sur la base complète. Si un argument est donné, il doit correspondre au nom de la table à analyser. Il est même possible d'indiquer les colonnes à traiter.

En fait, cette instruction va exécuter un calcul d'un certain nombre de statistiques. Elle ne va pas lire la table entière, mais seulement un échantillon. Sur cet échantillon, chaque colonne sera traitée pour récupérer quelques informations comme le pourcentage de valeurs NULL, les valeurs les plus fréquentes et leur fréquence, sans parler d'un histogramme des valeurs. Toutes ces informations sont stockées dans un catalogue système nommé pg_statistics.

Dans le cas d'une table vide, les anciennes statistiques sont conservées. S'il s'agit d'une nouvelle table, les statistiques sont initialement vides. La table n'est jamais considérée vide par l'optimiseur, qui utilise alors des valeurs par défaut.

3.6.9 FRÉQUENCE D'ANALYSE

- Dépend principalement de la fréquence des requêtes DML
- Cron
  - Avec psql
  - Avec vacuumdb (option --analyze-only en 9.0)
- Autovacuum fait du ANALYZE
  - Pas sur les tables temporaires
  - Pas assez rapidement dans certains cas

Les statistiques doivent être mises à jour fréquemment. La fréquence exacte dépend surtout de la fréquence des requêtes d'insertion/modification/ suppression des lignes des tables. Néanmoins, un ANALYZE tous les jours semble un minimum, sauf cas spécifique.

L'exécution périodique peut se faire avec cron (ou les tâches planifiées sous Windows). Il n'existe pas d'outils PostgreSQL pour lancer un seul ANALYZE . L'outil vaccumdb se voit doté d'une option --analyze-only pour combler ce manque. Avant, il était nécessaire de passer par psql et son option -c qui permet de préciser la requête à exécuter. En voici un exemple :

```
psql -c "ANALYZE" b1
```

Cet exemple exécute la commande ANALYZE sur la base b1 locale.

Le démon autovacuum fait aussi des ANALYZE. La fréquence dépend de sa configuration. Cependant, il faut connaître deux particularités de cet outil :

- Ce démon a sa propre connexion à la base. Il ne peut donc pas voir les tables temporaires appartenant aux autres sessions. Il ne sera donc pas capable de mettre à jour leurs statistiques.
- Après une insertion ou une mise à jour massive, autovacuum ne va pas forcément lancer un `ANALYZE` immédiat. En effet, `autovacuum` ne cherche les tables à traiter que toutes les minutes (par défaut). Si, après la mise à jour massive, une requête est immédiatement exécutée, il y a de fortes chances qu'elle s'exécute avec des statistiques obsolètes. Il est préférable dans ce cas de lancer un `ANALYZE` manuel sur la ou les tables ayant subi l'insertion ou la mise à jour massive.

3.6.10 ÉCHANTILLON STATISTIQUE

- Se configure dans postgresql.conf
  - `default_statistics_target = 100`
- Configurable par colonne

```
ALTER TABLE nom ALTER [ COLUMN ] colonne SET STATISTICS valeur;
```

- Par défaut, récupère 30000 lignes au hasard
  - `300 * default_statistics_target`
- Va conserver les 100 valeurs les plus fréquentes avec leur fréquence

Par défaut, un `ANALYZE` récupère 30000 lignes d'une table. Les statistiques générées à partir de cet échantillon sont bonnes si la table ne contient pas des millions de lignes. Si c'est le cas, il faudra augmenter la taille de l'échantillon. Pour cela, il faut augmenter la valeur du paramètre `default_statistics_target`. Ce dernier vaut 100 par défaut. La taille de l'échantillon est de `300 x default_statistics_target`. Augmenter ce paramètre va avoir plusieurs répercussions. Les statistiques seront plus précises grâce à un échantillon plus important. Mais du coup, les statistiques seront plus longues à calculer, prendront plus de place sur le disque, et demanderont plus de travail au planificateur pour générer le plan optimal. Augmenter cette valeur n'a donc pas que des avantages.

Du coup, les développeurs de PostgreSQL ont fait en sorte qu'il soit possible de le configurer colonne par colonne avec l'instruction suivante :

```
ALTER TABLE nom_table ALTER [ COLUMN ] nom_colonne SET STATISTICS valeur;
```

3.7 QU'EST-CE QU'UN PLAN D'EXÉCUTION ?

- Plan d'exécution
  - représente les différentes opérations pour répondre à la requête
  - sous forme arborescente
  - composé des nœuds d'exécution
  - plusieurs opérations simples mises bout à bout

3.7.1 NŒUD D'EXÉCUTION

- Nœud
  - opération simple : lectures, jointures, tris, etc.
  - unité de traitement
  - produit et consomme des données
- Enchaînement des opérations
  - chaque nœud produit les données consommées par le nœud parent
  - nœud final retourne les données à l'utilisateur

Les nœuds correspondent à des unités de traitement qui réalisent des opérations simples sur un ou deux ensembles de données : lecture d'une table, jointures entre deux tables, tri d'un ensemble, etc. Si le plan d'exécution était une recette, chaque nœud serait une étape de la recette.

Les nœuds peuvent produire et consommer des données.

3.7.2 LECTURE D'UN PLAN

```
                              QUERY PLAN
--------------------------------------------------------------------
 Hash Join  (cost=1.06..2.29 rows=4 width=48)
   Hash Cond: (emp.num_service = ser.num_service)
   ->  Seq Scan on employes emp  (cost=0.00..1.14 rows=14 width=35)
   ->  Hash  (cost=1.05..1.05 rows=1 width=21)
         ->  Seq Scan on services ser  (cost=0.00..1.05 rows=1 width=21)
               Filter: ((localisation)::text = 'Nantes'::text)
```

Un plan d'exécution est lu en partant du nœud se trouvant le plus à droite et en remontant jusqu'au nœud final. Quand le plan contient plusieurs nœuds, le premier nœud exécuté est celui qui se trouve le plus à droite. Celui qui est le plus à gauche (la première ligne) est le dernier nœud exécuté. Tous les nœuds sont exécutés simultanément, et traitent les données dès qu'elles sont transmises par le nœud parent (le ou les nœuds juste en dessous, à droite).

Chaque nœud montre les coûts estimés dans le premier groupe de parenthèses :

- `cost` est un couple de deux coûts
- la première valeur correspond au coût pour récupérer la première ligne (souvent nul dans le cas d'un parcours séquentiel) ;
- la deuxième valeur correspond au coût pour récupérer toutes les lignes (cette valeur dépend essentiellement de la taille de la table lue, mais aussi de l'opération de filtre ici présente) ;
- `rows` correspond au nombre de lignes que le planificateur pense récupérer à la sortie de ce nœud ;
- `width` est la largeur en octets de la ligne.

Cet exemple simple permet de voir le travail de l'optimiseur :

```
=> EXPLAIN SELECT matricule, nom, prenom, nom_service, fonction, localisation
  FROM employes emp
  JOIN services ser ON (emp.num_service = ser.num_service)
 WHERE ser.localisation = 'Nantes';

                              QUERY PLAN
-----------------------------------------------------------------------

 Hash Join  (cost=1.06..2.29 rows=4 width=48)
   Hash Cond: (emp.num_service = ser.num_service)
   ->  Seq Scan on employes emp  (cost=0.00..1.14 rows=14 width=35)
   ->  Hash  (cost=1.05..1.05 rows=1 width=21)
         ->  Seq Scan on services ser  (cost=0.00..1.05 rows=1 width=21)
               Filter: ((localisation)::text = 'Nantes'::text)
```

Ce plan débute par la lecture de la table `services`. L'optimiseur estime que cette lecture ramènera une seule ligne (`rows=1`), que cette ligne occupera 21 octets en mémoire (`width=21`). Il s'agit de la sélectivité du filtre `WHERE localisation = 'Nantes'`. Le coût de départ de cette lecture est de 0 (`cost=0.00`). Le coût total de cette lecture est de `1.05`, qui correspond à la lecture séquentielle d'un seul bloc (donc `seq_page_cost`) et à la manipulation des 4 lignes de la tables `services` (donc 4 * `cpu_tuple_cost` + 4 * `cpu_operator_cost`). Le résultat de cette lecture est ensuite haché par le nœud `Hash`, qui précède la jointure de type `Hash Join`.

La jointure peut maintenant commencer, avec le nœud `Hash Join`. Il est particulier, car il prend 2 entrées : la donnée hachée initialement, et les données issues de la lecture d'une seconde table (peu importe le type d'accès). Le nœud a un coût de démarrage de `1.06`, soit le coût du hachage additionné au coût de manipulation du tuple de départ. Il s'agit du coût de production du premier tuple de résultat. Le coût total de production du résultat est de `2.29`. La jointure par hachage démarre réellement lorsque la lecture de la table `employes` commence. Cette lecture remontera 14 lignes, sans application de filtre. La totalité de la table est donc remontée et elle est très petite donc tient sur un seul bloc de 8 Ko. Le coût d'accès total est donc facilement déduit à partir de cette information. À partir des sélectivités précédentes, l'optimiseur estime que la jointure ramènera 4 lignes au total.

3.7.3 OPTIONS DE L'EXPLAIN

- Des options supplémentaires
  - ANALYZE
  - BUFFERS
  - COSTS
  - TIMING
  - VERBOSE
  - SUMMARY
  - FORMAT
- Donnant des informations supplémentaires très utiles

Au fil des versions, `EXPLAIN` a gagné en options. L'une d'entre elles permet de sélectionner le format en sortie. Toutes les autres permettent d'obtenir des informations supplémentaires.

Option ANALYZE

Le but de cette option est d'obtenir les informations sur l'exécution réelle de la requête.

Avec cette option, la requête est réellement exécutée. Attention aux INSERT/ UPDATE/DELETE. Pensez à les englober dans une transaction que vous annulerez après coup.

Voici un exemple utilisant cette option :

```
b1=# EXPLAIN ANALYZE SELECT * FROM t1 WHERE c1 <1000;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..17.50 rows=1000 width=8)
```

```
                (actual time=0.015..0.504 rows=999 loops=1)
   Filter: (c1 < 1000)
 Total runtime: 0.766 ms
(3 rows)
```

Quatre nouvelles informations apparaissent, toutes liées à l'exécution réelle de la requête :

- `actual time`
- la première valeur correspond à la durée en milliseconde pour récupérer la première ligne ;
- la deuxième valeur est la durée en milliseconde pour récupérer toutes les lignes ;
- `rows` est le nombre de lignes réellement récupérées ;
- `loops` est le nombre d'exécution de ce nœud.

Multiplier la durée par le nombre de boucles pour obtenir la durée réelle d'exécution du nœud.

L'intérêt de cette option est donc de trouver l'opération qui prend du temps dans l'exécution de la requête, mais aussi de voir les différences entre les estimations et la réalité (notamment au niveau du nombre de lignes).

Option BUFFERS

Cette option apparaît en version 9.1. Elle n'est utilisable qu'avec l'option `ANALYZE`. Elle est désactivée par défaut.

Elle indique le nombre de blocs impactés par chaque nœud du plan d'exécution, en lecture comme en écriture.

Voici un exemple de son utilisation :

```
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c1 <1000;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..17.50 rows=1000 width=8)
                 (actual time=0.015..0.493 rows=999 loops=1)
   Filter: (c1 < 1000)
   Buffers: shared hit=5
 Total runtime: 0.821 ms
(4 rows)
```

La nouvelle ligne est la ligne `Buffers`. Elle peut contenir un grand nombre d'informations :

Informations	Type d'objet concerné	Explications
Shared hit	Table ou index standard	Lecture d'un bloc dans le cache

Informations	Type d'objet concerné	Explications
Shared read	Table ou index standard	Lecture d'un bloc hors du cache
Shared written	Table ou index standard	Écriture d'un bloc
Local hit	Table ou index temporaire	Lecture d'un bloc dans le cache
Local read	Table ou index temporaire	Lecture d'un bloc hors du cache
Local written	Table ou index temporaire	Écriture d'un bloc
Temp read	Tris et hachages	Lecture d'un bloc
Temp written	Tris et hachages	Écriture d'un bloc

Option COSTS

L'option `COSTS` apparaît avec la version 9.0. Une fois activée, elle indique les estimations du planificateur.

```
b1=# EXPLAIN (COSTS OFF) SELECT * FROM t1 WHERE c1 <1000;
           QUERY PLAN
--------------------------------
 Seq Scan on t1
   Filter: (c1 < 1000)
(2 rows)

b1=# EXPLAIN (COSTS ON) SELECT * FROM t1 WHERE c1 <1000;
                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..17.50 rows=1000 width=8)
   Filter: (c1 < 1000)
(2 rows)
```

Option TIMING

Cette option n'est disponible que depuis la version 9.2. Elle n'est utilisable qu'avec l'option `ANALYZE`.

Elle ajoute les informations sur les durées en milliseconde. Elle est activée par défaut. Sa désactivation peut être utile sur certains systèmes où le chronométrage prend beaucoup de temps et allonge inutilement la durée d'exécution de la requête.

Voici un exemple de son utilisation :

```
b1=# EXPLAIN (ANALYZE,TIMING ON) SELECT * FROM t1 WHERE c1 <1000;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..17.50 rows=1000 width=8)
                 (actual time=0.017..0.520 rows=999 loops=1)
   Filter: (c1 < 1000)
```

```
   Rows Removed by Filter: 1
 Total runtime: 0.783 ms
(4 rows)

b1=# EXPLAIN (ANALYZE,TIMING OFF) SELECT * FROM t1 WHERE c1 <1000;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..17.50 rows=1000 width=8) (actual rows=999 loops=1)
   Filter: (c1 < 1000)
   Rows Removed by Filter: 1
 Total runtime: 0.418 ms
(4 rows)
```

Option VERBOSE

L'option `VERBOSE` permet d'afficher des informations supplémentaires comme la liste des colonnes en sortie, le nom de la table qualifié du schéma, le nom de la fonction qualifié du schéma, le nom du trigger, etc. Elle est désactivée par défaut.

```
b1=# EXPLAIN (VERBOSE) SELECT * FROM t1 WHERE c1 <1000;
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on public.t1  (cost=0.00..17.50 rows=1000 width=8)
   Output: c1, c2
   Filter: (t1.c1 < 1000)
(3 rows)
```

On voit dans cet exemple que le nom du schéma est ajouté au nom de la table. La nouvelle section `Output` indique la liste des colonnes de l'ensemble de données en sortie du nœud.

Option SUMMARY

Cette option apparaît en version 10. Elle permet d'afficher ou non le résumé final indiquant la durée de la planification et de l'exécution. Un `EXPLAIN` simple n'affiche pas le résumé par défaut. Par contre, un `EXPLAIN ANALYZE` l'affiche par défaut.

```
b1=# EXPLAIN SELECT * FROM t1;
                      QUERY PLAN
-------------------------------------------------------
 Seq Scan on t1  (cost=0.00..35.50 rows=2550 width=4)
(1 row)

b1=# EXPLAIN (SUMMARY on) SELECT * FROM t1;
                      QUERY PLAN
-------------------------------------------------------
 Seq Scan on t1  (cost=0.00..35.50 rows=2550 width=4)
```

```
 Planning time: 0.080 ms
(2 rows)

b1=# EXPLAIN (ANALYZE) SELECT * FROM t1;
                                              QUERY PLAN
------------------------------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..35.50 rows=2550 width=4)
            (actual time=0.004..0.004 rows=0 loops=1)
 Planning time: 0.069 ms
 Execution time: 0.037 ms
(3 rows)

b1=# EXPLAIN (ANALYZE, SUMMARY off) SELECT * FROM t1;
                                              QUERY PLAN
------------------------------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..35.50 rows=2550 width=4)
            (actual time=0.002..0.002 rows=0 loops=1)
(1 row)
```

Option FORMAT

L'option `FORMAT` apparaît en version 9.0. Elle permet de préciser le format du texte en sortie. Par défaut, il s'agit du texte habituel, mais il est possible de choisir un format balisé parmi XML, JSON et YAML. Voici ce que donne la commande `EXPLAIN` avec le format XML :

```
b1=# EXPLAIN (FORMAT XML) SELECT * FROM t1 WHERE c1 <1000;
                       QUERY PLAN
-------------------------------------------------------
 <explain xmlns="http://www.postgresql.org/2009/explain">+
   <Query>                                              +
     <Plan>                                             +
       <Node-Type>Seq Scan</Node-Type>                  +
       <Relation-Name>t1</Relation-Name>                +
       <Alias>t1</Alias>                                +
       <Startup-Cost>0.00</Startup-Cost>                +
       <Total-Cost>17.50</Total-Cost>                   +
       <Plan-Rows>1000</Plan-Rows>                      +
       <Plan-Width>8</Plan-Width>                       +
       <Filter>(c1 &lt; 1000)</Filter>                  +
     </Plan>                                            +
   </Query>                                             +
 </explain>
(1 row)
```

3.7.4 DÉTECTER LES PROBLÈMES

- Différence importante entre l'estimation du nombre de lignes et la réalité
- Boucles
  - appels très nombreux dans une boucle (nested loop)
  - opération lente sur lesquels PostgreSQL boucle
- Temps d'exécution conséquent sur une opération
- Opérations utilisant beaucoup de blocs (option BUFFERS)

Lorsqu'une requête s'exécute lentement, cela peut être un problème dans le plan. La sortie de `EXPLAIN` peut apporter quelques informations qu'il faut savoir décoder. Une différence importante entre le nombre de lignes estimé et le nombre de lignes réel laisse un doute sur les statistiques présentes. Soit elles n'ont pas été réactualisées récemment, soit l'échantillon n'est pas suffisamment important pour que les statistiques donnent une vue proche du réel du contenu de la table.

L'option `BUFFERS` d'`EXPLAIN` permet également de mettre en valeur les opérations d'entrées/sorties lourdes. Cette option affiche notamment le nombre de blocs lus en/hors cache de PostgreSQL, sachant qu'un bloc fait généralement 8 Ko, il est aisé de déterminer le volume de données manipulé par une requête.

3.7.5 STATISTIQUES ET COÛTS

- Détermine à partir des statistiques
  - cardinalité des prédicats
  - cardinalité des jointures
- Coût d'accès déterminé selon
  - des cardinalités
  - volumétrie des tables

Afin de comparer les différents plans d'exécution possibles pour une requête et choisir le meilleur, l'optimiseur a besoin d'estimer un coût pour chaque nœud du plan.

L'estimation la plus cruciale est celle liée aux nœuds de parcours de données, car c'est d'eux que découlera la suite du plan. Pour estimer le coût de ces nœuds, l'optimiseur s'appuie sur les informations statistiques collectées, ainsi que sur la valeur de paramètres de configuration.

Les deux notions principales de ce calcul sont la cardinalité (nombre de lignes estimées en sortie d'un nœud) et la sélectivité (fraction des lignes conservées après l'application d'un filtre).

Voici ci-dessous un exemple de calcul de cardinalité et de détermination du coût associé.

Calcul de cardinalité

Pour chaque prédicat et chaque jointure, PostgreSQL va calculer sa sélectivité et sa cardinalité. Pour un prédicat, cela permet de déterminer le nombre de lignes retournées par le prédicat par rapport au nombre total de lignes de la table. Pour une jointure, cela permet de déterminer le nombre de lignes retournées par la jointure entre deux tables.

L'optimiseur dispose de plusieurs façons de calculer la cardinalité d'un filtre ou d'une jointure selon que la valeur recherchée est une valeur unique, que la valeur se trouve dans le tableau des valeurs les plus fréquentes ou dans l'histogramme. L'exemple ci-dessous montre comment calculer la cardinalité d'un filtre simple sur une table `pays` de 25 lignes. La valeur recherchée se trouve dans le tableau des valeurs les plus fréquentes, la cardinalité peut être calculée directement. Si ce n'était pas le cas, il aurait fallu passer par l'histogramme des valeurs pour calculer d'abord la sélectivité du filtre pour en déduire ensuite la cardinalité.

Dans l'exemple qui suit, une table `pays` contient 25 entrées

La requête suivante permet de récupérer la fréquence d'apparition de la valeur recherchée dans le prédicat `WHERE region_id = 1` :

```
SELECT tablename, attname, value, freq
  FROM (SELECT tablename, attname, mcv.value, mcv.freq FROM pg_stats,
              LATERAL ROWS FROM (unnest(most_common_vals::text::int[]),
                                 unnest(most_common_freqs)) AS mcv(value, freq)
         WHERE tablename = 'pays'
           AND attname = 'region_id') get_mcv
 WHERE value = 1;
 tablename |  attname  | value | freq
-----------+-----------+-------+------
 pays      | region_id |     1 |  0.2
(1 row)
```

L'optimiseur calcule la cardinalité du prédicat `WHERE region_id = 1` en multipliant cette fréquence de la valeur recherchée avec le nombre total de lignes de la table :

```
SELECT 0.2 * reltuples AS cardinalite_predicat
  FROM pg_class
 WHERE relname = 'pays';
 cardinalite_predicat
----------------------
```

```
                    5
(1 row)
```

On peut vérifier que le calcul est bon en obtenant le plan d'exécution de la requête impliquant la lecture de `pays` sur laquelle on applique le prédicat évoqué plus haut :

```
EXPLAIN SELECT * FROM pays WHERE region_id = 1;
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on pays  (cost=0.00..1.31 rows=5 width=49)
   Filter: (region_id = 1)
(2 rows)
```

Calcul de coût

Une table `pays` peuplée de 25 lignes va permettre de montrer le calcul des coûts réalisés par l'optimiseur. L'exemple présenté ci-dessous est simplifié. En réalité, les calculs sont plus complexes car ils tiennent également compte de la volumétrie réelle de la table.

Le coût de la lecture séquentielle de la table `pays` est calculé à partir de deux composantes. Toute d'abord, le nombre de pages (ou blocs) de la table permet de déduire le nombre de blocs à accéder pour lire la table intégralement. Le paramètre `seq_page_cost` sera appliqué ensuite pour indiquer le coût de l'opération :

```
SELECT relname, relpages * current_setting('seq_page_cost')::float AS cout_acces
  FROM pg_class
 WHERE relname = 'pays';
 relname | cout_acces
---------+------------
 pays    |          1
```

Cependant, le coût d'accès seul ne représente pas le coût de la lecture des données. Une fois que le bloc est monté en mémoire, PostgreSQL doit décoder chaque ligne individuellement. L'optimiseur utilise `cpu_tuple_cost` pour estimer le coût de manipulation des lignes :

```
SELECT relname,
       relpages * current_setting('seq_page_cost')::float
       + reltuples * current_setting('cpu_tuple_cost')::float AS cout
  FROM pg_class
 WHERE relname = 'pays';
 relname | cout
---------+------
 pays    | 1.25
```

On peut vérifier que le calcul est bon :

```
EXPLAIN SELECT * FROM pays;
                       QUERY PLAN
```

```
-------------------------------------------------------
 Seq Scan on pays  (cost=0.00..1.25 rows=25 width=53)
(1 ligne)
```

Si l'on applique un filtre à la requête, les traitements seront plus lourds. Par exemple, en ajoutant le prédicat `WHERE pays = 'FR'`.

Il faut non seulement extraire les lignes les unes après les autres, mais il faut également appliquer l'opérateur de comparaison utilisé. L'optimiseur utilise le paramètre `cpu_operator_cost` pour déterminer le coût d'application d'un filtre :

```
SELECT relname,
       relpages * current_setting('seq_page_cost')::float
       + reltuples * current_setting('cpu_tuple_cost')::float
       + reltuples * current_setting('cpu_operator_cost')::float AS cost
  FROM pg_class
 WHERE relname = 'pays';
 relname |  cost
---------+--------
 pays    | 1.3125
```

En récupérant le plan d'exécution de la requête à laquelle est appliqué le filtre `WHERE pays = 'FR'`, on s'aperçoit que le calcul est juste, à l'arrondi près :

```
EXPLAIN SELECT * FROM pays WHERE code_pays = 'FR';
                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on pays  (cost=0.00..1.31 rows=1 width=53)
   Filter: (code_pays = 'FR'::text)
(2 lignes)
```

Pour aller plus loin dans le calcul de sélectivité, de cardinalité et de coût, la documentation de PostgreSQL montre un exemple complet de calcul de sélectivité et indique les références des fichiers sources dans lesquels fouiller pour en savoir plus :Comment le planificateur utilise les statistiques[18] .

3.8 NŒUDS D'EXÉCUTION LES PLUS COURANTS

- Un plan est composé de nœuds
  - certains produisent des données
  - d'autres consomment des données et les retournent
  - le nœud final retourne les données à l'utilisateur

[18] http://docs.postgresql.fr/current/planner-stats-details.html

- chaque nœud consomme au fur et à mesure les données produites par les nœuds parents

---

3.8.1 NOEUDS DE TYPE PARCOURS

- Seq Scan
- Parallel Seq Scan
- Function Scan
- et des parcours d'index

Les parcours sont les seules opérations qui lisent les données des tables (normales, temporaires ou non journalisées). Elles ne prennent donc rien en entrée et fournissent un ensemble de données en sortie. Cet ensemble peut être trié ou non, filtré ou non.

Il existe plusieurs types de parcours possibles. Parmi les plus fréquents, on retrouve :

- le parcours de table ;
- le parcours de fonction ;
- les parcours d'index.

Depuis la version 9.6, les parcours de table sont parallélisables.

Les parcours d'index sont documentés par la suite.

L'opération `Seq Scan` correspond à une lecture séquentielle d'une table, aussi appelée `Full Table Scan` sur d'autres SGBD. Il consiste à lire l'intégralité de la table, du premier bloc au dernier bloc. Une clause de filtrage peut être appliquée.

On retrouve ce noeud lorsque la requête nécessite de lire l'intégralité de la table :

```
cave=# EXPLAIN SELECT * FROM region;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on region  (cost=0.00..1.19 rows=19 width=15)
```

Ce noeud peut également filtrer directement les données, la présence de la clause `Filter` montre le filtre appliqué à la lecture des données :

```
cave=# EXPLAIN SELECT * FROM region WHERE id=5;
                       QUERY PLAN
-----------------------------------------------------------
 Seq Scan on region  (cost=0.00..1.24 rows=1 width=15)
   Filter: (id = 5)
```

Le coût d'accès pour ce type de noeud sera dépendant du nombre de blocs à parcourir et du paramètre `seq_page_cost`.

Il est possible d'avoir un parcours parallélisé d'une table sous certaines conditions (la première étant qu'il faut avoir au minimum une version 9.6). Pour que ce type de parcours soit valable, il faut que l'optimiseur soit persuadé que le problème sera le temps CPU et non la bande passante disque. Autrement dit, dans la majorité des cas, il faut un filtre pour que la parallélisation se déclenche et il faut que la table soit suffisamment volumineuse.

```
postgres=# CREATE TABLE t20 AS SELECT id FROM generate_series(1, 1000000) g(id);
postgres=# SET max_parallel_workers_per_gather TO 6;
postgres=# EXPLAIN SELECT * FROM t20 WHERE id<10000;
                              QUERY PLAN
-----------------------------------------------------------------------
 Gather  (cost=1000.00..11676.13 rows=10428 width=4)
   Workers Planned: 2
   ->  Parallel Seq Scan on t20  (cost=0.00..9633.33 rows=4345 width=4)
         Filter: (id < 10000)
(4 rows)
```

Ici, deux processus supplémentaires seront exécutés pour réaliser la requête. Dans le cas de ce type de parcours, chaque processus traite toutes les lignes d'un bloc. Enfin quand un processus a terminé de traiter son bloc, il regarde quel est le prochain bloc à traiter et le traite.

On retrouve le noeud `Function Scan` lorsqu'une requête utilise directement le résultat d'une fonction. C'est un noeud que l'on rencontre lorsqu'on utilise les fonctions d'informations systèmes de PostgreSQL :

```
postgres=# EXPLAIN SELECT * from pg_get_keywords();
                              QUERY PLAN
-----------------------------------------------------------------------
 Function Scan on pg_get_keywords  (cost=0.03..4.03 rows=400 width=65)
(1 ligne)
```

En dehors des différents parcours d'index, on retrouve également d'autres types de parcours, mais PostgreSQL les utilise rarement. Ils sont néanmoins détaillés en annexe.

3.8.2 PARCOURS D'INDEX

- Index Scan
- Index Only Scan
- Bitmap Index Scan
- Et leurs versions parallélisées

PostgreSQL dispose de trois moyens d'accéder aux données à travers les index.

Le noeud `Index Scan` est le premier qui a été disponible. Il consiste à parcourir les blocs d'index jusqu'à trouver les pointeurs vers les blocs contenant les données. PostgreSQL lit ensuite les données de la table qui sont pointées par l'index.

```
tpc=# EXPLAIN SELECT * FROM clients WHERE client_id = 10000;
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Scan using clients_pkey on clients  (cost=0.42..8.44 rows=1 width=52)
   Index Cond: (client_id = 10000)
(2 lignes)
```

Ce type de noeud ne permet pas d'extraire directement les données à retourner depuis l'index, sans passer par la lecture des blocs correspondants de la table. Le noeud `Index Only Scan` permet cette optimisation, à condition que les colonnes retournées fassent partie de l'index :

```
tpc=# EXPLAIN SELECT client_id FROM clients WHERE client_id = 10000;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Index Only Scan using clients_pkey on clients  (cost=0.42..8.44 rows=1 width=8)
   Index Cond: (client_id = 10000)
(2 lignes)
```

Enfin, on retrouve le dernier parcours sur des opérations de type *range scan*, c'est-à-dire où PostgreSQL doit retourner une plage de valeurs. On le retrouve également lorsque PostgreSQL doit combiner le résultat de la lecture de plusieurs index.

Contrairement à d'autres SGBD, un index bitmap n'a aucune existence sur disque. Il est créé en mémoire lorsque son utilisation a un intérêt. Le but est de diminuer les déplacements de la tête de lecture en découplant le parcours de l'index du parcours de la table :

- Lecture en un bloc de l'index ;
- Lecture en un bloc de la partie intéressante de la table (dans l'ordre physique de la table, pas dans l'ordre logique de l'index).

```
tpc=# EXPLAIN SELECT * FROM clients WHERE client_id BETWEEN 10000 AND 12000;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on clients  (cost=44.99..1201.32 rows=2007 width=52)
   Recheck Cond: ((client_id >= 10000) AND (client_id <= 12000))
   ->  Bitmap Index Scan on clients_pkey  (cost=0.00..44.49 rows=2007 width=0)
         Index Cond: ((client_id >= 10000) AND (client_id <= 12000))
(4 lignes)
```

On retrouve aussi des Bitmap Index Scan lorsqu'il s'agit de combiner le résultat de la lecture de plusieurs index :

```
tpc=# EXPLAIN SELECT * FROM clients WHERE client_id
tpc=# BETWEEN 10000 AND 12000 AND segment_marche = 'AUTOMOBILE';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on clients  (cost=478.25..1079.74 rows=251 width=8)
   Recheck Cond: ((client_id >= 10000) AND (client_id <= 12000)
                  AND (segment_marche = 'AUTOMOBILE'::bpchar))
   ->  BitmapAnd  (cost=478.25..478.25 rows=251 width=0)
     ->  Bitmap Index Scan on clients_pkey  (cost=0.00..44.49 rows=2007 width=0)
           Index Cond: ((client_id >= 10000) AND (client_id <= 12000))
     ->  Bitmap Index Scan on idx_clients_segmarche
                                        (cost=0.00..433.38 rows=18795 width=0)
           Index Cond: (segment_marche = 'AUTOMOBILE'::bpchar)
(7 lignes)
```

À partir de la version 10, une infrastructure a été mise en place pour permettre un parcours parallélisé d'un index. Cela donne donc les noeuds `Parallel Index Scan`, `Parallel Index Only Scan` et `Parallel Bitmap Heap Scan`. Cette infrastructure est actuellement uniquement utilisé pour les index Btree. Par contre, pour le bitmap scan, seul le parcours de la table est parallélisé, ce qui fait que tous les types d'index sont concernés.

3.8.3 NOEUDS DE JOINTURE

- PostgreSQL implémente les 3 algorithmes de jointures habituels :
  - Nested Loop (boucle imbriquée)
  - Hash Join (hachage de la table interne)
  - Merge Join (tri-fusion)
- Parallélisation possible
  - version 9.6 pour Nested Loop et Hash Join
  - version 10 pour Merge Join
- Et pour `EXISTS`, `IN` et certaines jointures externes :
  - Semi Join et Anti Join

Le choix du type de jointure dépend non seulement des données mises en oeuvre, mais elle dépend également beaucoup du paramétrage de PostgreSQL, notamment des paramètres `work_mem`, `seq_page_cost` et `random_page_cost`.

La `Nested Loop` se retrouve principalement quand on joint de petits ensembles de don-

nées :

```
sql1=# EXPLAIN SELECT * FROM commandes
sql1=# JOIN lignes_commandes USING (numero_commande)
sql1=# WHERE numero_commande < 1000;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=0.84..4161.14 rows=1121 width=154)
   ->  Index Scan using orders_pkey on commandes
                                            (cost=0.42..29.64 rows=280 width=80)
         Index Cond: (numero_commande < 1000)
   ->  Index Scan using lignes_commandes_pkey on lignes_commandes
                                            (cost=0.42..14.71 rows=5 width=82)
         Index Cond: (numero_commande = commandes.numero_commande)
```

Le `Hash Join` se retrouve également lorsque l'ensemble de la table interne est très petit. L'optimiseur réalise alors un hachage des valeurs de la colonne de jointure sur la table externe et réalise ensuite une lecture de la table externe et compare les hachages de la clé de jointure avec le/les hachage(s) obtenus à la lecture de la table interne.

```
sql1=# EXPLAIN SELECT * FROM commandes
sql1=# JOIN lignes_commandes USING (numero_commande);
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=10690.31..59899.18 rows=667439 width=154)
   Hash Cond: (lignes_commandes.numero_commande = commandes.numero_commande)
   ->  Seq Scan on lignes_commandes  (cost=0.00..16325.39 rows=667439 width=82)
   ->  Hash  (cost=6489.25..6489.25 rows=166725 width=80)
         ->  Seq Scan on commandes  (cost=0.00..6489.25 rows=166725 width=80)
```

La jointure par tri- fusion, ou `Merge Join` prend deux ensembles de données triés en entrée et restitue l'ensemble de données après jointure. Cette jointure est assez lourde à initialiser si PostgreSQL ne peut pas utiliser d'index, mais elle a l'avantage de retourner les données triées directement :

```
sql1=# EXPLAIN SELECT * FROM commandes
sql1=# JOIN lignes_commandes USING (numero_commande)
sql1=# ORDER BY numero_commande DESC;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Merge Join  (cost=1.40..64405.98 rows=667439 width=154)
   Merge Cond: (commandes.numero_commande = lignes_commandes.numero_commande)
   ->  Index Scan Backward using orders_pkey on commandes
                                    (cost=0.42..12898.63 rows=166725 width=80)
   ->  Index Scan Backward using lignes_commandes_pkey on lignes_commandes
                                    (cost=0.42..42747.64 rows=667439 width=82)
```

Il s'agit d'un algorithme de jointure particulièrement efficace pour traiter les volumes de

données importants.

Les clauses `EXISTS` et `NOT EXISTS` mettent également en oeuvre des algorithmes dérivés de semi et anti jointures. Par exemple avec la clause `EXISTS` :

```
sql1=# EXPLAIN
 SELECT count(*)
   FROM commandes c
  WHERE EXISTS (SELECT 1
                  FROM lignes_commandes l
                 WHERE c.date_commande > l.date_expedition
                   AND c.numero_commande = l.numero_commande);
                                QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=42439.18..42439.19 rows=1 width=0)
   ->  Hash Semi Join  (cost=27927.38..42321.76 rows=46967 width=0)
         Hash Cond: (c.numero_commande = l.numero_commande)
         Join Filter: (c.date_commande > l.date_expedition)
         ->  Seq Scan on commandes c (cost=0.00..6489.25 rows=166725 width=12)
         ->  Hash  (cost=16325.39..16325.39 rows=667439 width=12)
               ->  Seq Scan on lignes_commandes l
                                     (cost=0.00..16325.39 rows=667439 width=12)
```

On obtient un plan sensiblement identique, avec `NOT EXISTS`. Le noeud `Hash Semi Join` est remplacé par `Hash Anti Join` :

```
sql1=# EXPLAIN
SELECT *
  FROM commandes
 WHERE NOT EXISTS (SELECT 1
                     FROM lignes_commandes l
                    WHERE l.numero_commande = commandes.numero_commande);
                                 QUERY PLAN
-------------------------------------------------------------------------------
 Hash Anti Join  (cost=27276.38..47110.99 rows=25824 width=80)
   Hash Cond: (commandes.numero_commande = l.numero_commande)
   ->  Seq Scan on commandes  (cost=0.00..6489.25 rows=166725 width=80)
   ->  Hash  (cost=16325.39..16325.39 rows=667439 width=8)
         ->  Seq Scan on lignes_commandes l
                               (cost=0.00..16325.39 rows=667439 width=8)
```

PostgreSQL dispose de la parallélisation depuis la version 9.6. Cela ne concernait que les jointures de type Nested Loop et Hash Join. Quant au Merge Join, il a fallu attendre la version 10 pour que la parallélisation soit supportée.

3.8.4 NOEUDS DE TRIS ET DE REGROUPEMENTS

- Un seul noeud de tri :
  - Sort
- Regroupement/Agrégation :
  - Aggregate
  - HashAggregate
  - GroupAggregate
  - Partial Aggregate/Finalize Aggregate

Pour réaliser un tri, PostgreSQL ne dispose que d'un seul noeud pour réaliser cela : `Sort`. Son efficacité va dépendre du paramètre `work_mem` qui va définir la quantité de mémoire que PostgreSQL pourra utiliser pour un tri.

```
sql1=# explain (ANALYZE) SELECT * FROM lignes_commandes
sql1=# WHERE numero_commande = 1000 ORDER BY quantite;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Sort  (cost=15.57..15.58 rows=5 width=82)
       (actual time=0.096..0.097 rows=4 loops=1)
   Sort Key: quantite
   Sort Method: quicksort  Memory: 25kB
   ->  Index Scan using lignes_commandes_pkey on lignes_commande
                  (cost=0.42..15.51 rows=5 width=82)
                  (actual time=0.017..0.021 rows=4 loops=1)
         Index Cond: (numero_commande = 1000)
```

Si le tri ne tient pas en mémoire, l'algorithme de tri gère automatiquement le débordement sur disque :

```
sql1=# EXPLAIN (ANALYZE) SELECT * FROM commandes ORDER BY prix_total ;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Sort  (cost=28359.74..28776.55 rows=166725 width=80)
       (actual time=993.441..1157.935 rows=166725 loops=1)
   Sort Key: prix_total
   Sort Method: external merge  Disk: 15608kB
   ->  Seq Scan on commandes  (cost=0.00..6489.25 rows=166725 width=80)
                              (actual time=173.615..236.712 rows=166725 loops=1)
```

Cependant, si un index existe, PostgreSQL peut également utiliser un index pour récupérer les données triées directement :

```
sql1=# EXPLAIN SELECT * FROM commandes ORDER BY date_commande;
                                    QUERY PLAN
-------------------------------------------------------------------------------
```

```
 Index Scan using idx_commandes_date_commande on commandes
                          (cost=0.42..23628.15 rows=166725 width=80)
```

Dans n'importe quel ordre de tri :

```
sql1=# EXPLAIN SELECT * FROM commandes ORDER BY date_commande DESC;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan Backward using idx_commandes_date_commande on commandes
                                  (cost=0.42..23628.15 rows=166725 width=80)
```

Le choix du type d'opération de regroupement dépend non seulement des données mises en oeuvres, mais elle dépend également beaucoup du paramétrage de PostgreSQL, notamment du paramètre work_mem.

Concernant les opérations d'agrégations, on retrouve un noeud de type Aggregate lorsque la requête réalise une opération d'agrégation simple, sans regroupement :

```
sql1=# EXPLAIN SELECT count(*) FROM commandes;
                                QUERY PLAN
---------------------------------------------------------------------------
 Aggregate  (cost=4758.11..4758.12 rows=1 width=0)
   ->  Index Only Scan using commandes_client_id_idx on commandes
                                  (cost=0.42..4341.30 rows=166725 width=0)
```

Si l'optimiseur estime que l'opération d'agrégation tient en mémoire (paramètre work_mem), il va utiliser un noeud de type HashAggregate :

```
sql1=# EXPLAIN SELECT code_pays, count(*) FROM contacts GROUP BY code_pays;
                            QUERY PLAN
------------------------------------------------------------------
 HashAggregate  (cost=3982.02..3982.27 rows=25 width=3)
   ->  Seq Scan on contacts  (cost=0.00..3182.01 rows=160001 width=3)
```

L'inconvénient de ce noeud est que sa consommation mémoire n'est pas limitée par work_mem, il continuera malgré tout à allouer de la mémoire. Dans certains cas, heureusement très rares, l'optimiseur peut se tromper suffisamment pour qu'un noeud HashAggregate consomme plusieurs giga-octets de mémoire et ne sature la mémoire du serveur.

Lorsque l'optimiseur estime que le volume de données à traiter ne tient pas dans work_mem, il utilise plutôt l'algorithme GroupAggregate :

```
sql1=# explain select numero_commande, count(*)
sql1=# FROM lignes_commandes group by numero_commande;
                                QUERY PLAN
---------------------------------------------------------------------------
 GroupAggregate  (cost=0.42..47493.84 rows=140901 width=8)
```

```
   ->  Index Only Scan using lignes_commandes_pkey on lignes_commandes
                                  (cost=0.42..42747.64 rows=667439 width=8)
```

Le calcul d' un agrégat peut être parallélisé à partir de la version 9.6. Dans ce cas, deux noeuds sont utilisés : un pour le calcul partiel de chaque processus (Partial Aggregate), et un pour le calcul final (Finalize Aggregate). Voici un exemple de plan :

```
SELECT count(*), min(C1), max(C1) FROM t1;

                                   QUERY PLAN
--------------------------------------------------------------------------------
 Finalize Aggregate (actual time=1766.820..1766.820 rows=1 loops=1)
   ->  Gather (actual time=1766.767..1766.799 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate (actual time=1765.236..1765.236 rows=1 loops=3)
               ->  Parallel Seq Scan on t1
                           (actual time=0.021..862.430 rows=6666667 loops=3)
 Planning time: 0.072 ms
 Execution time: 1769.164 ms
(8 rows)
```

3.8.5 LES AUTRES NOEUDS

- Limit
- Unique
- Append (UNION ALL), Except, Intersect
- Gather
- InitPlan, Subplan, etc.

On rencontre le noeud `Limit` lorsqu'on limite le résultat avec l'ordre `LIMIT` :

```
sql1=# EXPLAIN SELECT * FROM commandes LIMIT 1;
                                QUERY PLAN
--------------------------------------------------------------------------
 Limit  (cost=0.00..0.04 rows=1 width=80)
   ->  Seq Scan on commandes  (cost=0.00..6489.25 rows=166725 width=80)
```

À noter, que le noeud `Sort` utilisera une méthode de tri appelée `top-N heapsort` qui permet d'optimiser le tri pour retourner les *n* premières lignes :

```
sql1=# EXPLAIN ANALYZE SELECT * FROM commandes ORDER BY prix_total LIMIT 5;
                                      QUERY PLAN
--------------------------------------------------------------------------------
 Limit  (cost=9258.49..9258.50 rows=5 width=80)
```

```
        (actual time=86.332..86.333 rows=5 loops=1)
   ->  Sort  (cost=9258.49..9675.30 rows=166725 width=80)
             (actual time=86.330..86.331 rows=5 loops=1)
         Sort Key: prix_total
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Seq Scan on commandes  (cost=0.00..6489.25 rows=166725 width=80)
                                    (actual time=3.683..22.687 rows=166725 loops=1)
```

On retrouve le noeud `Unique` lorsque l'on utilise `DISTINCT` pour dédoublonner le résultat d'une requête :

```
sql1=# EXPLAIN SELECT DISTINCT numero_commande FROM lignes_commandes;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Unique  (cost=0.42..44416.23 rows=140901 width=8)
   ->  Index Only Scan using lignes_commandes_pkey on lignes_commandes
                                  (cost=0.42..42747.64 rows=667439 width=8)
```

À noter qu'il est souvent plus efficace d'utiliser `GROUP BY` pour dédoublonner les résultats d'une requête :

```
sql1=# EXPLAIN (ANALYZE) SELECT DISTINCT numero_commande
sql1=# FROM lignes_commandes GROUP BY numero_commande;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Unique  (cost=0.42..44768.49 rows=140901 width=8)
         (actual time=0.047..357.745 rows=166724 loops=1)
   ->  Group  (cost=0.42..44416.23 rows=140901 width=8)
              (actual time=0.045..306.550 rows=166724 loops=1)
         ->  Index Only Scan using lignes_commandes_pkey on lignes_commandes
                   (cost=0.42..42747.64 rows=667439 width=8)
                   (actual time=0.040..197.817 rows=667439 loops=1)
               Heap Fetches: 667439
 Total runtime: 365.315 ms

sql1=# EXPLAIN (ANALYZE) SELECT numero_commande
sql1=# FROM lignes_commandes GROUP BY numero_commande;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Group  (cost=0.42..44416.23 rows=140901 width=8)
        (actual time=0.053..302.875 rows=166724 loops=1)
   ->  Index Only Scan using lignes_commandes_pkey on lignes_commandes
                (cost=0.42..42747.64 rows=667439 width=8)
                (actual time=0.046..194.495 rows=667439 loops=1)
         Heap Fetches: 667439
 Total runtime: 310.506 ms
```

Le gain est infime, 50 millisecondes environ sur cette requête, mais laisse présager des

gains sur une volumétrie plus importante.

Les noeuds `Append`, `Except` et `Intersect` se rencontrent lorsqu'on utilise les opérateurs ensemblistes `UNION`, `EXCEPT` et `INTERSECT`. Par exemple, avec `UNION ALL` :

```
sql1=# EXPLAIN
SELECT * FROM pays
 WHERE region_id = 1
 UNION ALL
SELECT * FROM pays
 WHERE region_id = 2;
                            QUERY PLAN
-------------------------------------------------------------------
 Append  (cost=0.00..2.73 rows=10 width=53)
   ->  Seq Scan on pays  (cost=0.00..1.31 rows=5 width=53)
         Filter: (region_id = 1)
   ->  Seq Scan on pays pays_1  (cost=0.00..1.31 rows=5 width=53)
         Filter: (region_id = 2)
```

Le noeud Gather a été introduit en version 9.6 et est utilisé comme noeud de rassemblement des données pour les plans parallélisés.

Le noeud InitPlan apparaît lorsque PostgreSQL a besoin d'exécuter une première sous-requête pour ensuite exécuter le reste de la requête. Il est assez rare :

```
sql1=# EXPLAIN SELECT *,
sql1=# (SELECT nom_region FROM regions WHERE region_id=1)
sql1=# FROM pays WHERE region_id = 1;
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on pays  (cost=1.06..2.38 rows=5 width=53)
   Filter: (region_id = 1)
   InitPlan 1 (returns $0)
     ->  Seq Scan on regions  (cost=0.00..1.06 rows=1 width=26)
           Filter: (region_id = 1)
```

Le noeud SubPlan est utilisé lorsque PostgreSQL a besoin d'exécuter une sous-requête pour filtrer les données :

```
sql1=# EXPLAIN
SELECT * FROM pays
 WHERE region_id NOT IN (SELECT region_id FROM regions
                         WHERE nom_region = 'Europe');
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on pays  (cost=1.06..2.38 rows=12 width=53)
   Filter: (NOT (hashed SubPlan 1))
   SubPlan 1
```

```
->  Seq Scan on regions  (cost=0.00..1.06 rows=1 width=4)
      Filter: (nom_region = 'Europe'::bpchar)
```

D'autres types de noeud peuvent également être trouvés dans les plans d'exécution. L'annexe décrit tous ces noeuds en détail.

3.9 PROBLÈMES LES PLUS COURANTS

- L'optimiseur se trompe parfois
  - mauvaises statistiques
  - écriture particulière de la requête
  - problèmes connus de l'optimiseur

L'optimiseur de PostgreSQL est sans doute la partie la plus complexe de PostgreSQL. Il se trompe rarement, mais certains facteurs peuvent entraîner des temps d'exécution très lents, voire catastrophiques de certaines requêtes.

3.9.1 COLONNES CORRÉLÉES

```
SELECT * FROM t1 WHERE c1=1 AND c2=1
```

- `c1=1` est vrai pour 20% des lignes
- `c2=1` est vrai pour 10% des lignes
- Le planificateur va penser que le résultat complet ne récupérera que 20% * 10% (soit 2%) des lignes
  - En réalité, ça peut aller de 0 à 10% des lignes
- Problème corrigé en version 10
  - `CREATE STATISTICS` pour des statistiques multi-colonnes

PostgreSQL conserve des statistiques par colonne simple. Dans l'exemple ci- dessus, le planificateur sait que l'estimation pour `c1=1` est de 20% et que l'estimation pour `c2=1` est de 10%. Par contre, il n'a aucune idée de l'estimation pour `c1=1 AND c2=1`. En réalité, l'estimation pour cette formule va de 0 à 10% mais le planificateur doit statuer sur une seule valeur. Ce sera le résultat de la multiplication des deux estimations, soit 2% (20% * 10%).

La version 10 de PostgreSQL corrige cela en ajoutant la possibilité d'ajouter des statistiques sur plusieurs colonnes spécifiques. Ce n'est pas automatique, il faut créer un objet statistique avec l'ordre `CREATE STATISTICS`.

3.9.2 MAUVAISE ÉCRITURE DE PRÉDICATS

```
SELECT *
FROM commandes
WHERE extract('year' from date_commande) = 2014;
```

- L'optimiseur n'a pas de statistiques sur le résultat de la fonction `extract`
  - il estime la sélectivité du prédicat à 0.5%.

Dans un prédicat, lorsque les valeurs des colonnes sont transformées par un calcul ou par une fonction, l'optimiseur n'a aucun moyen pour connaître la sélectivité d'un prédicat. Il utilise donc une estimation codée en dur dans le code de l'optimiseur : 0,5% du nombre de lignes de la table.

Dans la requête suivante, l'optimiseur estime que la requête va ramener 834 lignes :

```
sql1=# EXPLAIN SELECT * FROM commandes
sql1=# WHERE extract('year' from date_commande) = 2014;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on commandes  (cost=0.00..7739.69 rows=834 width=80)
   Filter:
    (date_part('year'::text, (date_commande)::timestamp without time zone) =
        2014::double precision)
(2 lignes)
```

Ces 834 lignes correspondent à 0,5% de la table `commandes` :

```
sql1=# SELECT relname, reltuples, round(reltuples*0.005) AS estimé
  FROM pg_class
 WHERE relname = 'commandes';
  relname  | reltuples | estimé
-----------+-----------+--------
 commandes |    166725 |    834
(1 ligne)
```

3.9.3 PROBLÈME AVEC LIKE

```
SELECT * FROM t1 WHERE c2 LIKE 'x%';
```

- PostgreSQL peut utiliser un index dans ce cas
- Si l'encodage n'est pas C, il faut déclarer l'index avec une classe d'opérateur
  - varchar_pattern_ops, text_pattern_ops, etc

- En 9.1, il faut aussi faire attention au collationnement
- Ne pas oublier pg_trgm (surtout en 9.1) et FTS

Dans le cas d'une recherche avec préfixe, PostgreSQL peut utiliser un index sur la colonne. Il existe cependant une spécificité à PostgreSQL. Si l'encodage est autre chose que C, il faut utiliser une classe d'opérateur lors de la création de l'index. Cela donnera par exemple :

```
CREATE INDEX i1 ON t1 (c2 varchar_pattern_ops);
```

De plus, à partir de la version 9.1, il est important de faire attention au collationnement. Si le collationnement de la requête diffère du collationnement de la colonne de l'index, l'index ne pourra pas être utilisé.

3.9.4 PROBLÈMES AVEC LIMIT

- Exemple
  - EXPLAIN avec LIMIT 199
  - EXPLAIN avec LIMIT 200
- Corrigé en 9.2

Le contexte :

```
CREATE TABLE t1 (
    c1 integer PRIMARY KEY
);
INSERT INTO t1 SELECT generate_series(1, 1000);

CREATE TABLE t2 (
    c2 integer
);
INSERT INTO t2 SELECT generate_series(1, 1000);

ANALYZE;
```

Voici un problème survenant dans les versions antérieures à la 9.2.

```
EXPLAIN SELECT * FROM t1
WHERE c1 IN (SELECT c2 FROM t2 LIMIT 199);
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Semi Join (cost=7.46..27.30 rows=199 width=4)
   Hash Cond: (t1.c1 = t2.c2)
   -> Seq Scan on t1 (cost=0.00..15.00 rows=1000 width=4)
   -> Hash (cost=4.97..4.97 rows=199 width=4)
```

```
         -> Limit  (cost=0.00..2.98 rows=199 width=4)
               -> Seq Scan on t2  (cost=0.00..15.00 rows=1000 width=4)
(6 rows)
```

Tout se passe bien. PostgreSQL fait tout d'abord un parcours séquentiel sur la table `t2` et ne récupère que les 199 premières lignes grâce à la clause `LIMIT`. Le hachage se fait sur les 199 lignes et une comparaison est faite pour chaque ligne de `t1`.

Maintenant, cherchons à récupérer une ligne de plus avec un `LIMIT` à 200 :

```
EXPLAIN SELECT * FROM t1
WHERE c1 IN (SELECT c2 FROM t2 LIMIT 200);
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=10.00..30.75 rows=500 width=4)
   Hash Cond: (t1.c1 = t2.c2)
   -> Seq Scan on t1  (cost=0.00..15.00 rows=1000 width=4)
   -> Hash  (cost=7.50..7.50 rows=200 width=4)
         -> HashAggregate  (cost=5.50..7.50 rows=200 width=4)
               -> Limit  (cost=0.00..3.00 rows=200 width=4)
                     -> Seq Scan on t2  (cost=0.00..15.00 rows=1000
                                         width=4)
(7 rows)
```

La requête a légèrement changé : on passe d'un `LIMIT 199` à un `LIMIT 200`. L'estimation explose, elle passe de 199 lignes (estimation exacte) à 500 lignes (estimation plus que doublée). En fait, le nombre de lignes est calculé très simplement : nombre de lignes de la table t1 multiplié par 0,5. C'est codé en dur. La raison, jusqu'à PostgreSQL 9.1, est que par défaut une table sans statistiques est estimée posséder 200 valeurs distinctes. Quand l'optimiseur rencontre donc 200 enregistrements distincts en estimation, il pense que la fonction d'estimation de valeurs distinctes n'a pas de statistiques et lui a retourné une valeur par défaut, et applique donc un algorithme de sélectivité par défaut, au lieu de l'algorithme plus fin utilisé en temps normal.

Sur cet exemple, cela n'a pas un gros impact vu la quantité de données impliquées et le schéma choisi. Par contre, ça fait passer une requête de 9ms à 527ms si le `LIMIT 199` est passé à un `LIMIT 200` pour la même requête sur une table plus conséquente.

Ce problème est réglé en version 9.2 :

```
EXPLAIN SELECT * FROM t1
WHERE c1 IN (SELECT c2 FROM t2 LIMIT 200);
                               QUERY PLAN
------------------------------------------------------------------------
 Hash Semi Join  (cost=7.46..27.30 rows=200 width=4)
   Hash Cond: (t1.c1 = t2.c2)
   -> Seq Scan on t1  (cost=0.00..15.00 rows=1000 width=4)
```

```
   -> Hash (cost=4.97..4.97 rows=200 width=4)
         -> Limit (cost=0.00..2.98 rows=200 width=4)
               -> Seq Scan on t2 (cost=0.00..15.00 rows=1000 width=4)
(6 rows)
```

3.9.5 DELETE LENT

- DELETE lent
- Généralement un problème de clé étrangère

```
Delete  (actual time=111.251..111.251 rows=0 loops=1)
  ->  Hash Join  (actual time=1.094..21.402 rows=9347 loops=1)
        ->  Seq Scan on lot_a30_descr_lot
            (actual time=0.007..11.248 rows=34934 loops=1)
        ->  Hash  (actual time=0.501..0.501 rows=561 loops=1)
              ->  Bitmap Heap Scan on lot_a10_pdl
                  (actual time=0.121..0.326 rows=561 loops=1)
                    Recheck Cond: (id_fantoir_commune = 320013)
                    ->  Bitmap Index Scan on...
                        (actual time=0.101..0.101 rows=561 loops=1)
                          Index Cond: (id_fantoir_commune = 320013)
Trigger for constraint fk_lotlocal_lota30descrlot:
  time=1010.358 calls=9347
Trigger for constraint fk_nonbatia21descrsuf_lota30descrlot:
  time=2311695.025 calls=9347
Total runtime: 2312835.032 ms
```

Parfois, un DELETE peut prendre beaucoup de temps à s'exécuter. Cela peut être dû à un grand nombre de lignes à supprimer. Cela peut aussi être dû à la vérification des contraintes étrangères.

Dans l'exemple ci-dessus, le DELETE met 38 minutes à s'exécuter (2312835 ms), pour ne supprimer aucune ligne. En fait, c'est la vérification de la contrainte fk_nonbatia21descrsuf_lota30descrlot qui prend pratiquement tout le temps. C'est d'ailleurs pour cette raison qu'il est recommandé de positionner des index sur les clés étrangères, car cet index permet d'accélérer la recherche liée à la contrainte.

Attention donc aux contraintes de clés étrangères pour les instructions DML.

3.9.6 DÉDOUBLONNAGE

```
SELECT DISTINCT t1.* FROM t1 JOIN t2 ON (t1.id=t2.t1_id);
```

- DISTINCT est souvent utilisé pour dédoublonner les lignes de t1
  - mais génère un tri qui pénalise les performances
- GROUP BY est plus rapide
- Une clé primaire permet de dédoublonner efficacement des lignes
  - à utiliser avec GROUP BY

L'exemple ci-dessous montre une requête qui récupère les commandes qui ont des lignes de commandes et réalise le dédoublonnage avec DISTINCT. Le plan d'exécution montre une opération de tri qui a nécessité un fichier temporaire de 60Mo. Toutes ces opérations sont assez gourmandes, la requête répond en 5,9s :

```
tpc=# EXPLAIN (ANALYZE on, COSTS off)
tpc-# SELECT DISTINCT commandes.* FROM commandes
sql1=# JOIN lignes_commandes USING (numero_commande);
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Unique (actual time=5146.904..5833.600 rows=168749 loops=1)
   ->  Sort (actual time=5146.902..5307.633 rows=675543 loops=1)
         Sort Key: commandes.numero_commande, commandes.client_id,
                   commandes.etat_commande, commandes.prix_total,
                   commandes.date_commande, commandes.priorite_commande,
                   commandes.vendeur, commandes.priorite_expedition,
                   commandes.commentaire
         Sort Method: external sort  Disk: 60760kB
         ->  Merge Join (actual time=0.061..601.674 rows=675543 loops=1)
               Merge Cond: (commandes.numero_commande =
                            lignes_commandes.numero_commande)
               ->  Index Scan using orders_pkey on commandes
                     (actual time=0.026..71.544 rows=168750 loops=1)
               ->  Index Only Scan using lignes_com_pkey on lignes_commandes
                     (actual time=0.025..175.321 rows=675543 loops=1)
                     Heap Fetches: 0
 Total runtime: 5849.996 ms
```

En restreignant les colonnes récupérées à celle réellement intéressante et en utilisant GROUP BY au lieu du DISTINCT, le temps d'exécution tombe à 4,5s :

```
tpc=# EXPLAIN (ANALYZE on, COSTS off)
SELECT commandes.numero_commande, commandes.etat_commande,
    commandes.prix_total, commandes.date_commande,
    commandes.priorite_commande, commandes.vendeur,
    commandes.priorite_expedition
 FROM commandes
 JOIN lignes_commandes
      USING (numero_commande)
 GROUP BY commandes.numero_commande, commandes.etat_commande,
```

```
         commandes.prix_total, commandes.date_commande,
         commandes.priorite_commande, commandes.vendeur,
         commandes.priorite_expedition;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Group (actual time=4025.069..4663.992 rows=168749 loops=1)
   ->  Sort (actual time=4025.065..4191.820 rows=675543 loops=1)
         Sort Key: commandes.numero_commande, commandes.etat_commande,
                   commandes.prix_total, commandes.date_commande,
                   commandes.priorite_commande, commandes.vendeur,
                   commandes.priorite_expedition
         Sort Method: external sort  Disk: 46232kB
         ->  Merge Join (actual time=0.062..579.852 rows=675543 loops=1)
               Merge Cond: (commandes.numero_commande =
                            lignes_commandes.numero_commande)
               ->  Index Scan using orders_pkey on commandes
                            (actual time=0.027..70.212 rows=168750 loops=1)
               ->  Index Only Scan using lignes_com_pkey on lignes_commandes
                            (actual time=0.026..170.555 rows=675543 loops=1)
                     Heap Fetches: 0
 Total runtime: 4676.829 ms
```

Mais, à partir de PostgreSQL 9.1, il est possible d'améliorer encore les temps d'exécution de cette requête. Dans le plan d'exécution précédent, on voit que l'opération `Sort` est très gourmande car le tri des lignes est réalisé sur plusieurs colonnes. Or, la table `commandes` a une clé primaire sur la colonne `numero_commande`. Cette clé primaire permet d'assurer que toutes les lignes sont uniques dans la table `commandes`. Si l'opération `GROUP BY` ne porte plus que la clé primaire, PostgreSQL peut utiliser le résultat de la lecture par index sur commandes pour faire le regroupement. Le temps d'exécution passe à environ 580ms :

```
tpc=# EXPLAIN (ANALYZE on, COSTS off)
SELECT commandes.numero_commande, commandes.etat_commande,
    commandes.prix_total, commandes.date_commande,
    commandes.priorite_commande, commandes.vendeur,
    commandes.priorite_expedition
  FROM commandes
  JOIN lignes_commandes
       USING (numero_commande)
 GROUP BY commandes.numero_commande;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Group (actual time=0.067..580.198 rows=168749 loops=1)
   ->  Merge Join (actual time=0.061..435.154 rows=675543 loops=1)
         Merge Cond: (commandes.numero_commande =
                      lignes_commandes.numero_commande)
```

```
        ->  Index Scan using orders_pkey on commandes
                    (actual time=0.027..49.784 rows=168750 loops=1)
        ->  Index Only Scan using lignes_commandes_pkey on lignes_commandes
                    (actual time=0.025..131.606 rows=675543 loops=1)
              Heap Fetches: 0
Total runtime: 584.624 ms
```

Les opérations de dédoublonnages sont régulièrement utilisées pour assurer que les lignes retournées par une requête apparaissent de manière unique. Elles sont souvent inutiles, ou peuvent à minima être largement améliorées en utilisant les propriétés du modèle de données (les clés primaires) et des opérations plus adéquates (`GROUP BY clé_primaire`). Lorsque vous rencontrez des requêtes utilisant DISTINCT, vérifiez que le DISTINCT est vraiment pertinent ou s'il ne peut pas être remplacé par un GROUP BY qui pourrait tirer partie de la lecture d'un index.

Pour aller plus loin, n'hésitez pas à consulter cet article de blog[19] .

3.9.7 INDEX INUTILISÉS

- Trop de lignes retournées
- Prédicat incluant une transformation :

  ```
  WHERE col1 + 2 > 5
  ```

- Statistiques pas à jour ou peu précises
- Opérateur non-supporté par l'index :

  ```
  WHERE col1 <> 'valeur';
  ```

- Paramétrage de PostgreSQL : `effective_cache_size`

PostgreSQL offre de nombreuses possibilités d'indexation des données :

- Type d'index : B-tree, GiST, GIN, SP-GiST, BRIN et hash.
- Index multi-colonnes : `CREATE INDEX ... ON (col1, col2...);`
- Index partiel : `CREATE INDEX ... WHERE colonne = valeur`
- Index fonctionnel : `CREATE INDEX ... ON (fonction(colonne))`
- Extension offrant des fonctionnalités supplémentaires : pg_trgm

Malgré toutes ces possibilités, une question revient souvent lorsqu'un index vient d'être ajouté : pourquoi cet index n'est pas utilisé ?

[19] http://www.depesz.com/index.php/2010/04/19/getting-unique-elements/

L'optimiseur de PostgreSQL est très avancé et il y a peu de cas où il est mis en défaut. Malgré cela, certains index ne sont pas utilisés comme on le souhaiterait. Il peut y avoir plusieurs raisons à cela.

Problèmes de statistiques

Le cas le plus fréquent concerne les statistiques qui ne sont pas à jour. Cela arrive souvent après le chargement massif d'une table ou une mise à jour massive sans avoir fait une nouvelle collecte des statistiques à l'issue de ces changements.

On utilisera l'ordre `ANALYZE table` pour déclencher explicitement la collecte des statistiques après un tel traitement. En effet, bien qu'autovacuum soit présent, il peut se passer un certain temps entre le moment où le traitement est fait et le moment où autovacuum déclenche une collecte de statistiques. Ou autovacuum peut ne simplement pas se déclencher car le traitement complet est imbriqué dans une seule transaction.

Un traitement batch devra comporter des ordres `ANALYZE` juste après les ordres SQL qui modifient fortement les données :

```
COPY table_travail FROM '/tmp/fichier.csv';
ANALYZE table_travail;
SELECT ... FROM table_travail;
```

Un autre problème qui peut se poser avec les statistiques concerne les tables de très forte volumétrie. Dans certain cas, l'échantillon de données ramené par `ANALYZE` n'est pas assez précis pour donner à l'optimiseur de PostgreSQL une vision suffisamment précise des données. Il choisira alors de mauvais plans d'exécution.

Il est possible d'augmenter la précision de l'échantillon de données ramené à l'aide de l'ordre :

```
ALTER TABLE ... ALTER COLUMN ... SET STATISTICS ...;
```

Problèmes de prédicats

Dans d'autres cas, les prédicats d'une requête ne permettent pas à l'optimiseur de choisir un index pour répondre à une requête. C'est le cas lorsque le prédicat inclut une transformation de la valeur d'une colonne.

L'exemple suivant est assez naïf, mais démontre bien le problème :

```
SELECT * FROM table WHERE col1 + 10 = 10;
```

Avec une telle construction, l'optimiseur ne saura pas tirer partie d'un quelconque index, à moins d'avoir créé un index fonctionnel sur `col1 + 10`, mais cet index est largement contre-productif par rapport à une réécriture de la requête.

Ce genre de problème se rencontre plus souvent sur des prédicats sur des dates :

```
SELECT * FROM table WHERE date_trunc('month', date_debut) = 12
```

ou encore plus fréquemment rencontré :

```
SELECT * FROM table WHERE extract('year' from date_debut) = 2013
```

Opérateurs non-supportés

Les index B-tree supportent la plupart des opérateurs généraux sur les variables scalaires ((entiers, chaînes, dates, mais pas types composés comme géométries, hstore...)), mais pas la différence (<> ou !=). Par nature, il n'est pas possible d'utiliser un index pour déterminer *toutes les valeurs sauf une*. Mais ce type de construction est parfois utilisé pour exclure les valeurs les plus fréquentes d'une colonne. Dans ce cas, il est possible d'utiliser un index partiel, qui en plus sera très petit car il n'indexera qu'une faible quantité de données par rapport à la totalité de la table :

```
CREATE TABLE test (id serial PRIMARY KEY, v integer);
INSERT INTO test (v) SELECT 0 FROM generate_series(1, 10000);
INSERT INTO test (v) SELECT 1;
ANALYZE test;
CREATE INDEX idx_test_v ON test(v);
EXPLAIN SELECT * FROM test WHERE v <> 0;
                      QUERY PLAN
-------------------------------------------------------
 Seq Scan on test  (cost=0.00..170.03 rows=1 width=8)
   Filter: (v <> 0)

DROP INDEX idx_test_v;
```

La création d'un index partiel permet d'en tirer partie :

```
CREATE INDEX idx_test_v_partiel ON test (v) WHERE v<>0;
CREATE INDEX
Temps : 67,014 ms
postgres=# EXPLAIN SELECT * FROM test WHERE v <> 0;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using idx_test_v_partiel on test  (cost=0.00..8.27 rows=1 width=8)
```

Paramétrage de PostgreSQL

Plusieurs paramètres de PostgreSQL influencent le choix ou non d'un index :

- `random_page_cost` : indique à PostgreSQL la vitesse d'un accès aléatoire par rapport à un accès séquentiel (`seq_page_cost`).
- `effective_cache_size` : indique à PostgreSQL une estimation de la taille du cache disque du système.

Le paramètre `random_page_cost` a une grande influence sur l'utilisation des index en général. Il indique à PostgreSQL le coût d'un accès disque aléatoire. Il est à comparer au paramètre `seq_page_cost` qui indique à PostgreSQL le coût d'un accès disque séquentiel. Ces coûts d'accès sont purement arbitraires et n'ont aucune réalité physique. Dans sa configuration par défaut, PostgreSQL estime qu'un accès aléatoire est 4 fois plus coûteux qu'un accès séquentiel. Les accès aux index étant par nature aléatoires alors que les parcours de table étant par nature séquentiels, modifier ce paramètre revient à favoriser l'un par rapport à l'autre. Cette valeur est bonne dans la plupart des cas. Mais si le serveur de bases de données dispose d'un système disque rapide, c'est-à-dire une bonne carte RAID et plusieurs disques SAS rapides en RAID10, ou du SSD, il est possible de baisser ce paramètre à 3 voir 2.

Enfin, le paramètre `effective_cache_size` indique à PostgreSQL une estimation de la taille du cache disque du système. Une bonne pratique est de positionner ce paramètre à 2/3 de la quantité totale de RAM du serveur. Sur un système Linux, il est possible de donner une estimation plus précise en s'appuyant sur la valeur de colonne `cached` de la commande `free`. Mais si le cache n'est que peu utilisé, la valeur trouvée peut être trop basse pour pleinement favoriser l'utilisation des index.

Pour aller plus loin, n'hésitez pas à consulter cet article de blog[20]

3.9.8 ÉCRITURE DU SQL

- `NOT IN` avec une sous-requête
  - à remplacer par `NOT EXISTS`
- Utilisation systématique de `UNION` au lieu de `UNION ALL`
  - entraîne un tri systématique
- Sous-requête dans le `SELECT`
  - utiliser `LATERAL`

La façon dont une requête SQL est écrite peut aussi avoir un effet négatif sur les performances. Il n'est pas possible d'écrire tous les cas possibles, mais certaines formes d'écritures reviennent souvent.

La clause `NOT IN` n'est pas performance lorsqu'elle est utilisée avec une sous-requête. L'optimiseur ne parvient pas à exécuter ce type de requête efficacement.

```
SELECT *
  FROM commandes
```

[20] http://www.depesz.com/index.php/2010/09/09/why-is-my-index-not-being-used/

```
 WHERE numero_commande NOT IN (SELECT numero_commande
                               FROM lignes_commandes);
```

Il est nécessaire de la réécrire avec la clause `NOT EXISTS`, par exemple :

```
SELECT *
  FROM commandes
 WHERE NOT EXISTS (SELECT 1
                   FROM lignes_commandes l
                   WHERE l.numero_commande = commandes.numero_commande);
```

3.10 OUTILS

- pgAdmin3
- explain.depesz.com
- pev
- auto_explain
- plantuner

Il existe quelques outils intéressants dans le cadre du planificateur : deux applications externes pour mieux appréhender un plan d'exécution, deux modules pour changer le comportement du planificateur.

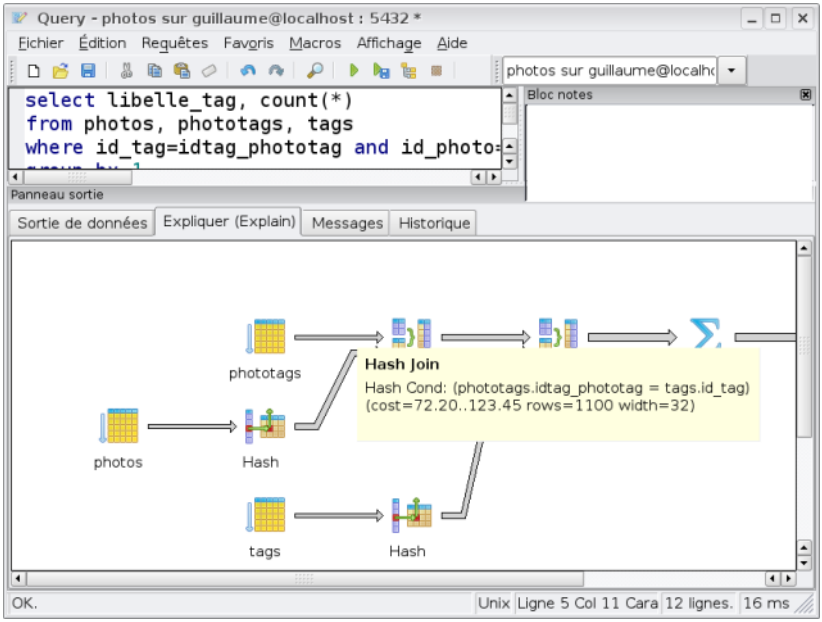
3.10.1 PGADMIN3

- Vision graphique d'un EXPLAIN
- Une icône par nœud
- La taille des flèches dépend de la quantité de données
- Le détail de chaque nœud est affiché en survolant les nœuds

pgAdmin propose depuis très longtemps un affichage graphique de l' `EXPLAIN`. Cet affichage est intéressant car il montre simplement l'ordre dans lequel les opérations sont effectuées. Chaque nœud est représenté par une icône. Les flèches entre chaque nœud indiquent où sont envoyés les flux de données, la taille de la flèche précisant la volumétrie des données.

Les statistiques ne sont affichées qu'en survolant les nœuds.

3.10.2 PGADMIN3 - COPIE D'ÉCRAN



Voici un exemple d'un `EXPLAIN` graphique réalisé par pgAdmin. En passant la souris sur les nœuds, un message affiche les informations statistiques sur le nœud.

3.10.3 SITE EXPLAIN.DEPESZ.COM

- Site web proposant un affichage particulier du EXPLAIN ANALYZE
- Il ne travaille que sur les informations réelles
- Les lignes sont colorées pour indiquer les problèmes
  - Blanc, tout va bien
  - Jaune, inquiétant
  - Marron, plus inquiétant
  - Rouge, très inquiétant
- Installable en local

Hubert Lubaczewski est un contributeur très connu dans la communauté PostgreSQL. Il publie notamment un grand nombre d'articles sur les nouveautés des prochaines versions.

17.12

Cependant, il est aussi connu pour avoir créé un site web d'analyse des plans d'exécution. Ce site web est disponible à cette adresse[21] .

Il suffit d'aller sur ce site, de coller le résultat d'un `EXPLAIN ANALYZE`, et le site affichera le plan d'exécution avec des codes couleurs pour bien distinguer les nœuds performants des autres.

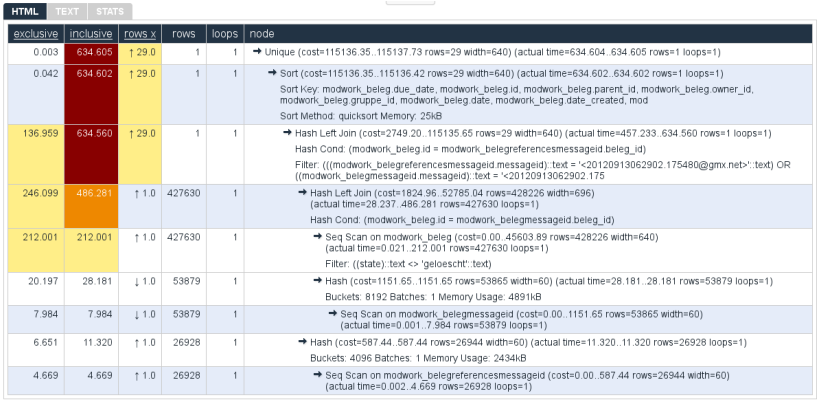
Le code couleur est simple :

- Blanc, tout va bien
- Jaune, inquiétant
- Marron, plus inquiétant
- Rouge, très inquiétant

Plutôt que d'utiliser ce serveur web, il est possible d'installer ce site en local :

- le module explain en Perl[22]
- la partie site web[23]

3.10.4 EXPLAIN.DEPESZ.COM - COPIE D'ÉCRAN



HTML | TEXT | STATS

exclusive	inclusive	rows x	rows	loops	node
0.003	634.605	↑ 29.0	1	1	→ Unique (cost=115136.35..115137.73 rows=29 width=640) (actual time=634.604..634.605 rows=1 loops=1)
0.042	634.602	↑ 29.0	1	1	→ Sort (cost=115136.35..115136.42 rows=29 width=640) (actual time=634.602..634.602 rows=1 loops=1) Sort Key: modwork_beleg.due_date, modwork_beleg.id, modwork_beleg.parent_id, modwork_beleg.owner_id, modwork_beleg.gruppe_id, modwork_beleg.date, modwork_beleg.date_created, mod Sort Method: quicksort Memory: 25kB
136.959	634.560	↑ 29.0	1	1	→ Hash Left Join (cost=2749.20..115135.65 rows=29 width=640) (actual time=457.233..634.560 rows=1 loops=1) Hash Cond: (modwork_beleg.id = modwork_belegreferencesmessageid.beleg_id) Filter: (((modwork_belegreferencesmessageid.messageid)::text = '<20120913062902.175480@gmx.net>'::text) OR ((modwork_belegmessageid.messageid)::text = '<20120913062902.175
246.099	486.281	↑ 1.0	427630	1	→ Hash Left Join (cost=1824.96..52785.04 rows=428226 width=696) (actual time=28.237..486.281 rows=427630 loops=1) Hash Cond: (modwork_beleg.id = modwork_belegmessageid.beleg_id)
212.001	212.001	↑ 1.0	427630	1	→ Seq Scan on modwork_beleg (cost=0.00..45603.89 rows=428226 width=640) (actual time=0.021..212.001 rows=427630 loops=1) Filter: ((state)::text <> 'geloescht'::text)
20.197	28.181	↓ 1.0	53879	1	→ Hash (cost=1151.65..1151.65 rows=53865 width=60) (actual time=28.181..28.181 rows=53879 loops=1) Buckets: 8192 Batches: 1 Memory Usage: 4891kB
7.984	7.984	↓ 1.0	53879	1	→ Seq Scan on modwork_belegmessageid (cost=0.00..1151.65 rows=53865 width=60) (actual time=0.001..7.984 rows=53879 loops=1)
6.651	11.320	↑ 1.0	26928	1	→ Hash (cost=587.44..587.44 rows=26944 width=60) (actual time=11.320..11.320 rows=26928 loops=1) Buckets: 4096 Batches: 1 Memory Usage: 2434kB
4.669	4.669	↑ 1.0	26928	1	→ Seq Scan on modwork_belegreferencesmessageid (cost=0.00..587.44 rows=26944 width=60) (actual time=0.002..4.669 rows=26928 loops=1)

Cet exemple montre un affichage d'un plan sur le site explain.depesz.com.

Voici la signification des différentes colonnes :

[21] http://explain.depesz.com
[22] https://github.com/depesz/Pg--Explain
[23] https://github.com/depesz/explain.depesz.com

- `Exclusive`, durée passée exclusivement sur un nœud ;
- `Inclusive`, durée passée sur un nœud et ses fils ;
- `Rows x`, facteur d'échelle de l'erreur d'estimation du nombre de lignes ;
- `Rows`, nombre de lignes renvoyées ;
- `Loops`, nombre de boucles.

Sur une exécution de 600 ms, un tiers est passé à lire la table avec un parcours séquentiel.

3.10.5 SITE PEV

- Site web proposant un affichage particulier du EXPLAIN ANALYZE
  - mais différent de celui de Depesz
- Fournir un plan d'exécution en JSON
- Installable en local

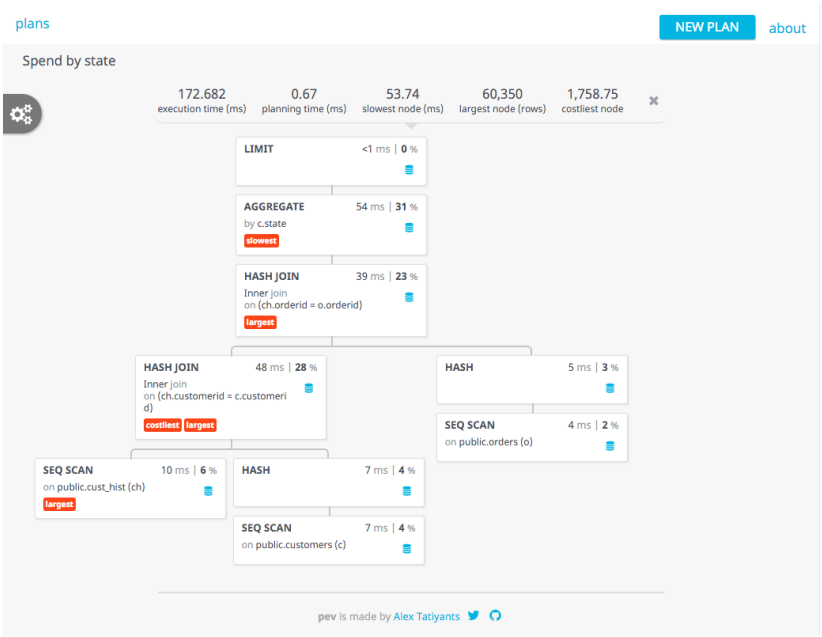
PEV est un outil librement téléchargeable sur ce dépôt github[24] . Il offre un affichage graphique du plan d'exécution et indique le nœud le plus coûteux, le plus long, le plus volumineux, etc.

Il est utilisable sur internet[25] mais aussi installable en local.

---

[24] https://github.com/AlexTatiyants/pev
[25] http://tatiyants.com/pev/#/plans

3.10.6 PEV - COPIE D'ÉCRAN



3.10.7 EXTENSION AUTO_EXPLAIN

- Extension pour PostgreSQL >= 8.4
- Connaître les requêtes lentes est bien
- Mais difficile de connaître leur plan d'exécution au moment où elles ont été lentes
- D'où le module auto_explain

Le but est donc de tracer automatiquement le plan d'exécution des requêtes. Pour éviter de trop écrire dans les fichiers de trace, il est possible de ne tracer que les requêtes dont la durée d'exécution a dépassé une certaine limite. Pour cela, il faut configurer le paramètre `auto_explain.log_min_duration`. D'autres options existent, qui permettent d'activer ou non certaines options du `EXPLAIN` : `log_analyze`, `log_verbose`, `log_buffers`, `log_format`.

3.10.8 EXTENSION PLANTUNER

- Extension, pour PostgreSQL >= 8.4
- Suivant la configuration
  - Interdit l'utilisation de certains index
  - Force à zéro les statistiques d'une table vide

Cette extension est disponible à cette adresse[26] .

Voici un exemple d'utilisation :

```
LOAD 'plantuner';
CREATE TABLE test(id int);
CREATE INDEX id_idx ON test(id);
CREATE INDEX id_idx2 ON test(id);
\d test
     Table "public.test"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
Indexes:
    "id_idx" btree (id)
    "id_idx2" btree (id)

EXPLAIN SELECT id FROM test WHERE id=1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=4.34..15.03 rows=12 width=4)
   Recheck Cond: (id = 1)
   ->  Bitmap Index Scan on id_idx2  (cost=0.00..4.34 rows=12 width=0)
         Index Cond: (id = 1)
(4 rows)

SET enable_seqscan TO off;
SET plantuner.forbid_index TO 'id_idx2';
EXPLAIN SELECT id FROM test WHERE id=1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=4.34..15.03 rows=12 width=4)
   Recheck Cond: (id = 1)
   ->  Bitmap Index Scan on id_idx  (cost=0.00..4.34 rows=12 width=0)
         Index Cond: (id = 1)
(4 rows)

SET plantuner.forbid_index TO 'id_idx2,id_idx';
```

[26] http://www.sai.msu.su/~megera/wiki/plantuner

17.12

```
EXPLAIN SELECT id FROM test WHERE id=1;
                                QUERY PLAN
---------------------------------------------------------------------------
 Seq Scan on test  (cost=10000000000.00..10000000040.00 rows=12 width=4)
   Filter: (id = 1)
(2 rows)
```

Un des intérêts de cette extension est de pouvoir interdire l'utilisation d'un index, afin de pouvoir ensuite le supprimer de manière transparente, c'est-à-dire sans bloquer aucune requête applicative.

3.11 CONCLUSION

- Planificateur très avancé
- Mais faillible
- Cependant
  - ne pensez pas être plus intelligent que le planificateur

Certains SGBD concurrents supportent les *hints*, qui permettent au DBA de forcer l'optimiseur à choisir des plans d'exécution qu'il avait jugé trop coûteux. Ces *hints* sont exprimés sous la forme de commentaires et ne seront donc pas pris en compte par PostgreSQL, qui ne gère pas ces hints.

L'avis de la communauté PostgreSQL (voir https://wiki.postgresql.org/wiki/OptimizerHintsDiscussion) est que les hints mènent à des problèmes de maintenabilité du code applicatif, interfèrent avec les mises à jour, risquent d'être contre-productifs au fur et à mesure que vos tables grossissent, et sont généralement inutiles. Sur le long terme il vaut mieux rapporter un problème rencontré avec l'optimiseur pour qu'il soit définitivement corrigé.

Si le plan d'exécution généré n'est pas optimal, il est préférable de chercher à comprendre d'où vient l'erreur. Nous avons vu dans ce module quelles pouvaient être les causes entrainant des erreurs d'estimation :

- Mauvaise écriture de requête
- Modèle de données pas optimal
- Statistiques pas à jour
- Colonnes corrélées
- ...

3.11.1 QUESTIONS

N'hésitez pas, c'est le moment !

3.12 ANNEXE : NŒUDS D'UN PLAN

- Quatre types de nœuds
  - Parcours (de table, d'index, de TID, etc.)
  - Jointures (Nested Loop, Sort/Merge Join, Hash Join)
  - Opérateurs sur des ensembles (Append, Except, Intersect, etc.)
  - Et quelques autres (Sort, Aggregate, Unique, Limit, Materialize)

Un plan d'exécution est un arbre. Chaque nœud de l'arbre est une opération à effectuer par l'exécuteur. Le planificateur arrange les nœuds pour que le résultat final soit le bon, et qu'il soit récupéré le plus rapidement possible.

Il y a quatre types de nœuds :

- les parcours, qui permettent de lire les données dans les tables en passant :
- soit par la table ;
- soit par l'index ;
- les jointures, qui permettent de joindre deux ensembles de données
- les opérateurs sur des ensembles, qui là aussi vont joindre deux ensembles ou plus
- et les opérations sur un seul ensemble : tri, limite, agrégat, etc.

Cette partie va permettre d'expliquer chaque type de nœuds, ses avantages et inconvénients.

3.12.1 PARCOURS

- Ne prend rien en entrée
- Mais renvoie un ensemble de données
  - Trié ou non, filtré ou non
- Exemples typiques
  - Parcours séquentiel d'une table, avec ou sans filtrage des enregistrements produits
  - Parcours par un index, avec ou sans filtrage supplémentaire

17.12

Les parcours sont les seules opérations qui lisent les données des tables (standards, temporaires ou non journalisées). Elles ne prennent donc rien en entrée et fournissent un ensemble de données en sortie. Cet ensemble peut être trié ou non, filtré ou non.

Il existe trois types de parcours que nous allons détailler :

- le parcours de table ;
- le parcours d'index ;
- le parcours de bitmap index, tous les trois pouvant recevoir des filtres supplémentaires en sortie.

Nous verrons aussi que PostgreSQL propose d'autres types de parcours.

3.12.2 PARCOURS DE TABLE

- Parcours séquentiel de la table (Sequential Scan, ou SeqScan)
- Aussi appelé FULL TABLE SCAN par d'autres SGBD
- La table est lue entièrement
  - Même si seulement quelques lignes satisfont la requête
  - Sauf dans le cas de la clause LIMIT sans ORDER BY
- Elle est lue séquentiellement par bloc de 8 Ko
- Optimisation synchronize_seqscans

Le parcours le plus simple est le parcours séquentiel. La table est lue complètement, de façon séquentielle, par bloc de 8 Ko. Les données sont lues dans l'ordre physique sur disque, donc les données ne sont pas envoyées triées au nœud supérieur.

Cela fonctionne dans tous les cas, car il n'y a besoin de rien de plus pour le faire (un parcours d'index nécessite un index, un parcours de table ne nécessite rien de plus que la table).

Le parcours de table est intéressant pour les performances dans deux cas :

- les très petites tables ;
- les grosses tables où la majorité des lignes doit être renvoyée.

Lors de son calcul de coût, le planificateur ajoute la valeur du paramètre `seq_page_cost` à chaque bloc lu.

Une optimisation des parcours séquentiels a eu lieu en version 8.3. Auparavant, quand deux processus parcouraient en même temps la même table de façon séquentielle, ils lisaient chacun la table. À partir de la 8.3, si le paramètre `synchronize_seqscans` est activé, le processus qui entame une lecture séquentielle cherche en premier lieu si un autre

processus ne ferait pas une lecture séquentielle de la même table. Si c'est le cas, Le second processus démarre son scan de table à l'endroit où le premier processus est en train de lire, ce qui lui permet de profiter des données mises en cache par ce processus. L'accès au disque étant bien plus lent que l'accès mémoire, les processus restent naturellement synchronisés pour le reste du parcours de la table, et les lectures ne sont donc réalisées qu'une seule fois. Le début de la table restera à être lu indépendamment. Cette optimisation permet de diminuer le nombre de blocs lus par chaque processus en cas de lectures parallèles de la même table.

Il est possible, pour des raisons de tests, ou pour maintenir la compatibilité avec du code partant de l'hypothèse (erronée) que les données d'une table sont toujours retournées dans le même ordre, de désactiver ce type de parcours en positionnant le paramètre `synchronize_seqscans` à `off`.

3.12.3 PARCOURS D'INDEX

- Parcours aléatoire de l'index
- Pour chaque enregistrement correspondant à la recherche
  - Parcours non séquentiel de la table (pour vérifier la visibilité de la ligne)
- Sur d'autres SGBD, cela revient à un
  - INDEX RANGE SCAN, suivi d'un TABLE ACCESS BY INDEX ROWID
- Gros gain en performance quand le filtre est très sélectif
- L'ensemble de lignes renvoyé est trié

Parcourir une table prend du temps, surtout quand on cherche à ne récupérer que quelques lignes de cette table. Le but d'un index est donc d'utiliser une structure de données optimisée pour satisfaire une recherche particulière (on parle de prédicat).

Cette structure est un arbre. La recherche consiste à suivre la structure de l'arbre pour trouver le premier enregistrement correspondant au prédicat, puis suivre les feuilles de l'arbre jusqu'au dernier enregistrement vérifiant le prédicat. De ce fait, et étant donné la façon dont l'arbre est stocké sur disque, cela peut provoquer des déplacements de la tête de lecture.

L'autre problème des performances sur les index (mais cette fois, spécifique à PostgreSQL) est que les informations de visibilité des lignes sont uniquement stockées dans la table. Cela veut dire que, pour chaque élément de l'index correspondant au filtre, il va falloir lire la ligne dans la table pour vérifier si cette dernière est visible pour la transaction en cours. Il est de toute façons, pour la plupart des requêtes, nécessaire d'aller inspecter l'enregistrement de la table pour récupérer les autres colonnes nécessaires au

bon déroulement de la requête, qui ne sont la plupart du temps pas stockées dans l'index. Ces enregistrements sont habituellement éparpillés dans la table, et retournés dans un ordre totalement différent de leur ordre physique par le parcours sur l'index. Cet accès à la table génère donc énormément d'accès aléatoires. Or, ce type d'activité est généralement le plus lent sur un disque magnétique. C'est pourquoi le parcours d'une large portion d'un index est très lent. PostgreSQL ne cherchera à utiliser un index que s'il suppose qu'il aura peu de lignes à récupérer.

Voici l'algorithme permettant un parcours d'index avec PostgreSQL :

- Pour tous les éléments de l'index
- Chercher l'élément souhaité dans l'index
- Lorsqu'un élément est trouvé
- Vérifier qu'il est visible par la transaction en lisant la ligne dans la table et récupérer les colonnes supplémentaires de la table

Cette manière de procéder est identique à ce que proposent d'autres SGBD sous les termes d'« INDEX RANGE SCAN », suivi d'un « TABLE ACCESS BY INDEX ROWID ».

Un parcours d'index est donc très coûteux, principalement à cause des déplacements de la tête de lecture. Le paramètre lié au coût de lecture aléatoire d'une page est par défaut quatre fois supérieur à celui de la lecture séquentielle d'une page. Ce paramètre s'appelle `random_page_cost`. Un parcours d'index n'est préférable à un parcours de table que si la recherche ne va ramener qu'un très faible pourcentage de la table. Et dans ce cas, le gain possible est très important par rapport à un parcours séquentiel de table. Par contre, il se révèle très lent pour lire un gros pourcentage de la table (les accès aléatoires diminuent spectaculairement les performances).

Il est à noter que, contrairement au parcours de table, le parcours d'index renvoie les données triées. C'est le seul parcours à le faire. Il peut même servir à honorer la clause `ORDER BY` d'une requête. L'index est aussi utilisable dans le cas des tris descendants. Dans ce cas, le nœud est nommé « Index Scan Backward ». Ce renvoi de données triées est très intéressant lorsqu'il est utilisé en conjonction avec la clause `LIMIT`.

Il ne faut pas oublier aussi le coût de mise à jour de l'index. Si un index n'est pas utilisé, il coûte cher en maintenance (ajout des nouvelles entrées, suppression des entrées obsolètes, etc).

Enfin, il est à noter que ce type de parcours est consommateur aussi en CPU.

Voici un exemple montrant les deux types de parcours et ce que cela occasionne comme lecture disque :

Commençons par créer une table, lui insérer quelques données et lui ajouter un index :

```
b1=# CREATE TABLE t1 (id integer);
CREATE TABLE
b1=# INSERT INTO t1 (id) VALUES (1), (2), (3);
INSERT 0 3
b1=# CREATE INDEX i1 ON t1(id);
CREATE INDEX
```

Réinitialisons les statistiques d'activité :

```
b1=# SELECT pg_stat_reset();
 pg_stat_reset
---------------


(1 row)
```

Essayons maintenant de lire la table avec un simple parcours séquentiel :

```
b1=# EXPLAIN ANALYZE SELECT * FROM t1 WHERE id=2;
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..1.04 rows=1 width=4)
                 (actual time=0.011..0.012 rows=1 loops=1)
   Filter: (id = 2)
 Total runtime: 0.042 ms
(3 rows)
```

`Seq Scan` est le titre du nœud pour un parcours séquentiel. Profitons-en pour noter qu'il a fait de lui-même un parcours séquentiel. En effet, la table est tellement petite (8 Ko) qu'utiliser l'index couterait forcément plus cher. Maintenant regardons les statistiques sur les blocs lus :

```
b1=# SELECT relname, heap_blks_read, heap_blks_hit,
 idx_blks_read, idx_blks_hit
 FROM pg_statio_user_tables
 WHERE relname='t1';

 relname | heap_blks_read | heap_blks_hit | idx_blks_read | idx_blks_hit
---------+----------------+---------------+---------------+--------------
 t1      |              0 |             1 |             0 |            0
(1 row)
```

Seul un bloc a été lu, et il a été lu dans la table (colonne `heap_blks_hit` à 1).

Pour faire un parcours d'index, nous allons désactiver les parcours séquentiels.

```
b1=# SET enable_seqscan TO off;
SET
```

Il existe aussi un paramètre, appelé `enable_indexscan`, pour désactiver les parcours d'index.

Nous allons de nouveau réinitialiser les statistiques :

```
b1=# SELECT pg_stat_reset();
 pg_stat_reset
---------------


(1 row)
```

Maintenant relançons la requête :

```
b1=# EXPLAIN ANALYZE SELECT * FROM t1 WHERE id=2;
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using i1 on t1  (cost=0.00..8.27 rows=1 width=4)
                            (actual time=0.088..0.090 rows=1 loops=1)
   Index Cond: (id = 2)
 Total runtime: 0.121 ms
(3 rows)
```

Nous avons bien un parcours d'index. Vérifions les statistiques sur l'activité :

```
b1=# SELECT relname, heap_blks_read, heap_blks_hit,
 idx_blks_read, idx_blks_hit
 FROM pg_statio_user_tables
 WHERE relname='t1';
 relname | heap_blks_read | heap_blks_hit | idx_blks_read | idx_blks_hit
---------+----------------+---------------+---------------+--------------
 t1      |              0 |             1 |             0 |            1
(1 row)
```

Une page disque a été lue dans l'index (colonne `idx_blks_hit` à 1) et une autre a été lue dans la table (colonne `heap_blks_hit` à 1). Le plus impactant est l'accès aléatoire sur l'index et la table. Il serait bon d'avoir une lecture de l'index, puis une lecture séquentielle de la table. C'est le but du Bitmap Index Scan.

3.12.4 PARCOURS D'INDEX BITMAP

- En VO, Bitmap Index Scan / Bitmap Heap Scan
- Disponible à partir de la 8.1
- Diminuer les déplacements de la tête de lecture en découplant le parcours de l'index du parcours de la table
  - Lecture en un bloc de l'index
  - Lecture en un bloc de la partie intéressante de la table
- Autre intérêt : pouvoir combiner plusieurs index en mémoire
  - Nœud BitmapAnd

  - Nœud BitmapOr
- Coût de démarrage généralement important
  - Parcours moins intéressant avec une clause LIMIT

Contrairement à d'autres SGBD, un index bitmap n'a aucune existence sur disque. Il est créé en mémoire lorsque son utilisation a un intérêt. Le but est de diminuer les déplacements de la tête de lecture en découplant le parcours de l'index du parcours de la table :

- Lecture en un bloc de l'index ;
- Lecture en un bloc de la partie intéressante de la table (dans l'ordre physique de la table, pas dans l'ordre logique de l'index).

Il est souvent utilisé quand il y a un grand nombre de valeurs à filtrer, notamment pour les clauses `IN` et `ANY`. En voici un exemple :

```
b1=# CREATE TABLE t1(c1 integer, c2 integer);
CREATE TABLE
b1=# INSERT INTO t1 SELECT i, i+1 FROM generate_series(1, 1000) AS i;
INSERT 0 1000
b1=# CREATE INDEX ON t1(c1);
CREATE INDEX
b1=# CREATE INDEX ON t1(c2);
CREATE INDEX
b1=# EXPLAIN SELECT * FROM t1 WHERE c1 IN (10, 40, 60, 100, 600);
                                QUERY PLAN
--------------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=17.45..22.85 rows=25 width=8)
   Recheck Cond: (c1 = ANY ('{10,40,60,100,600}'::integer[]))
   ->  Bitmap Index Scan on t1_c1_idx  (cost=0.00..17.44 rows=25 width=0)
         Index Cond: (c1 = ANY ('{10,40,60,100,600}'::integer[]))
(4 rows)
```

La partie `Bitmap Index Scan` concerne le parcours de l'index, et la partie `Bitmap Heap Scan` concerne le parcours de table.

L'algorithme pourrait être décrit ainsi :

- Chercher tous les éléments souhaités dans l'index
- Les placer dans une structure (de TID) de type bitmap en mémoire
- Faire un parcours séquentiel partiel de la table

Ce champ de bits a deux codages possibles :

- 1 bit par ligne
- Ou 1 bit par bloc si trop de données.

Dans ce dernier (mauvais) cas, il y a une étape de revérification (`Recheck Condition`).

Ce type d'index présente un autre gros intérêt : pouvoir combiner plusieurs index en mémoire. Les bitmaps de TID se combinent facilement avec des opérations booléennes AND et OR. Dans ce cas, on obtient les nœuds BitmapAnd et Nœud BitmapOr. Voici un exemple de ce dernier :

```
b1=# EXPLAIN SELECT * FROM t1
 WHERE c1 IN (10, 40, 60, 100, 600) OR c2 IN (300, 400, 500);
                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=30.32..36.12 rows=39 width=8)
   Recheck Cond: ((c1 = ANY ('{10,40,60,100,600}'::integer[]))
                 OR (c2 = ANY ('{300,400,500}'::integer[])))
   ->  BitmapOr  (cost=30.32..30.32 rows=40 width=0)
         ->  Bitmap Index Scan on t1_c1_idx
               (cost=0.00..17.44 rows=25 width=0)
               Index Cond: (c1 = ANY ('{10,40,60,100,600}'::integer[]))
         ->  Bitmap Index Scan on t1_c2_idx
               (cost=0.00..12.86 rows=15 width=0)
               Index Cond: (c2 = ANY ('{300,400,500}'::integer[]))
(7 rows)
```

Le coût de démarrage est généralement important à cause de la lecture préalable de l'index et du tri des TID. Du coup, ce type de parcours est moins intéressant si une clause LIMIT est présente. Un parcours d'index simple sera généralement choisi dans ce cas.

Le paramètre enable_bitmapscan permet d'activer ou de désactiver l'utilisation des parcours d'index bitmap.

À noter que ce type de parcours n'est disponible qu'à partir de PostgreSQL 8.1.

3.12.5 PARCOURS D'INDEX SEUL

- Avant la 9.2, pour une requête de ce type
  - SELECT c1 FROM t1 WHERE c1<10
- PostgreSQL devait lire l'index et la table
  - car les informations de visibilité ne se trouvent que dans la table
- En 9.2, le planificateur peut utiliser la « Visibility Map »
  - nouveau nœud « Index Only Scan »
  - Index B-Tree (9.2+)
  - Index SP-GiST (9.2+)
  - Index GiST (9.5+) => Types : point, box, inet, range

Voici un exemple en 9.1 :

```
b1=# CREATE TABLE demo_i_o_scan (a int, b text);
CREATE TABLE
b1=# INSERT INTO demo_i_o_scan
b1-# SELECT random()*10000000, a
b1-# FROM generate_series(1,10000000) a;
INSERT 0 10000000
b1=# CREATE INDEX demo_idx ON demo_i_o_scan (a,b);
CREATE INDEX
b1=# VACUUM ANALYZE demo_i_o_scan ;
VACUUM
b1=# EXPLAIN ANALYZE SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2299.83..59688.65 rows=89565 width=11)
                             (actual time=209.569..3314.717 rows=89877 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2277.44 rows=89565 width=0)
                            (actual time=197.177..197.177 rows=89877 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
 Total runtime: 3323.497 ms
(5 rows)

b1=# EXPLAIN ANALYZE SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2299.83..59688.65 rows=89565 width=11)
                              (actual time=48.620..269.907 rows=89877 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2277.44 rows=89565 width=0)
                             (actual time=35.780..35.780 rows=89877 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
 Total runtime: 273.761 ms
(5 rows)
```

Donc 3 secondes pour la première exécution (avec un cache pas forcément vide), et 273 millisecondes pour la deuxième exécution (et les suivantes, non affichées ici).

Voici ce que cet exemple donne en 9.2 :

```
b1=# CREATE TABLE demo_i_o_scan (a int, b text);
CREATE TABLE
b1=# INSERT INTO demo_i_o_scan
b1-# SELECT random()*10000000, a
b1-# FROM (select generate_series(1,10000000)) AS t(a);
INSERT 0 10000000
```

```
b1=# CREATE INDEX demo_idx ON demo_i_o_scan (a,b);
CREATE INDEX
b1=# VACUUM ANALYZE demo_i_o_scan ;
VACUUM
b1=# EXPLAIN ANALYZE SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using demo_idx on demo_i_o_scan
                              (cost=0.00..3084.77 rows=86656 width=11)
                              (actual time=0.080..97.942 rows=89432 loops=1)
   Index Cond: ((a >= 10000) AND (a <= 100000))
   Heap Fetches: 0
 Total runtime: 108.134 ms
(4 rows)

b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using demo_idx on demo_i_o_scan
                              (cost=0.00..3084.77 rows=86656 width=11)
                              (actual time=0.024..26.954 rows=89432 loops=1)
   Index Cond: ((a >= 10000) AND (a <= 100000))
   Heap Fetches: 0
   Buffers: shared hit=347
 Total runtime: 34.352 ms
(5 rows)
```

Donc, même à froid, il est déjà pratiquement trois fois plus rapide que la version 9.1, à chaud. La version 9.2 est dix fois plus rapide à chaud.

Essayons maintenant en désactivant les parcours d'index seul :

```
b1=# SET enable_indexonlyscan TO off;
SET
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2239.88..59818.53 rows=86656 width=11)
                             (actual time=29.256..2992.289 rows=89432 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   Rows Removed by Index Recheck: 6053582
   Buffers: shared hit=346 read=43834 written=2022
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2218.21 rows=86656 width=0)
                              (actual time=27.004..27.004 rows=89432 loops=1)
```

```
        Index Cond: ((a >= 10000) AND (a <= 100000))
        Buffers: shared hit=346
 Total runtime: 3000.502 ms
(8 rows)

b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2239.88..59818.53 rows=86656 width=11)
                             (actual time=23.533..1141.754 rows=89432 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   Rows Removed by Index Recheck: 6053582
   Buffers: shared hit=2 read=44178
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2218.21 rows=86656 width=0)
                            (actual time=21.592..21.592 rows=89432 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
         Buffers: shared hit=2 read=344
 Total runtime: 1146.538 ms
(8 rows)
```

On retombe sur les performances de la version 9.1.

Maintenant, essayons avec un cache vide (niveau PostgreSQL et système) :

- en 9.1

```
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                       QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2299.83..59688.65 rows=89565 width=11)
                            (actual time=126.624..9750.245 rows=89877 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   Buffers: shared hit=2 read=44250
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2277.44 rows=89565 width=0)
                           (actual time=112.542..112.542 rows=89877 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
         Buffers: shared hit=2 read=346
 Total runtime: 9765.670 ms
(7 rows)
```

- en 9.2

```
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                       QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using demo_idx on demo_i_o_scan
```

```
                                (cost=0.00..3084.77 rows=86656 width=11)
                                (actual time=11.592..63.379 rows=89432 loops=1)
   Index Cond: ((a >= 10000) AND (a <= 100000))
   Heap Fetches: 0
   Buffers: shared hit=2 read=345
 Total runtime: 70.188 ms
(5 rows)
```

La version 9.1 met 10 secondes à exécuter la requête, alors que la version 9.2 ne met que 70 millisecondes (elle est donc 142 fois plus rapide).

Voir aussi cet article de blog[27] .

3.12.6 PARCOURS : AUTRES

- TID Scan
- Function Scan
- Values
- Result

Il existe d'autres parcours, bien moins fréquents ceci dit.

`TID` est l'acronyme de `Tuple ID`. C'est en quelque sorte un pointeur vers une ligne. Un `TID Scan` est un parcours de `TID`. Ce type de parcours est généralement utilisé en interne par PostgreSQL. Notez qu'il est possible de le désactiver via le paramètre `enable_tidscan`.

Un `Function Scan` est utilisé par les fonctions renvoyant des ensembles (appelées `SRF` pour `Set Returning Functions`). En voici un exemple :

```
b1=# EXPLAIN SELECT * FROM generate_series(1, 1000);
                              QUERY PLAN
-----------------------------------------------------------------------
 Function Scan on generate_series  (cost=0.00..10.00 rows=1000 width=4)
(1 row)
```

`VALUES` est une clause de l'instruction `INSERT`, mais `VALUES` peut aussi être utilisé comme une table dont on spécifie les valeurs. Par exemple :

```
b1=# VALUES (1), (2);
 column1
---------
       1
       2
```

[27] http://pgsnaga.blogspot.com/2011/10/index-only-scans-and-heap-block-reads.html

```
(2 rows)

b1=# SELECT * FROM (VALUES ('a', 1), ('b', 2), ('c', 3)) AS tmp(c1, c2);
 c1 | c2
----+----
 a  |  1
 b  |  2
 c  |  3
(3 rows)
```

Le planificateur utilise un nœud spécial appelé `Values Scan` pour indiquer un parcours sur cette clause :

```
b1=# EXPLAIN
b1-# SELECT *
b1-# FROM (VALUES ('a', 1), ('b', 2), ('c', 3))
b1-# AS tmp(c1, c2);
                          QUERY PLAN
--------------------------------------------------------------
 Values Scan on "*VALUES*"  (cost=0.00..0.04 rows=3 width=36)
(1 row)
```

Enfin, le nœud `Result` n'est pas à proprement parler un nœud de type parcours. Il y ressemble dans le fait qu'il ne prend aucun ensemble de données en entrée et en renvoie un en sortie. Son but est de renvoyer un ensemble de données suite à un calcul. Par exemple :

```
b1=# EXPLAIN SELECT 1+2;
                QUERY PLAN
-----------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=0)
(1 row)
```

3.12.7 JOINTURES

- Prend deux ensembles de données en entrée
  - L'une est appelée inner (interne)
  - L'autre est appelée outer (externe)
- Et renvoie un seul ensemble de données
- Exemples typiques
  - Nested Loop, Merge Join, Hash Join

Le but d'une jointure est de grouper deux ensembles de données pour n'en produire qu'un seul. L'un des ensembles est appelé ensemble interne (`inner set`), l'autre est appelé

ensemble externe (`outer set`).

Le planificateur de PostgreSQL est capable de traiter les jointures grâce à trois nœuds :

- `Nested Loop`, une boucle imbriquée ;
- `Merge Join`, un parcours des deux ensembles triés ;
- `Hash Join`, une jointure par tests des données hachées.

3.12.8 NESTED LOOP

- Pour chaque ligne de la relation externe
  - Pour chaque ligne de la relation interne
  - Si la condition de jointure est avérée
    - Émettre la ligne en résultat
- L'ensemble externe n'est parcouru qu'une fois
- L'ensemble interne est parcouru pour chaque ligne de l'ensemble externe
  - Avoir un index utilisable sur l'ensemble interne augmente fortement les performances

Étant donné le pseudo-code indiqué ci-dessus, on s'aperçoit que l'ensemble externe n'est parcouru qu'une fois alors que l'ensemble interne est parcouru pour chaque ligne de l'ensemble externe. Le coût de ce nœud est donc proportionnel à la taille des ensembles. Il est intéressant pour les petits ensembles de données, et encore plus lorsque l'ensemble interne dispose d'un index satisfaisant la condition de jointure.

En théorie, il s'agit du type de jointure le plus lent, mais il a un gros intérêt. Il n'est pas nécessaire de trier les données ou de les hacher avant de commencer à traiter les données. Il a donc un coût de démarrage très faible, ce qui le rend très intéressant si cette jointure est couplée à une clause `LIMIT`, ou si le nombre d'itérations (donc le nombre d'enregistrements de la relation externe) est faible.

Il est aussi très intéressant, car il s'agit du seul nœud capable de traiter des jointures sur des conditions différentes de l'égalité ainsi que des jointures de type `CROSS JOIN`.

Voici un exemple avec deux parcours séquentiels :

```
b1=# EXPLAIN SELECT *
    FROM pg_class, pg_namespace
 WHERE pg_class.relnamespace=pg_namespace.oid;
                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop  (cost=0.00..37.18 rows=281 width=307)
   Join Filter: (pg_class.relnamespace = pg_namespace.oid)
```

```
   ->  Seq Scan on pg_class  (cost=0.00..10.81 rows=281 width=194)
   ->  Materialize  (cost=0.00..1.09 rows=6 width=117)
         ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=117)
(5 rows)
```

Et un exemple avec un parcours séquentiel et un parcours d'index :

```
b1=# SET random_page_cost TO 0.5;
b1=# EXPLAIN SELECT *
    FROM pg_class, pg_namespace
 WHERE pg_class.relnamespace=pg_namespace.oid;
                                QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=0.00..33.90 rows=281 width=307)
   ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=117)
   ->  Index Scan using pg_class_relname_nsp_index on pg_class
       (cost=0.00..4.30 rows=94 width=194)
         Index Cond: (relnamespace = pg_namespace.oid)
(4 rows)
```

Le paramètre `enable_nestloop` permet d'activer ou de désactiver ce type de nœud.

3.12.9 MERGE JOIN

- Trier l'ensemble interne
- Trier l'ensemble externe
- Tant qu'il reste des lignes dans un des ensembles
  - Lire les deux ensembles en parallèle
  - Lorsque la condition de jointure est avérée
  - Émettre la ligne en résultat
- Parcourir les deux ensembles triés (d'où Sort-Merge Join)
- Ne gère que les conditions avec égalité
- Produit un ensemble résultat trié
- Le plus rapide sur de gros ensembles de données

Contrairement au `Nested Loop`, le `Merge Join` ne lit qu'une fois chaque ligne, sauf pour les valeurs dupliquées. C'est d'ailleurs son principal atout.

L'algorithme est assez simple. Les deux ensembles de données sont tout d'abord triés, puis ils sont parcourus ensemble. Lorsque la condition de jointure est vraie, la ligne résultante est envoyée dans l'ensemble de données en sortie.

L'inconvénient de cette méthode est que les données en entrée doivent être triées. Trier les données peut prendre du temps, surtout si les ensembles de données sont volumineux.

Cela étant dit, le `Merge Join` peut s'appuyer sur un index pour accélérer l'opération de tri (ce sera alors forcément un `Index Scan`). Une table clusterisée peut aussi accélérer l'opération de tri. Néanmoins, il faut s'attendre à avoir un coût de démarrage important pour ce type de nœud, ce qui fait qu'il sera facilement disqualifié si une clause LIMIT est à exécuter après la jointure.

Le gros avantage du tri sur les données en entrée est que les données reviennent triées. Cela peut avoir son avantage dans certains cas.

Voici un exemple pour ce nœud :

```
b1=# EXPLAIN SELECT *
    FROM pg_class, pg_namespace
 WHERE pg_class.relnamespace=pg_namespace.oid;
                                QUERY PLAN
------------------------------------------------------------------------
 Merge Join  (cost=23.38..27.62 rows=281 width=307)
   Merge Cond: (pg_namespace.oid = pg_class.relnamespace)
   ->  Sort  (cost=1.14..1.15 rows=6 width=117)
         Sort Key: pg_namespace.oid
         ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=117)
   ->  Sort  (cost=22.24..22.94 rows=281 width=194)
         Sort Key: pg_class.relnamespace
         ->  Seq Scan on pg_class  (cost=0.00..10.81 rows=281 width=194)
(8 rows)
```

Le paramètre `enable_mergejoin` permet d' activer ou de désactiver ce type de nœud.

3.12.10 HASH JOIN

- Calculer le hachage de chaque ligne de l'ensemble interne
- Tant qu'il reste des lignes dans l'ensemble externe
  - Hacher la ligne lue
  - Comparer ce hachage aux lignes hachées de l'ensemble interne
  - Si une correspondance est trouvée
  - Émettre la ligne trouvée en résultat
- Ne gère que les conditions avec égalité
- Idéal pour joindre une grande table à une petite table
- Coût de démarrage important à cause du hachage de la table

La vérification de la condition de jointure peut se révéler assez lente dans beaucoup de cas : elle nécessite un accès à un enregistrement par un index ou un parcours de la table

interne à chaque itération dans un Nested Loop par exemple. Le Hash Join cherche à supprimer ce problème en créant une table de hachage de la table interne. Cela sous-entend qu'il faut au préalable calculer le hachage de chaque ligne de la table interne. Ensuite, il suffit de parcourir la table externe, hacher chaque ligne l'une après l'autre et retrouver le ou les enregistrements de la table interne pouvant correspondre à la valeur hachée de la table externe. On vérifie alors qu'ils répondent bien aux critères de jointure (il peut y avoir des collisions dans un hachage, ou des prédicats supplémentaires à vérifier).

Ce type de nœud est très rapide à condition d'avoir suffisamment de mémoire pour stocker le résultat du hachage de l'ensemble interne. Du coup, le paramétrage de work_mem peut avoir un gros impact. De même, diminuer le nombre de colonnes récupérées permet de diminuer la mémoire à utiliser pour le hachage et du coup d'améliorer les performances d'un Hash Join. Cependant, si la mémoire est insuffisante, il est possible de travailler par groupes de lignes (batch). L'algorithme est alors une version améliorée de l'algorithme décrit plus haut, permettant justement de travailler en partitionnant la table interne (on parle de Hybrid Hash Join). Il est à noter que ce type de nœud est souvent idéal pour joindre une grande table à une petite table.

Le coût de démarrage peut se révéler important à cause du hachage de la table interne. Il ne sera probablement pas utilisé par l'optimiseur si une clause LIMIT est à exécuter après la jointure.

Attention, les données retournées par ce nœud ne sont pas triées.

De plus, ce type de nœud peut être très lent si l'estimation de la taille des tables est mauvaise.

Voici un exemple de Hash Join :

```
b1=# EXPLAIN SELECT *
    FROM pg_class, pg_namespace
 WHERE pg_class.relnamespace=pg_namespace.oid;
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=1.14..15.81 rows=281 width=307)
   Hash Cond: (pg_class.relnamespace = pg_namespace.oid)
   ->  Seq Scan on pg_class  (cost=0.00..10.81 rows=281 width=194)
   ->  Hash  (cost=1.06..1.06 rows=6 width=117)
         ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=117)
(5 rows)
```

Le paramètre enable_hashjoin permet d' activer ou de désactiver ce type de nœud.

3.12.11 SUPPRESSION D'UNE JOINTURE

```
SELECT pg_class.relname, pg_class.reltuples
FROM pg_class
LEFT JOIN pg_namespace
       ON pg_class.relnamespace=pg_namespace.oid;
```

- Un index unique existe sur la colonne oid de pg_namespace
- Jointure inutile
  - sa présence ne change pas le résultat
- PostgreSQL peut supprimer la jointure à partir de la 9.0

Sur la requête ci-dessus, la jointure est inutile. En effet, il existe un index unique sur la colonne `oid` de la table `pg_namespace`. De plus, aucune colonne de la table `pg_namespace` ne va apparaître dans le résultat. Autrement dit, que la jointure soit présente ou non, cela ne va pas changer le résultat. Dans ce cas, il est préférable de supprimer la jointure. Si le développeur ne le fait pas, PostgreSQL le fera (pour les versions 9.0 et ultérieures de PostgreSQL). Cet exemple le montre.

Voici la requête exécutée en 8.4 :

```
b1=# EXPLAIN SELECT pg_class.relname, pg_class.reltuples
    FROM pg_class
 LEFT JOIN pg_namespace ON pg_class.relnamespace=pg_namespace.oid;
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Left Join  (cost=1.14..12.93 rows=244 width=68)
   Hash Cond: (pg_class.relnamespace = pg_namespace.oid)
   ->  Seq Scan on pg_class  (cost=0.00..8.44 rows=244 width=72)
   ->  Hash  (cost=1.06..1.06 rows=6 width=4)
         ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=4)
(5 rows)
```

Et la même requête exécutée en 9.0 :

```
b1=# EXPLAIN SELECT pg_class.relname, pg_class.reltuples
    FROM pg_class
 LEFT JOIN pg_namespace ON pg_class.relnamespace=pg_namespace.oid;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on pg_class  (cost=0.00..10.81 rows=281 width=72)
(1 row)
```

On constate que la jointure est ignorée.

Ce genre de requête peut fréquemment survenir surtout avec des générateurs de requêtes comme les ORM. L'utilisation de vues imbriquées peut aussi être la source de ce type de problème.

3.12.12 ORDRE DE JOINTURE

- Trouver le bon ordre de jointure est un point clé dans la recherche de performances
- Nombre de possibilités en augmentation factorielle avec le nombre de tables
- Si petit nombre, recherche exhaustive
- Sinon, utilisation d'heuristiques et de GEQO
  - Limite le temps de planification et l'utilisation de mémoire
  - GEQO remplacé par Simulated Annealing ? (recuit simulé en VF)

Sur une requête comme `SELECT * FROM a, b, c...`, les tables a, b et c ne sont pas forcément jointes dans cet ordre. PostgreSQL teste différents ordres pour obtenir les meilleures performances.

Prenons comme exemple la requête suivante :

```
SELECT * FROM a JOIN b ON a.id=b.id JOIN c ON b.id=c.id;
```

Avec une table a contenant un million de lignes, une table b n'en contenant que 1000 et une table c en contenant seulement 10, et une configuration par défaut, son plan d'exécution est celui-ci :

```
b1=# EXPLAIN SELECT * FROM a JOIN b ON a.id=b.id JOIN c ON b.id=c.id;
                                QUERY PLAN
--------------------------------------------------------------------------
 Nested Loop  (cost=1.23..18341.35 rows=1 width=12)
   Join Filter: (a.id = b.id)
   ->  Seq Scan on b  (cost=0.00..15.00 rows=1000 width=4)
   ->  Materialize  (cost=1.23..18176.37 rows=10 width=8)
         ->  Hash Join  (cost=1.23..18176.32 rows=10 width=8)
               Hash Cond: (a.id = c.id)
               ->  Seq Scan on a  (cost=0.00..14425.00 rows=1000000 width=4)
               ->  Hash  (cost=1.10..1.10 rows=10 width=4)
                     ->  Seq Scan on c  (cost=0.00..1.10 rows=10 width=4)
(9 rows)
```

Le planificateur préfère joindre tout d' abord la table a à la table c, puis son résultat à la table b. Cela lui permet d'avoir un ensemble de données en sortie plus petit (donc moins de consommation mémoire) avant de faire la jointure avec la table b.

Cependant, si PostgreSQL se trouve face à une jointure de 25 tables, le temps de calculer tous les plans possibles en prenant en compte l'ordre des jointures sera très important. En fait, plus le nombre de tables jointes est important, et plus le temps de planification va augmenter. Il est nécessaire de prévoir une échappatoire à ce système. En fait, il en existe

plusieurs. Les paramètres `from_collapse_limit` et `join_collapse_limit` permettent de spécifier une limite en nombre de tables. Si cette limite est dépassée, PostgreSQL ne cherchera plus à traiter tous les cas possibles de réordonnancement des jointures. Par défaut, ces deux paramètres valent 8, ce qui fait que, dans notre exemple, le planificateur a bien cherché à changer l'ordre des jointures. En configurant ces paramètres à une valeur plus basse, le plan va changer :

```
b1=# SET join_collapse_limit TO 2;
SET
b1=# EXPLAIN SELECT * FROM a JOIN b ON a.id=b.id JOIN c ON b.id=c.id;
                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop  (cost=27.50..18363.62 rows=1 width=12)
   Join Filter: (a.id = c.id)
   ->  Hash Join  (cost=27.50..18212.50 rows=1000 width=8)
         Hash Cond: (a.id = b.id)
         ->  Seq Scan on a  (cost=0.00..14425.00 rows=1000000 width=4)
         ->  Hash  (cost=15.00..15.00 rows=1000 width=4)
               ->  Seq Scan on b  (cost=0.00..15.00 rows=1000 width=4)
   ->  Materialize  (cost=0.00..1.15 rows=10 width=4)
         ->  Seq Scan on c  (cost=0.00..1.10 rows=10 width=4)
(9 rows)
```

Avec un `join_collapse_limit` à 2, PostgreSQL décide de ne pas tester l'ordre des jointures. Le plan fourni fonctionne tout aussi bien, mais son estimation montre qu'elle semble être moins performante (coût de 18363 au lieu de 18341 précédemment).

Une autre technique mise en place pour éviter de tester tous les plans possibles est GEQO (*GEnetic Query Optimizer*). Cette technique est très complexe, et dispose d'un grand nombre de paramètres que très peu savent réellement configurer. Comme tout algorithme génétique, il fonctionne par introduction de mutations aléatoires sur un état initial donné. Il permet de planifier rapidement une requête complexe, et de fournir un plan d'exécution acceptable.

Malgré l'introduction de ces mutations aléatoires, le moteur arrive tout de même à conserver un fonctionnement déterministe (depuis la version 9.1, voir ce commit[28] pour plus de détails). Tant que le paramètre `geqo_seed` ainsi que les autres paramètres contrôlant GEQO restent inchangés, le plan obtenu pour une requête donnée restera inchangé. Il est possible de faire varier la valeur de `geqo_seed` pour obtenir d'autres plans (voir la documentation officielle[29] pour approfondir ce point).

[28] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=f5bc74192d2ffb32952a06c62b3458d28ff7f98f
[29] https://www.postgresql.org/docs/current/static/geqo-pg-intro.html#AEN116279

3.12.13 OPÉRATIONS ENSEMBLISTES

- Prend un ou plusieurs ensembles de données en entrée
- Et renvoie un ensemble de données
- Concernent principalement les requêtes sur des tables partitionnées ou héritées
- Exemples typiques
  - Append
  - Intersect
  - Except

Ce type de nœuds prend un ou plusieurs ensembles de données en entrée et renvoie un seul ensemble de données. Cela concerne surtout les requêtes visant des tables partitionnées ou héritées.

3.12.14 APPEND

- Prend plusieurs ensembles de données
- Fournit un ensemble de données en sortie
  - Non trié
- Utilisé par les requêtes
  - Sur des tables héritées (partitionnement inclus)
  - Ayant des UNION ALL et des UNION
  - Attention que le UNION sans ALL élimine les duplicats, ce qui nécessite une opération supplémentaire de tri

Un nœud `Append` a pour but de concaténer plusieurs ensembles de données pour n'en faire qu'un, non trié. Ce type de nœud est utilisé dans les requêtes concaténant explicitement des tables (clause `UNION`) ou implicitement (requêtes sur une table mère).

Supposons que la table t1 est une table mère. Plusieurs tables héritent de cette table : t1_0, t1_1, t1_2 et t1_3. Voici ce que donne un `SELECT` sur la table mère :

```
b1=# EXPLAIN SELECT * FROM t1;
                                QUERY PLAN
-------------------------------------------------------------------------
 Result  (cost=0.00..89.20 rows=4921 width=36)
   ->  Append  (cost=0.00..89.20 rows=4921 width=36)
         ->  Seq Scan on t1  (cost=0.00..0.00 rows=1 width=36)
         ->  Seq Scan on t1_0 t1  (cost=0.00..22.30 rows=1230 width=36)
         ->  Seq Scan on t1_1 t1  (cost=0.00..22.30 rows=1230 width=36)
         ->  Seq Scan on t1_2 t1  (cost=0.00..22.30 rows=1230 width=36)
```

```
         ->  Seq Scan on t1_3 t1  (cost=0.00..22.30 rows=1230 width=36)
(7 rows)
```

Nouvel exemple avec un filtre sur la clé de partitionnement :

```
b1=# SHOW constraint_exclusion ;
 constraint_exclusion
----------------------
 off
(1 row)
b1=# EXPLAIN SELECT * FROM t1 WHERE c1>250;
                              QUERY PLAN
-----------------------------------------------------------------------
 Result  (cost=0.00..101.50 rows=1641 width=36)
   ->  Append  (cost=0.00..101.50 rows=1641 width=36)
         ->  Seq Scan on t1  (cost=0.00..0.00 rows=1 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_0 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_1 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_2 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_3 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
(12 rows)
```

Le paramètre `constraint_exclusion` permet d'éviter de parcourir les tables filles qui ne peuvent pas accueillir les données qui nous intéressent. Pour que le planificateur comprenne qu'il peut ignorer certaines tables filles, ces dernières doivent avoir des contraintes `CHECK` qui assurent le planificateur qu'elles ne peuvent pas contenir les données en question :

```
b1=# SHOW constraint_exclusion ;
 constraint_exclusion
----------------------
 on
(1 row)
b1=# EXPLAIN SELECT * FROM t1 WHERE c1>250;
                              QUERY PLAN
-----------------------------------------------------------------------
 Result  (cost=0.00..50.75 rows=821 width=36)
   ->  Append  (cost=0.00..50.75 rows=821 width=36)
         ->  Seq Scan on t1  (cost=0.00..0.00 rows=1 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_2 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
```

```
         ->  Seq Scan on t1_3 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
(8 rows)
```

Une requête utilisant `UNION ALL` passera aussi par un nœud `Append` :

```
b1=# EXPLAIN SELECT 1 UNION ALL SELECT 2;
                        QUERY PLAN
----------------------------------------------------
 Result  (cost=0.00..0.04 rows=2 width=4)
   ->  Append  (cost=0.00..0.04 rows=2 width=4)
         ->  Result  (cost=0.00..0.01 rows=1 width=0)
         ->  Result  (cost=0.00..0.01 rows=1 width=0)
(4 rows)
```

`UNION ALL` récupère toutes les lignes des deux ensembles de données, même en cas de duplicat. Pour n'avoir que les lignes distinctes, il est possible d'utiliser `UNION` sans la clause `ALL` mais cela nécessite un tri des données pour faire la distinction (un peu comme un `Merge Join`).

Attention que le UNION sans ALL élimine les duplicats, ce qui nécessite une opération supplémentaire de tri :

```
b1=# EXPLAIN SELECT 1 UNION SELECT 2;
                           QUERY PLAN
-----------------------------------------------------------------
 Unique  (cost=0.05..0.06 rows=2 width=0)
   ->  Sort  (cost=0.05..0.06 rows=2 width=0)
         Sort Key: (1)
         ->  Append  (cost=0.00..0.04 rows=2 width=0)
               ->  Result  (cost=0.00..0.01 rows=1 width=0)
               ->  Result  (cost=0.00..0.01 rows=1 width=0)
(6 rows)
```

3.12.15 MERGEAPPEND

- Append avec optimisation
- Fournit un ensemble de données en sortie trié
- Utilisé par les requêtes
  - UNION ALL ou partitionnement/héritage
  - Utilisant des parcours triés
  - Idéal avec Limit

Le nœud MergeAppend est une optimisation spécifiquement conçue pour le partitionnement, introduite en 9.1.

Cela permet de répondre plus efficacement aux requêtes effectuant un tri sur un `UNION ALL`, soit explicite, soit induit par un héritage/partitionnement. Considérons la requête suivante :

```
SELECT *
FROM (
 SELECT t1.a, t1.b FROM t1
 UNION ALL
 SELECT t2.a, t2.c FROM t2
) t
ORDER BY a;
```

Il est facile de répondre à cette requête si l'on dispose d'un index sur les colonnes `a` des tables `t1` et `t2`: il suffit de parcourir chaque index en parallèle (assurant le tri sur a), en renvoyant la valeur la plus petite.

Pour comparaison, avant la 9.1 et l'introduction du nœud `MergeAppend`, le plan obtenu était celui-ci :

```
                                        QUERY PLAN
-----------------------------------------------------------------------------------
 Sort  (cost=24129.64..24629.64 rows=200000 width=22)
       (actual time=122.705..133.403 rows=200000 loops=1)
   Sort Key: t1.a
   Sort Method:  quicksort  Memory: 21770kB
   ->  Result  (cost=0.00..6520.00 rows=200000 width=22)
               (actual time=0.013..76.527 rows=200000 loops=1)
         ->  Append  (cost=0.00..6520.00 rows=200000 width=22)
                     (actual time=0.012..54.425 rows=200000 loops=1)
               ->  Seq Scan on t1  (cost=0.00..2110.00 rows=100000 width=23)
                                   (actual time=0.011..19.379 rows=100000 loops=1)
               ->  Seq Scan on t2  (cost=0.00..4410.00 rows=100000 width=22)
                                   (actual time=1.531..22.050 rows=100000 loops=1)
 Total runtime: 141.708 ms
```

Depuis la 9.1, l'optimiseur est capable de détecter qu'il existe un **parcours paramétré**, renvoyant les données triées sur la clé demandée (a), et utilise la stratégie `MergeAppend` :

```
                                        QUERY PLAN
-----------------------------------------------------------------------------------
 Merge Append  (cost=0.72..14866.72 rows=300000 width=23)
               (actual time=0.040..76.783 rows=300000 loops=1)
   Sort Key: t1.a
   ->  Index Scan using t1_pkey on t1  (cost=0.29..3642.29 rows=100000 width=22)
                                       (actual time=0.014..18.876 rows=100000 loops=1)
```

```
   ->  Index Scan using t2_pkey on t2  (cost=0.42..7474.42 rows=200000 width=23)
                                       (actual time=0.025..35.920 rows=200000 loops=1)
 Total runtime: 85.019 ms
```

Cette optimisation est d'autant plus intéressante si l'on utilise une clause `LIMIT`.

Sans `MergeAppend` :

```
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Limit  (cost=9841.93..9841.94 rows=5 width=22)
        (actual time=119.946..119.946 rows=5 loops=1)
   ->  Sort  (cost=9841.93..10341.93 rows=200000 width=22)
             (actual time=119.945..119.945 rows=5 loops=1)
         Sort Key: t1.a
         Sort Method:  top-N heapsort  Memory: 25kB
         ->  Result  (cost=0.00..6520.00 rows=200000 width=22)
                     (actual time=0.008..75.482 rows=200000 loops=1)
               ->  Append  (cost=0.00..6520.00 rows=200000 width=22)
                           (actual time=0.008..53.644 rows=200000 loops=1)
                     ->  Seq Scan on t1
                                 (cost=0.00..2110.00 rows=100000 width=23)
                                 (actual time=0.006..18.819 rows=100000 loops=1)
                     ->  Seq Scan on t2
                                 (cost=0.00..4410.00 rows=100000 width=22)
                                 (actual time=1.550..22.119 rows=100000 loops=1)
 Total runtime: 119.976 ms
(9 lignes)
```

Avec `MergeAppend` :

```
 Limit  (cost=0.72..0.97 rows=5 width=23)
        (actual time=0.055..0.060 rows=5 loops=1)
   ->  Merge Append  (cost=0.72..14866.72 rows=300000 width=23)
                     (actual time=0.053..0.058 rows=5 loops=1)
         Sort Key: t1.a
         ->  Index Scan using t1_pkey on t1
                       (cost=0.29..3642.29 rows=100000 width=22)
                       (actual time=0.033..0.036 rows=3 loops=1)
         ->  Index Scan using t2_pkey on t2
                       (cost=0.42..7474.42 rows=200000 width=23) =
                       (actual time=0.019..0.021 rows=3 loops=1)
 Total runtime: 0.117 ms
```

On voit ici que chacun des parcours d'index renvoie 3 lignes, ce qui est suffisant pour renvoyer les 5 lignes ayant la plus faible valeur pour a.

3.12.16 AUTRES

- Nœud HashSetOp Except
  - instructions EXCEPT et EXCEPT ALL
- Nœud HashSetOp Intersect
  - instructions INTERSECT et INTERSECT ALL

La clause `UNION` permet de concaténer deux ensembles de données. Les clauses `EXCEPT` et `INTERSECT` permettent de supprimer une partie de deux ensembles de données.

Voici un exemple basé sur `EXCEPT` :

```
b1=# EXPLAIN SELECT oid FROM pg_proc
    EXCEPT  SELECT oid FROM pg_proc;
                              QUERY PLAN
-----------------------------------------------------------------------
 HashSetOp Except  (cost=0.00..219.39 rows=2342 width=4)
   ->  Append  (cost=0.00..207.68 rows=4684 width=4)
         ->  Subquery Scan on "*SELECT* 1"
             (cost=0.00..103.84 rows=2342 width=4)
               ->  Seq Scan on pg_proc
                   (cost=0.00..80.42 rows=2342 width=4)
         ->  Subquery Scan on "*SELECT* 2"
             (cost=0.00..103.84 rows=2342 width=4)
               ->  Seq Scan on pg_proc
                   (cost=0.00..80.42 rows=2342 width=4)
(6 rows)
```

Et un exemple basé sur `INTERSECT` :

```
b1=# EXPLAIN   SELECT oid FROM pg_proc
    INTERSECT SELECT oid FROM pg_proc;
                              QUERY PLAN
-----------------------------------------------------------------------
 HashSetOp Intersect  (cost=0.00..219.39 rows=2342 width=4)
   ->  Append  (cost=0.00..207.68 rows=4684 width=4)
         ->  Subquery Scan on "*SELECT* 1"
             (cost=0.00..103.84 rows=2342 width=4)
               ->  Seq Scan on pg_proc
                   (cost=0.00..80.42 rows=2342 width=4)
         ->  Subquery Scan on "*SELECT* 2"
             (cost=0.00..103.84 rows=2342 width=4)
               ->  Seq Scan on pg_proc
                   (cost=0.00..80.42 rows=2342 width=4)
(6 rows)
```

3.12.17 DIVERS

- Prend un ensemble de données en entrée
- Et renvoie un ensemble de données
- Exemples typiques
  - Sort
  - Aggregate
  - Unique
  - Limit
  - InitPlan, SubPlan

Tous les autres nœuds que nous allons voir prennent un seul ensemble de données en entrée et en renvoient un aussi. Ce sont des nœuds d'opérations simples comme le tri, l'agrégat, l'unicité, la limite, etc.

3.12.18 SORT

- Utilisé pour le ORDER BY
  - Mais aussi DISTINCT, GROUP BY, UNION
  - Les jointures de type Merge Join
- Gros délai de démarrage
- Trois types de tri
  - En mémoire, tri quicksort
  - En mémoire, tri top-N heapsort (si clause LIMIT)
  - Sur disque

PostgreSQL peut faire un tri de trois façons.

Les deux premières sont manuelles. Il lit toutes les données nécessaires et les trie en mémoire. La quantité de mémoire utilisable dépend du paramètre `work_mem`. S'il n'a pas assez de mémoire, il utilisera un stockage sur disque. La rapidité du tri dépend principalement de la mémoire utilisable mais aussi de la puissance des processeurs. Le tri effectué est un tri quicksort sauf si une clause `LIMIT` existe, auquel cas, le tri sera un top-N heapsort. La troisième méthode est de passer par un index Btree. En effet, ce type d'index stocke les données de façon triée. Dans ce cas, PostgreSQL n'a pas besoin de mémoire.

Le choix entre ces trois méthodes dépend principalement de `work_mem`. En fait, le pseudo-code ci-dessous explique ce choix :

```
Si les données de tri tiennent dans work_mem
    Si une clause LIMIT est présente
```

```
        Tri top-N heapsort
    Sinon
        Tri quicksort
Sinon
    Tri sur disque
```

Voici quelques exemples :

- un tri externe

```
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id;
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=150385.45..153040.45 rows=1062000 width=4)
       (actual time=807.603..941.357 rows=1000000 loops=1)
   Sort Key: id
   Sort Method: external sort  Disk: 17608kB
   ->  Seq Scan on t2  (cost=0.00..15045.00 rows=1062000 width=4)
                 (actual time=0.050..143.918 rows=1000000 loops=1)
 Total runtime: 1021.725 ms
(5 rows)
```

- un tri en mémoire

```
b1=# SET work_mem TO '100MB';
SET
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id;
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=121342.45..123997.45 rows=1062000 width=4)
       (actual time=308.129..354.035 rows=1000000 loops=1)
   Sort Key: id
   Sort Method: quicksort  Memory: 71452kB
   ->  Seq Scan on t2  (cost=0.00..15045.00 rows=1062000 width=4)
                 (actual time=0.088..142.787 rows=1000000 loops=1)
 Total runtime: 425.160 ms
(5 rows)
```

- un tri en mémoire

```
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id LIMIT 10000;
                                 QUERY PLAN
-----------------------------------------------------------------------
 Limit  (cost=85863.56..85888.56 rows=10000 width=4)
        (actual time=271.674..272.980 rows=10000 loops=1)
   ->  Sort  (cost=85863.56..88363.56 rows=1000000 width=4)
             (actual time=271.671..272.240 rows=10000 loops=1)
         Sort Key: id
         Sort Method: top-N heapsort  Memory: 1237kB
```

```
        ->  Seq Scan on t2  (cost=0.00..14425.00 rows=1000000 width=4)
                    (actual time=0.031..146.306 rows=1000000 loops=1)
 Total runtime: 273.665 ms
(6 rows)
```

- un tri par un index

```
b1=# CREATE INDEX ON t2(id);
CREATE INDEX
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id;
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using t2_id_idx on t2
    (cost=0.00..30408.36 rows=1000000 width=4)
    (actual time=0.145..308.651 rows=1000000 loops=1)
 Total runtime: 355.175 ms
(2 rows)
```

Le paramètre `enable_sort` permet de défavoriser l'utilisation d'un tri. Dans ce cas, le planificateur tendra à préférer l'utilisation d'un index, qui retourne des données déjà triées.

Augmenter la valeur du paramètre `work_mem` aura l'effet inverse : favoriser un tri plutôt que l'utilisation d'un index.

3.12.19 AGGREGATE

- Agrégat complet
- Pour un seul résultat

Il existe plusieurs façons de réaliser un agrégat :

- l'agrégat standard;
- l'agrégat par tri des données;
- et l'agrégat par hachage;

ces deux derniers sont utilisés quand la clause `SELECT` contient des colonnes en plus de la fonction d'agrégat.

Par exemple, pour un seul résultat count(*), nous aurons ce plan d'exécution :

```
b1=# EXPLAIN SELECT count(*) FROM pg_proc;
                            QUERY PLAN
-------------------------------------------------------------------
 Aggregate  (cost=86.28..86.29 rows=1 width=0)
   ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=0)
(2 rows)
```

Seul le parcours séquentiel est possible ici car count() doit compter toutes les lignes.

Autre exemple avec une fonction d'agrégat max.

```
b1=# EXPLAIN SELECT max(proname) FROM pg_proc;
                            QUERY PLAN
-------------------------------------------------------------------
 Aggregate  (cost=92.13..92.14 rows=1 width=64)
   ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=64)
(2 rows)
```

Il existe une autre façon de récupérer la valeur la plus petite ou la plus grande : passer par l'index. Ce sera très rapide car l'index est trié.

```
b1=# EXPLAIN SELECT max(oid) FROM pg_proc;
                            QUERY PLAN
-------------------------------------------------------------------------
 Result  (cost=0.13..0.14 rows=1 width=0)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.00..0.13 rows=1 width=4)
           ->  Index Scan Backward using pg_proc_oid_index on pg_proc
               (cost=0.00..305.03 rows=2330 width=4)
                 Index Cond: (oid IS NOT NULL)
(5 rows)
```

Il est à noter que ce n'est pas valable pour les valeurs de type booléen jusqu'en 9.2.

3.12.20 HASH AGGREGATE

- Hachage de chaque n-uplet de regroupement (group by)
- accès direct à chaque n-uplet pour appliquer fonction d'agrégat
- Intéressant si l'ensemble des valeurs distinctes tient en mémoire, dangereux sinon

Voici un exemple de ce type de nœud :

```
b1=# EXPLAIN SELECT proname, count(*) FROM pg_proc GROUP BY proname;
                            QUERY PLAN
-------------------------------------------------------------------
 HashAggregate  (cost=92.13..111.24 rows=1911 width=64)
   ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=64)
(2 rows)
```

Le hachage occupe de la place en mémoire, le plan n'est choisi que si PostgreSQL estime que si la table de hachage générée tient dans work_mem. **C'est le seul type de nœud qui peut dépasser work_mem :** la seule façon d'utiliser le HashAggregate est en mémoire, il est donc agrandi s'il est trop petit.

Quant au paramètre `enable_hashagg`, il permet d'activer et de désactiver l'utilisation de ce type de nœud.

3.12.21 GROUP AGGREGATE

- Reçoit des données déjà triées
- Parcours des données
  - Regroupement du groupe précédent arrivé à une donnée différente

Voici un exemple de ce type de nœud :

```
b1=# EXPLAIN SELECT proname, count(*) FROM pg_proc GROUP BY proname;
                              QUERY PLAN
-----------------------------------------------------------------------
 GroupAggregate  (cost=211.50..248.17 rows=1911 width=64)
   ->  Sort  (cost=211.50..217.35 rows=2342 width=64)
         Sort Key: proname
         ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=64)
(4 rows)
```

Un parcours d'index est possible pour remplacer le parcours séquentiel et le tri.

3.12.22 UNIQUE

- Reçoit des données déjà triées
- Parcours des données
  - Renvoi de la donnée précédente une fois arrivé à une donnée différente
- Résultat trié

Le nœud `Unique` permet de ne conserver que les lignes différentes. L'opération se réalise en triant les données, puis en parcourant le résultat trié. Là aussi, un index aide à accélérer ce type de nœud.

En voici un exemple :

```
b1=# EXPLAIN SELECT DISTINCT pronamespace FROM pg_proc;
                              QUERY PLAN
-----------------------------------------------------------------------
 Unique  (cost=211.57..223.28 rows=200 width=4)
   ->  Sort  (cost=211.57..217.43 rows=2343 width=4)
         Sort Key: pronamespace
```

```
        ->  Seq Scan on sample4  (cost=0.00..80.43 rows=2343 width=4)
(4 rows)
```

3.12.23 LIMIT

- Permet de limiter le nombre de résultats renvoyés
- Utilisé par
  - clauses LIMIT et OFFSET d'une requête SELECT
  - fonctions min() et max() quand il n'y a pas de clause WHERE et qu'il y a un index
- Le nœud précédent sera de préférence un nœud dont le coût de démarrage est peu élevé (SeqScan, NestedLoop)

Voici un exemple de l'utilisation d'un nœud `Limit` :

```
b1=# EXPLAIN SELECT 1 FROM pg_proc LIMIT 10;
                            QUERY PLAN
------------------------------------------------------------------
 Limit  (cost=0.00..0.34 rows=10 width=0)
   ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=0)
(2 rows)
```

3.13 TRAVAUX PRATIQUES

3.13.1 ÉNONCÉS

Préambule

- Utilisez `\timing` dans `psql` pour afficher les temps d'exécution de la recherche.
- Afin d'éviter tout effet dû au cache, autant du plan que des pages de données, nous utilisons parfois une sous-requête avec un résultat non déterministe (`random`).
- N'oubliez pas de lancer plusieurs fois les requêtes. Vous pouvez les rappeler avec `\g`, ou utiliser la touche **flèche haut** du clavier si votre installation utilise `readline` ou `libedit`.

- Vous devrez disposer de la base `cave` pour ce TP.
- Les valeurs (taille, temps d'exécution) varieront à cause de plusieurs critères :
  - les machines sont différentes ;
  - le jeu de données peut avoir partiellement changé depuis la rédaction du TP.

Affichage de plans de requêtes simples

Recherche de motif texte

- Affichez le plan de cette requête (sur la base `cave`) :

```
SELECT * FROM appellation WHERE libelle LIKE 'Brouilly%';
```

Que constatez-vous ?

- Affichez maintenant le nombre de blocs accédés par cette requête.
- Cette requête ne passe pas par un index. Essayez de lui forcer la main.
- L'index n'est toujours pas utilisé. L'index « par défaut » n'est pas capable de répondre à des questions sur motif.
- Créez un index capable de réaliser ces opérations. Testez à nouveau le plan.
- Réactivez `enable_seqscan`. Testez à nouveau le plan.
- Quelle est la conclusion ?

Recherche de motif texte avancé

La base `cave` ne contient pas de données textuelles appropriées, nous allons en utiliser une autre.

- Lancez `textes.sql` ou `textes_10pct.sql` (préférable sur une machine peu puissante, ou une instance PostgreSQL non paramétrée).

```
psql < textes_10pct.sql
```

Ce script crée une table `textes`, contenant le texte intégral d'un grand nombre de livres en français du projet Gutenberg, soit 10 millions de lignes pour 85 millions de mots.

Nous allons rechercher toutes les références à « Fantine » dans les textes. On devrait trouver beaucoup d'enregistrements provenant des « Misérables ».

- La méthode SQL standard pour écrire cela est :

17.12

```
SELECT * FROM textes WHERE contenu ILIKE '%fantine%';
```

Exécutez cette requête, et regardez son plan d'exécution.

Nous lisons toute la table à chaque fois. C'est normal et classique avec une base de données : non seulement la recherche est insensible à la casse, mais elle commence par %, ce qui est incompatible avec une indexation btree classique.

Nous allons donc utiliser l'extension pg_trgm :

- Créez un index trigramme :

```
textes=# CREATE EXTENSION pg_trgm;
CREATE INDEX idx_trgm ON textes USING gist (contenu gist_trgm_ops);
-- ou CREATE INDEX idx_trgm ON textes USING gin (contenu gin_trgm_ops);
```

- Quelle est la taille de l'index ?
- Réexécutez la requête. Que constatez-vous ?
- Suivant que vous ayez opté pour GiST ou Gin, refaites la manipulation avec l'autre méthode d'indexation.
- Essayez de créer un index « Full Text » à la place de l'index trigramme. Quels sont les résultats ?

Optimisation d'une requête

Schéma de la base cave

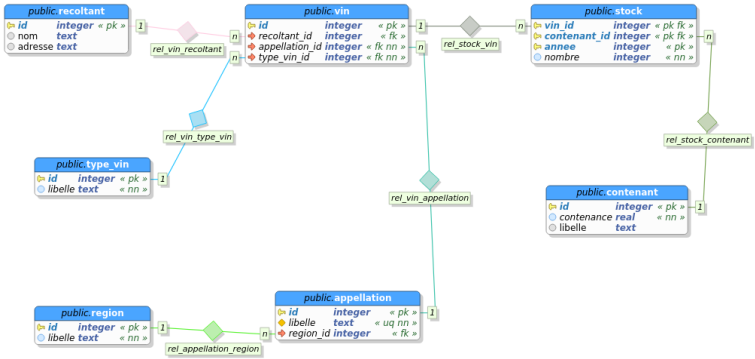


FIGURE 4: SCHÉMA DE LA BASE CAVE

Optimisation 1

Nous travaillerons sur la requête contenue dans le fichier `requete1.sql` pour cet exercice :

```
-- \timing

-- explain analyze
select
      m.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
from
     contenant c
     join stock s
       on s.contenant_id = c.id
     join (select round(random()*50)+1950 as annee) m
       on s.annee =  m.annee
     join vin v
       on s.vin_id = v.id
     left join appellation a
     on v.appellation_id = a.id
group by m.annee||' - '||a.libelle;
```

- Exécuter la requête telle quelle et noter le plan et le temps d'exécution.
- Créer un index sur la colonne `stock.annee`.
- Exécuter la requête juste après la création de l'index
- Faire un `ANALYZE stock`.
- Exécuter à nouveau la requête.
- Interdire à PostgreSQL les *sequential scans* avec la commande `set enable_seqscan to off ;` dans votre session dans `psql`.
- Exécuter à nouveau la requête.
- Tenter de réécrire la requête pour l'optimiser.

Optimisation 2

L'exercice nous a amené à la réécriture de la requête

- Voici la requête que nous avons à présent :

```
explain analyze
select
      s.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
from
     contenant c
     join stock s
       on s.contenant_id = c.id
     join vin v
       on s.vin_id = v.id
     left join appellation a
       on v.appellation_id = a.id
where s.annee =  (select round(random()*50)+1950 as annee)
group by s.annee||' - '||a.libelle;
```

Cette écriture n'est pas optimale, pourquoi ?

Indices

- Vérifiez le schéma de données de la base `cave`.
- Faites les requêtes de vérification nécessaires pour vous assurer que vous avez bien trouvé une anomalie dans la requête.
- Réécrivez la requête une nouvelle fois et faites un `EXPLAIN ANALYZE` pour vérifier que le plan d'exécution est plus simple et plus rapide avec cette nouvelle écriture.

Optimisation 3

Un dernier problème existe dans cette requête. Il n'est visible qu'en observant le plan d'exécution de la requête précédente.

Indice

Cherchez une opération présente dans le plan qui n'apparaît pas dans la requête. Comment modifier la requête pour éviter cette opération ?

Corrélation entre colonnes

- Importez le fichier `correlations.sql`.

Dans la table `villes`, on trouve les villes et leur code postal. Ces colonnes sont très fortement corrélées, mais pas identiques : plusieurs villes peuvent partager le même code postal, et une ville peut avoir plusieurs codes postaux. On peut aussi, bien sûr, avoir

plusieurs villes avec le même nom, mais pas le même code postal (dans des départements différents par exemple). Pour obtenir la liste des villes pouvant poser problème :

```
SELECT *
FROM villes
WHERE localite IN
    (SELECT localite
     FROM villes
     GROUP BY localite HAVING count(*) >1)
  AND codepostal IN
    (SELECT codepostal
     FROM villes
     GROUP BY codepostal HAVING count(*) >1);
```

Avec cette requête, on récupére toutes les villes ayant plusieurs occurrences et dont au moins une possède un code postal partagé. Ces villes ont donc besoin du code postal ET du nom pour être identifiées.

Un exemple de requête problématique est le suivant :

```
SELECT * FROM colis
WHERE id_ville IN
   (SELECT id_ville FROM villes
    WHERE localite ='PARIS'
      AND codepostal LIKE '75%')
```

- Exécutez cette requête, et regardez son plan d'exécution. Où est le problème ?
- Exécutez cette requête sans la dernière clause `AND codepostal LIKE '75%'`. Que constatez-vous ?
- Quelle solution pourrait-on adopter, si on doit réellement spécifier ces deux conditions ?

3.13.2 SOLUTIONS

Affichage de plans de requêtes simples

Recherche de motif texte

- Affichez le plan de cette requête (sur la base `cave`).

```
SELECT * FROM appellation WHERE libelle LIKE 'Brouilly%';
```

```
cave=# explain SELECT * FROM appellation WHERE libelle LIKE 'Brouilly%';
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on appellation  (cost=0.00..6.99 rows=1 width=24)
```

```
   Filter: (libelle ~~ 'Brouilly%'::text)
(2 lignes)
```

Que constatez-vous ?

- Affichez maintenant le nombre de blocs accédés par cette requête.

```
cave=# explain (analyze,buffers) SELECT * FROM appellation
cave=# WHERE libelle LIKE 'Brouilly%';
                                      QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on appellation  (cost=0.00..6.99 rows=1 width=24)
                          (actual time=0.066..0.169 rows=1 loops=1)
   Filter: (libelle ~~ 'Brouilly%'::text)
   Rows Removed by Filter: 318
   Buffers: shared hit=3
 Total runtime: 0.202 ms
(5 lignes)
```

- Cette requête ne passe pas par un index. Essayez de lui forcer la main.

```
cave=# set enable_seqscan TO off;
SET
cave=# explain (analyze,buffers) SELECT * FROM appellation
cave=# WHERE libelle LIKE 'Brouilly%';
                                      QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on appellation  (cost=10000000000.00..10000000006.99 rows=1 width=24)
                          (actual time=0.073..0.197 rows=1 loops=1)
   Filter: (libelle ~~ 'Brouilly%'::text)
   Rows Removed by Filter: 318
   Buffers: shared hit=3
 Total runtime: 0.238 ms
(5 lignes)
```

Passer `enable_seqscan` à « off » n'interdit pas l'utilisation des scans séquentiels. Il ne fait que les défavoriser fortement : regardez le coût estimé du scan séquentiel.

- L'index n'est toujours pas utilisé. L'index « par défaut » n'est pas capable de répondre à des questions sur motif.

En effet, l'index par défaut trie les données par la collation de la colonne de la table. Il lui est impossible de savoir que `libelle LIKE 'Brouilly%'` est équivalent à `libelle >= 'Brouilly' AND libelle < 'Brouillz'`. Ce genre de transformation n'est d'ailleurs pas forcément trivial, ni même possible. Il existe dans certaines langues des équivalences (ß et ss en allemand par exemple) qui rendent ce genre de transformation au mieux hasardeuse.

- Créez un index capable de ces opérations. Testez à nouveau le plan.

Pour pouvoir répondre à cette question, on doit donc avoir un index spécialisé, qui compare les chaînes non plus par rapport à leur collation, mais à leur valeur binaire (octale en fait).

```
CREATE INDEX appellation_libelle_key_search
    ON appellation (libelle text_pattern_ops);
```

On indique par cette commande à PostgreSQL de ne plus utiliser la classe d'opérateurs habituelle de comparaison de texte, mais la classe `text_pattern_ops`, qui est spécialement faite pour les recherches `LIKE 'xxxx%'` : cette classe ne trie plus les chaînes par leur ordre alphabétique, mais par leur valeur octale.

Si on redemande le plan :

```
cave=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM appellation
cave=# WHERE libelle LIKE 'Brouilly%';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using appellation_libelle_key_search on appellation
                              (cost=0.27..8.29 rows=1 width=24)
                              (actual time=0.057..0.059 rows=1 loops=1)
   Index Cond: ((libelle ~>=~ 'Brouilly'::text)
                AND (libelle ~<~ 'Brouillz'::text))
   Filter: (libelle ~~ 'Brouilly%'::text)
   Buffers: shared hit=1 read=2
 Total runtime: 0.108 ms
(5 lignes)
```

On utilise enfin un index.

- Réactivez `enable_seqscan`. Testez à nouveau le plan.

```
cave=# reset enable_seqscan ;
RESET
cave=# explain (analyze,buffers) SELECT * FROM appellation
cave=# WHERE libelle LIKE 'Brouilly%';
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on appellation  (cost=0.00..6.99 rows=1 width=24)
                          (actual time=0.063..0.172 rows=1 loops=1)
   Filter: (libelle ~~ 'Brouilly%'::text)
   Rows Removed by Filter: 318
   Buffers: shared hit=3
 Total runtime: 0.211 ms
(5 lignes)
```

- Quelle est la conclusion ?

PostgreSQL choisit de ne pas utiliser cet index. Le temps d'exécution est pourtant un peu

meilleur avec l'index (60 microsecondes contre 172 microsecondes). Néanmoins, cela n'est vrai que parce que les données sont en cache. En cas de données hors du cache, le plan par parcours séquentiel (*seq scan*) est probablement meilleur. Certes il prend plus de temps CPU puisqu'il doit consulter 318 enregistrements inutiles. Par contre, il ne fait qu'un accès à 3 blocs séquentiels (les 3 blocs de la table), ce qui est le plus sûr.

La table est trop petite pour que PostgreSQL considère l'utilisation d'un index.

Recherche de motif texte avancé

La base `cave` ne contient pas de données textuelles appropriées, nous allons en utiliser une autre.

- Lancez `textes.sql` ou `textes_10pct.sql` (préférable sur une machine peu puissante, ou une instance PostgreSQL non paramétrée).

```
psql < textes.sql
```

Ce script crée une table `textes`, contenant le texte intégral d'un grand nombre de livres en français du projet Gutenberg, soit 10 millions de lignes pour 85 millions de mots.

Nous allons rechercher toutes les références à « Fantine » dans les textes. On devrait trouver beaucoup d'enregistrements provenant des « Misérables ».

- La méthode SQL standard pour écrire cela est :

```
SELECT * FROM textes WHERE contenu ILIKE '%fantine%';
```

Exécutez cette requête, et regardez son plan d'exécution.

```
textes=# explain (analyze,buffers) SELECT * FROM textes
textes=# WHERE contenu ILIKE '%fantine%';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on textes  (cost=0.00..325809.40 rows=874 width=102)
                     (actual time=224.634..22567.231 rows=921 loops=1)
   Filter: (contenu ~~* '%fantine%'::text)
   Rows Removed by Filter: 11421523
   Buffers: shared hit=130459 read=58323
 Total runtime: 22567.679 ms
(5 lignes)
```

Cette requête ne peut pas être optimisée avec les index standard (`btree`) : c'est une recherche insensible à la casse et avec plusieurs % dont un au début.

Avec GiST

- Créez un index trigramme:

```
textes=# CREATE EXTENSION pg_trgm;

textes=# CREATE INDEX idx_trgm ON textes USING gist (contenu gist_trgm_ops);
CREATE INDEX
Temps : 962794,399 ms
```

- Quelle est la taille de l'index ?

L'index fait cette taille (pour une table de 1,5Go) :

```
textes=# select pg_size_pretty(pg_relation_size('idx_trgm'));
 pg_size_pretty
----------------
 2483 MB
(1 ligne)
```

- Réexécutez la requête. Que constatez-vous ?

```
textes=# explain (analyze,buffers) SELECT * FROM textes
textes=# WHERE contenu ILIKE '%fantine%';
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=111.49..3573.39 rows=912 width=102)
                             (actual time=1942.872..1949.393 rows=922 loops=1)
   Recheck Cond: (contenu ~~* '%fantine%'::text)
   Rows Removed by Index Recheck: 75
   Buffers: shared hit=16030 read=144183 written=14741
   ->  Bitmap Index Scan on idx_trgm  (cost=0.00..111.26 rows=912 width=0)
                                      (actual time=1942.671..1942.671 rows=997 loops=1)
         Index Cond: (contenu ~~* '%fantine%'::text)
         Buffers: shared hit=16029 read=143344 written=14662
 Total runtime: 1949.565 ms
(8 lignes)

Temps : 1951,175 ms
```

PostgreSQL dispose de mécanismes spécifiques avancés pour certains types de données. Ils ne sont pas toujours installés en standard, mais leur connaissance peut avoir un impact énorme sur les performances.

Le mécanisme GiST est assez efficace pour répondre à ce genre de questions. Il nécessite quand même un accès à un grand nombre de blocs, d'après le plan : 160 000 blocs lus, 15 000 écrits (dans un fichier temporaire, on pourrait s'en débarrasser en augmentant le `work_mem`). Le gain est donc conséquent, mais pas gigantesque : le plan initial lisait 190 000 blocs. On gagne surtout en temps de calcul, car on accède directement aux bons enregistrements. Le parcours de l'index, par contre, est coûteux.

Avec Gin

- Créez un index trigramme:

```
textes=# CREATE EXTENSION pg_trgm;

textes=# CREATE INDEX idx_trgm ON textes USING gin (contenu gin_trgm_ops);
CREATE INDEX
Temps : 591534,917 ms
```

L'index fait cette taille (pour une table de 1,5Go) :

```
textes=# select pg_size_pretty(pg_total_relation_size('textes'));
 pg_size_pretty
----------------
 4346 MB
(1 ligne)
```

L'index est très volumineux.

- Réexécutez la requête. Que constatez-vous ?

```
textes=# explain (analyze,buffers) SELECT * FROM textes
textes=# WHERE contenu ILIKE '%fantine%';
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=103.06..3561.22 rows=911 width=102)
                             (actual time=777.469..780.834 rows=921 loops=1)
   Recheck Cond: (contenu ~~* '%fantine%'::text)
   Rows Removed by Index Recheck: 75
   Buffers: shared hit=2666
   ->  Bitmap Index Scan on idx_trgm  (cost=0.00..102.83 rows=911 width=0)
                              (actual time=777.283..777.283 rows=996 loops=1)
         Index Cond: (contenu ~~* '%fantine%'::text)
         Buffers: shared hit=1827
 Total runtime: 780.954 ms
(8 lignes)
```

PostgreSQL dispose de mécanismes spécifiques avancés pour certains types de données. Ils ne sont pas toujours installés en standard, mais leur connaissance peut avoir un impact énorme sur les performances. Le mécanisme Gin est vraiment très efficace pour répondre à ce genre de questions. Il s'agit de répondre en moins d'une seconde à « quelles lignes contiennent la chaîne "fantine" ? » sur 12 millions de lignes de texte. Les Index Gin sont par contre très coûteux à maintenir. Ici, on n'accède qu'à 2 666 blocs, ce qui est vraiment excellent. Mais l'index est bien plus volumineux que l'index GiST.

Avec le Full Text Search

Le résultat sera bien sûr différent, et le FTS est moins souple.

Version GiST :

```
textes=# create index idx_fts
          on textes
          using gist (to_tsvector('french',contenu));
CREATE INDEX
Temps : 1807467,811 ms


textes=# EXPLAIN (analyze,buffers) SELECT * FROM textes
textes=# WHERE to_tsvector('french',contenu) @@ to_tsquery('french','fantine');
                 QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=2209.51..137275.87 rows=63109 width=97)
                             (actual time=648.596..659.733 rows=311 loops=1)
   Recheck Cond: (to_tsvector('french'::regconfig, contenu) @@
                  '''fantin'''::tsquery)
   Buffers: shared hit=37165
   ->  Bitmap Index Scan on idx_fts  (cost=0.00..2193.74 rows=63109 width=0)
                                (actual time=648.493..648.493 rows=311 loops=1)
         Index Cond: (to_tsvector('french'::regconfig, contenu) @@
                      '''fantin'''::tsquery)
         Buffers: shared hit=37016
 Total runtime: 659.820 ms
(7 lignes)

Temps : 660,364 ms
```

Et la taille de l'index :

```
textes=# select pg_size_pretty(pg_relation_size('idx_fts'));
 pg_size_pretty
----------------
 671 MB
(1 ligne)
```

Version Gin :

```
textes=# CREATE INDEX idx_fts ON textes
textes=# USING gin (to_tsvector('french',contenu));
CREATE INDEX
Temps : 491499,599 ms
textes=# EXPLAIN (analyze,buffers) SELECT * FROM textes
textes=# WHERE to_tsvector('french',contenu) @@ to_tsquery('french','fantine');
                                                    QUERY PLAN
```

```
------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=693.10..135759.45 rows=63109 width=97)
                             (actual time=0.278..0.699 rows=311 loops=1)
   Recheck Cond: (to_tsvector('french'::regconfig, contenu) @@
                 '''fantin'''::tsquery)
   Buffers: shared hit=153
   ->  Bitmap Index Scan on idx_fts  (cost=0.00..677.32 rows=63109 width=0)
                                     (actual time=0.222..0.222 rows=311 loops=1)
         Index Cond: (to_tsvector('french'::regconfig, contenu) @@
                      '''fantin'''::tsquery)
         Buffers: shared hit=4
 Total runtime: 0.793 ms
(7 lignes)

Temps : 1,534 ms
```

Taille de l'index :

```
textes=# select pg_size_pretty(pg_relation_size('idx_fts'));
 pg_size_pretty
----------------
 593 MB
(1 ligne)
```

On constate donc que le Full Text Search est bien plus efficace que le trigramme, du moins pour le Full Text Search + Gin : trouver 1 mot parmi plus de cent millions, dans 300 endroits différents dure 1,5 ms.

Par contre, le trigramme permet des recherches floues (orthographe approximative), et des recherches sur autre chose que des mots, même si ces points ne sont pas abordés ici.

Optimisation d'une requête

Optimisation 1

Nous travaillerons sur la requête contenue dans le fichier `requete1.sql` pour cet exercice:

```
-- \timing

-- explain analyze
select
      m.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
```

```
from
     contenant c
     join stock s
       on s.contenant_id = c.id
     join (select round(random()*50)+1950 as annee) m
       on s.annee =  m.annee
     join vin v
       on s.vin_id = v.id
     left join appellation a
     on v.appellation_id = a.id
group by m.annee||' - '||a.libelle;
```

L'exécution de la requête donne le plan suivant, avec un temps qui peut varier en fonction de la machine utilisée et de son activité:

```
HashAggregate  (cost=12763.56..12773.13 rows=319 width=32)
               (actual time=1542.472..1542.879 rows=319 loops=1)
   ->  Hash Left Join  (cost=184.59..12741.89 rows=2889 width=32)
                       (actual time=180.263..1520.812 rows=11334 loops=1)
         Hash Cond: (v.appellation_id = a.id)
         ->  Hash Join  (cost=174.42..12663.10 rows=2889 width=20)
                        (actual time=179.426..1473.270 rows=11334 loops=1)
               Hash Cond: (s.contenant_id = c.id)
               ->  Hash Join  (cost=173.37..12622.33 rows=2889 width=20)
                              (actual time=179.401..1446.687 rows=11334 loops=1)
                     Hash Cond: (s.vin_id = v.id)
                     ->  Hash Join  (cost=0.04..12391.22 rows=2889 width=20)
                            (actual time=164.388..1398.643 rows=11334 loops=1)
                           Hash Cond: ((s.annee)::double precision =
                                ((round((random() * 50::double precision)) +
                                1950::double precision)))
                           ->  Seq Scan on stock s
                                (cost=0.00..9472.86 rows=577886 width=16)
                                (actual time=0.003..684.039 rows=577886 loops=1)
                           ->  Hash  (cost=0.03..0.03 rows=1 width=8)
                                     (actual time=0.009..0.009 rows=1 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                 ->  Result  (cost=0.00..0.02 rows=1 width=0)
                                     (actual time=0.005..0.006 rows=1 loops=1)
                     ->  Hash  (cost=97.59..97.59 rows=6059 width=8)
                               (actual time=14.987..14.987 rows=6059 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 237kB
```

```
                          ->  Seq Scan on vin v
                                (cost=0.00..97.59 rows=6059 width=8)
                                (actual time=0.009..7.413 rows=6059 loops=1)
                ->  Hash  (cost=1.02..1.02 rows=2 width=8)
                          (actual time=0.013..0.013 rows=2 loops=1)
                      Buckets: 1024  Batches: 1  Memory Usage: 1kB
                      ->  Seq Scan on contenant c
                             (cost=0.00..1.02 rows=2 width=8)
                             (actual time=0.003..0.005 rows=2 loops=1)
          ->  Hash  (cost=6.19..6.19 rows=319 width=20)
                    (actual time=0.806..0.806 rows=319 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 17kB
                ->  Seq Scan on appellation a
                     (cost=0.00..6.19 rows=319 width=20)
                     (actual time=0.004..0.379 rows=319 loops=1)
 Total runtime: 1543.242 ms
(23 rows)
```

Nous créons à présent un index sur `stock.annee` comme suit :

```
create index stock_annee on stock (annee) ;
```

Et exécutons à nouveau la requête. Hélas nous constatons que rien ne change, ni le plan, ni le temps pris par la requête.

Nous n'avons pas lancé `ANALYZE`, cela explique que l'optimiseur n'utilise pas l'index : il n'en a pas encore la connaissance.

```
ANALYZE STOCK ;
```

Le plan n'a toujours pas changé ! Ni le temps d'exécution ?!

Interdisons donc de faire les *seq scans* à l'optimiseur :

```
SET ENABLE_SEQSCAN TO OFF;
```

Nous remarquons que le plan d'exécution est encore pire :

```
 HashAggregate  (cost=40763.39..40772.96 rows=319 width=32)
                (actual time=2022.971..2023.390 rows=319 loops=1)
   ->  Hash Left Join  (cost=313.94..40741.72 rows=2889 width=32)
                       (actual time=18.149..1995.889 rows=11299 loops=1)
         Hash Cond: (v.appellation_id = a.id)
         ->  Hash Join  (cost=290.92..40650.09 rows=2889 width=20)
                        (actual time=17.172..1937.644 rows=11299 loops=1)
               Hash Cond: (s.vin_id = v.id)
```

```
                ->  Nested Loop  (cost=0.04..40301.43 rows=2889 width=20)
                                 (actual time=0.456..1882.531 rows=11299 loops=1)
                      Join Filter: (s.contenant_id = c.id)
                      ->  Hash Join  (cost=0.04..40202.48 rows=2889 width=20)
                                 (actual time=0.444..1778.149 rows=11299 loops=1)
                            Hash Cond: ((s.annee)::double precision =
                                 ((round((random() * 50::double precision)) +
                                 1950::double precision)))
                            ->  Index Scan using stock_pkey on stock s
                                 (cost=0.00..37284.12 rows=577886 width=16)
                                 (actual time=0.009..1044.061 rows=577886 loops=1)
                            ->  Hash  (cost=0.03..0.03 rows=1 width=8)
                                       (actual time=0.011..0.011 rows=1 loops=1)
                                  Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                  ->  Result  (cost=0.00..0.02 rows=1 width=0)
                                       (actual time=0.005..0.006 rows=1 loops=1)
                      ->  Materialize  (cost=0.00..12.29 rows=2 width=8)
                                 (actual time=0.001..0.003 rows=2 loops=11299)
                            ->  Index Scan using contenant_pkey on contenant c
                                    (cost=0.00..12.28 rows=2 width=8)
                                    (actual time=0.004..0.010 rows=2 loops=1)
                ->  Hash  (cost=215.14..215.14 rows=6059 width=8)
                          (actual time=16.699..16.699 rows=6059 loops=1)
                      Buckets: 1024  Batches: 1  Memory Usage: 237kB
                      ->  Index Scan using vin_pkey on vin v
                             (cost=0.00..215.14 rows=6059 width=8)
                             (actual time=0.010..8.871 rows=6059 loops=1)
          ->  Hash  (cost=19.04..19.04 rows=319 width=20)
                    (actual time=0.936..0.936 rows=319 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 17kB
                ->  Index Scan using appellation_pkey on appellation a
                          (cost=0.00..19.04 rows=319 width=20)
                          (actual time=0.016..0.461 rows=319 loops=1)
 Total runtime: 2023.742 ms
(22 rows)
```

Que faire alors ?

Il convient d'autoriser à nouveau les *seq scan*, puis, peut-être, de réécrire la requête.

Nous réécrivons la requête comme suit (fichier `requete2.sql`) :

17.12

```
explain analyze
select
      s.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
from
     contenant c
     join stock s
       on s.contenant_id = c.id
     join vin v
       on s.vin_id = v.id
     left join appellation a
       on v.appellation_id = a.id
where s.annee =  (select round(random()*50)+1950 as annee)
group by s.annee||' - '||a.libelle;
```

Il y a une jointure en moins, ce qui est toujours appréciable. Nous pouvons faire cette réécriture parce que la requête `select round(random()*50)+1950 as annee` ne ramène qu'un seul enregistrement.

Voici le résultat :

```
 HashAggregate  (cost=12734.64..12737.10 rows=82 width=28)
                (actual time=265.899..266.317 rows=319 loops=1)
   InitPlan 1 (returns $0)
     ->  Result  (cost=0.00..0.02 rows=1 width=0)
                 (actual time=0.005..0.006 rows=1 loops=1)
   ->  Hash Left Join  (cost=184.55..12712.96 rows=2889 width=28)
                       (actual time=127.787..245.314 rows=11287 loops=1)
         Hash Cond: (v.appellation_id = a.id)
         ->  Hash Join  (cost=174.37..12634.17 rows=2889 width=16)
                        (actual time=126.950..208.077 rows=11287 loops=1)
               Hash Cond: (s.contenant_id = c.id)
               ->  Hash Join  (cost=173.33..12593.40 rows=2889 width=16)
                              (actual time=126.925..181.867 rows=11287 loops=1)
                     Hash Cond: (s.vin_id = v.id)
                     ->  Seq Scan on stock s
                           (cost=0.00..12362.29 rows=2889 width=16)
                           (actual time=112.101..135.932 rows=11287 loops=1)
                         Filter: ((annee)::double precision = $0)
                     ->  Hash  (cost=97.59..97.59 rows=6059 width=8)
                               (actual time=14.794..14.794 rows=6059 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 237kB
```

```
                        ->  Seq Scan on vin v
                              (cost=0.00..97.59 rows=6059 width=8)
                              (actual time=0.010..7.321 rows=6059 loops=1)
               ->  Hash  (cost=1.02..1.02 rows=2 width=8)
                         (actual time=0.013..0.013 rows=2 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 1kB
                     ->  Seq Scan on contenant c
                            (cost=0.00..1.02 rows=2 width=8)
                            (actual time=0.004..0.006 rows=2 loops=1)
         ->  Hash  (cost=6.19..6.19 rows=319 width=20)
                   (actual time=0.815..0.815 rows=319 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 17kB
               ->  Seq Scan on appellation a
                     (cost=0.00..6.19 rows=319 width=20)
                     (actual time=0.004..0.387 rows=319 loops=1)
 Total runtime: 266.663 ms
(21 rows)
```

Nous sommes ainsi passés de 2 s à 250 ms : la requête est donc environ 10 fois plus rapide.

Que peut-on conclure de cet exercice ?

- que la création d'un index est une bonne idée ; cependant l'optimiseur peut ne pas l'utiliser, pour de bonnes raisons ;
- qu'interdire les *seq scan* est toujours une mauvaise idée (ne présumez pas de votre supériorité sur l'optimiseur !)

Optimisation 2

Voici la requête 2 telle que nous l'avons trouvé dans l'exercice précédent :

```
explain analyze
select
      s.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
from
     contenant c
     join stock s
       on s.contenant_id = c.id
     join vin v
       on s.vin_id = v.id
```

```
    left join appellation a
      on v.appellation_id = a.id
where s.annee =  (select round(random()*50)+1950 as annee)
group by s.annee||' - '||a.libelle;
```

On peut se demander si la jointure externe (LEFT JOIN) est fondée... On va donc vérifier l'utilité de la ligne suivante :

```
vin v left join appellation a on v.appellation_id = a.id
```

Cela se traduit par « récupérer tous les tuples de la table vin, et pour chaque correspondance dans appellation, la récupérer, si elle existe ».

En regardant la description de la table `vin` (`\d vin` dans `psql`), on remarque la contrainte de clé étrangère suivante :

```
« vin_appellation_id_fkey »
  FOREIGN KEY (appellation_id)
  REFERENCES appellation(id)
```

Cela veut dire qu'on a la certitude que pour chaque vin, si une référence à la table appellation est présente, elle est nécessairement vérifiable.

De plus, on remarque :

```
appellation_id | integer | not null
```

Ce qui veut dire que la valeur de ce champ ne peut être nulle. Elle contient donc obligatoirement une valeur qui est présente dans la table `appellation`.

On peut vérifier au niveau des tuples en faisant un `COUNT(*)` du résultat, une fois en `INNER JOIN` et une fois en `LEFT JOIN`. Si le résultat est identique, la jointure externe ne sert à rien :

```
select count(*)
from vin v
  inner join appellation a on (v.appellation_id = a.id);

 count
-------
  6057

select count(*)
from vin v
  left join appellation a on (v.appellation_id = a.id);

 count
-------
  6057
```

On peut donc réécrire la requête 2 sans la jointure externe inutile, comme on vient de le démontrer :

```
explain analyze
select
      s.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
from
     contenant c
     join stock s
       on s.contenant_id = c.id
     join vin v
       on s.vin_id = v.id
     join appellation a
       on v.appellation_id = a.id
where s.annee =  (select round(random()*50)+1950 as annee)
group by s.annee||' - '||a.libelle;
```

Voici le résultat :

```
 HashAggregate  (cost=12734.64..12737.10 rows=82 width=28)
                (actual time=266.916..267.343 rows=319 loops=1)
   InitPlan 1 (returns $0)
     ->  Result  (cost=0.00..0.02 rows=1 width=0)
                 (actual time=0.005..0.006 rows=1 loops=1)
   ->  Hash Join  (cost=184.55..12712.96 rows=2889 width=28)
                  (actual time=118.759..246.391 rows=11299 loops=1)
         Hash Cond: (v.appellation_id = a.id)
         ->  Hash Join  (cost=174.37..12634.17 rows=2889 width=16)
                        (actual time=117.933..208.503 rows=11299 loops=1)
               Hash Cond: (s.contenant_id = c.id)
               ->  Hash Join  (cost=173.33..12593.40 rows=2889 width=16)
                              (actual time=117.914..182.501 rows=11299 loops=1)
                     Hash Cond: (s.vin_id = v.id)
                     ->  Seq Scan on stock s
                            (cost=0.00..12362.29 rows=2889 width=16)
                            (actual time=102.903..135.451 rows=11299 loops=1)
                           Filter: ((annee)::double precision = $0)
                     ->  Hash  (cost=97.59..97.59 rows=6059 width=8)
                               (actual time=14.979..14.979 rows=6059 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 237kB
                           ->  Seq Scan on vin v
```

```
                                (cost=0.00..97.59 rows=6059 width=8)
                                (actual time=0.010..7.387 rows=6059 loops=1)
                  ->  Hash  (cost=1.02..1.02 rows=2 width=8)
                            (actual time=0.009..0.009 rows=2 loops=1)
                        Buckets: 1024  Batches: 1  Memory Usage: 1kB
                        ->  Seq Scan on contenant c
                              (cost=0.00..1.02 rows=2 width=8)
                              (actual time=0.002..0.004 rows=2 loops=1)
            ->  Hash  (cost=6.19..6.19 rows=319 width=20)
                      (actual time=0.802..0.802 rows=319 loops=1)
                  Buckets: 1024  Batches: 1  Memory Usage: 17kB
                  ->  Seq Scan on appellation a
                        (cost=0.00..6.19 rows=319 width=20)
                        (actual time=0.004..0.397 rows=319 loops=1)
 Total runtime: 267.688 ms
(21 rows)
```

Cette réécriture n'a pas d'effet sur le temps d'exécution de la requête dans notre cas. Mais il est probable qu'avec des cardinalités différentes dans la base, cette réécriture aurait eu un impact. Remplacer un `LEFT JOIN` par un `JOIN` est le plus souvent intéressant, car il laisse davantage de liberté au moteur sur le sens de planification des requêtes.

Optimisation 3

Si on observe attentivement le plan, on constate qu'on a toujours le parcours séquentiel de la table `stock`, qui est notre plus grosse table. Pourquoi a-t-il lieu ?

Si on regarde le filtre (ligne `Filter`) du parcours de la table `stock`, on constate qu'il est écrit :

```
  Filter: ((annee)::double precision = $0)
```

Ceci signifie que pour tous les enregistrements de la table, l'année est convertie en nombre en double précision (un nombre à virgule flottante), afin d'être comparée à $0, une valeur filtre appliquée à la table. Cette valeur est le résultat du calcul :

```
select round(random()*50)+1950 as annee
```

comme indiquée par le début du plan (les lignes de l'initplan 1).

Pourquoi compare-t-il l'année, déclarée comme un entier (`integer`), en la convertissant en un nombre à virgule flottante ?

Parce que la fonction `round()` retourne un nombre à virgule flottante. La somme d'un nombre à virgule flottante et d'un entier est évidemment un nombre à virgule flottante. Si on veut que la fonction `round()` retourne un entier, il faut forcer explicitement sa conversion, via `CAST(xxx as int)` ou `::int`.

Réécrivons encore une fois cette requête :

```
explain analyze
select
      s.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
from
     contenant c
     join stock s
       on s.contenant_id = c.id
     join vin v
       on s.vin_id = v.id
     join appellation a
       on v.appellation_id = a.id
where s.annee =  (select cast(round(random()*50) as int)+1950 as annee)
group by s.annee||' - '||a.libelle;
```

Voici son plan :

```
 HashAggregate  (cost=1251.12..1260.69 rows=319 width=28)
                (actual time=138.418..138.825 rows=319 loops=1)
   InitPlan 1 (returns $0)
     ->  Result  (cost=0.00..0.02 rows=1 width=0)
                 (actual time=0.005..0.006 rows=1 loops=1)
   ->  Hash Join  (cost=267.86..1166.13 rows=11329 width=28)
                  (actual time=31.108..118.193 rows=11389 loops=1)
         Hash Cond: (s.contenant_id = c.id)
         ->  Hash Join  (cost=266.82..896.02 rows=11329 width=28)
                        (actual time=31.071..80.980 rows=11389 loops=1)
               Hash Cond: (s.vin_id = v.id)
               ->  Index Scan using stock_annee on stock s
                     (cost=0.00..402.61 rows=11331 width=16)
                     (actual time=0.049..17.191 rows=11389 loops=1)
                      Index Cond: (annee = $0)
               ->  Hash  (cost=191.08..191.08 rows=6059 width=20)
                         (actual time=31.006..31.006 rows=6059 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 313kB
                     ->  Hash Join  (cost=10.18..191.08 rows=6059 width=20)
```

```
                                    (actual time=0.814..22.856 rows=6059 loops=1)
                              Hash Cond: (v.appellation_id = a.id)
                              ->  Seq Scan on vin v
                                    (cost=0.00..97.59 rows=6059 width=8)
                                    (actual time=0.005..7.197 rows=6059 loops=1)
                              ->  Hash  (cost=6.19..6.19 rows=319 width=20)
                                          (actual time=0.800..0.800 rows=319 loops=1)
                                     Buckets: 1024  Batches: 1  Memory Usage: 17kB
                                    ->  Seq Scan on appellation a
                                       (cost=0.00..6.19 rows=319 width=20)
                                       (actual time=0.002..0.363 rows=319 loops=1)
        ->  Hash  (cost=1.02..1.02 rows=2 width=8)
                  (actual time=0.013..0.013 rows=2 loops=1)
            Buckets: 1024  Batches: 1  Memory Usage: 1kB
            ->  Seq Scan on contenant c  (cost=0.00..1.02 rows=2 width=8)
                                    (actual time=0.003..0.006 rows=2 loops=1)
 Total runtime: 139.252 ms
(21 rows)
```

On constate qu'on utilise enfin l'index de `stock`. Le temps d'exécution a encore été divisé par deux.

NB : ce problème d'incohérence de type était la cause du plus gros ralentissement de la requête. En reprenant la requête initiale, et en ajoutant directement le cast, la requête s'exécute déjà en 160 millisecondes.

Corrélation entre colonnes

Importez le fichier `correlations.sql`.

```
createdb correlations
psql correlations < correlations.sql
```

- Exécutez cette requête, et regardez son plan d'exécution. Où est le problème ?

Cette requête a été exécutée dans un environnement où le cache a été intégralement vidé, pour être dans la situation la plus défavorable possible. Vous obtiendrez probablement des performances meilleures, surtout si vous réexécutez cette requête.

```
explain (analyze,buffers)
   SELECT * FROM colis WHERE id_ville IN (
     SELECT id_ville
     FROM villes
     WHERE localite ='PARIS'
```

```
      AND codepostal LIKE '75%'
);
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------
 Nested Loop  (cost=6.75..13533.81 rows=3265 width=16)
              (actual time=38.020..364383.516 rows=170802 loops=1)
   Buffers: shared hit=91539 read=82652
   I/O Timings: read=359812.828
   ->  Seq Scan on villes  (cost=0.00..1209.32 rows=19 width=
                           (actual time=23.979..45.383 rows=940 loops=1)
         Filter: ((codepostal ~~ '75%'::text) AND (localite = 'PARIS'::text))
         Rows Removed by Filter: 54015
         Buffers: shared hit=1 read=384
         I/O Timings: read=22.326
   ->  Bitmap Heap Scan on colis (cost=6.75..682.88 rows=181 width=16)
                                 (actual time=1.305..387.239 rows=182 loops=940)
         Recheck Cond: (id_ville = villes.id_ville)
         Buffers: shared hit=91538 read=82268
         I/O Timings: read=359790.502
         ->  Bitmap Index Scan on idx_colis_ville
               (cost=0.00..6.70 rows=181 width=0)
               (actual time=0.115..0.115 rows=182 loops=940)
             Index Cond: (id_ville = villes.id_ville)
             Buffers: shared hit=2815 read=476
             I/O Timings: read=22.862
 Total runtime: 364466.458 ms
(17 lignes)
```

On constate que l'optimiseur part sur une boucle extrêmement coûteuse : 940 parcours sur `colis`, par `id_ville`. En moyenne, ces parcours durent environ 400 ms. Le résultat est vraiment très mauvais.

Il fait ce choix parce qu'il estime que la condition

```
localite ='PARIS' AND codepostal LIKE '75%'
```

va ramener 19 enregistrements. En réalité, elle en ramène 940, soit 50 fois plus, d'où un très mauvais choix. Pourquoi PostgreSQL fait-il cette erreur ?

```
marc=# EXPLAIN SELECT * FROM villes;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on villes  (cost=0.00..934.55 rows=54955 width=27)
(1 ligne)
```

17.12

```
marc=# EXPLAIN SELECT * FROM villes WHERE localite='PARIS';
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on villes  (cost=0.00..1071.94 rows=995 width=27)
   Filter: (localite = 'PARIS'::text)
(2 lignes)

marc=# EXPLAIN SELECT * FROM villes WHERE codepostal LIKE '75%';
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on villes  (cost=0.00..1071.94 rows=1042 width=27)
   Filter: (codepostal ~~ '75%'::text)
(2 lignes)


marc=# EXPLAIN SELECT * FROM villes WHERE localite='PARIS'
marc=# AND codepostal LIKE '75%';
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on villes  (cost=0.00..1209.32 rows=19 width=27)
   Filter: ((codepostal ~~ '75%'::text) AND (localite = 'PARIS'::text))
(2 lignes)
```

D'après les statistiques, villes contient 54955 enregistrements, 995 contenant PARIS (presque 2%), 1042 commençant par 75 (presque 2%).

Il y a donc 2% d'enregistrements vérifiant chaque critère (c'est normal, ils sont presque équivalents). PostgreSQL, n'ayant aucune autre information, part de l'hypothèse que les colonnes ne sont pas liées, et qu'il y a donc 2% de 2% (soit environ 0,04%) des enregistrements qui vérifient les deux.

Si on fait le calcul exact, on a donc :

```
(995/54955)*(1042/54955)*54955
```

soit 18,8 enregistrements (arrondi à 19) qui vérifient le critère. Ce qui est évidemment faux.

- Exécutez cette requête sans la dernière clause `AND codepostal LIKE '75%'`. Que constatez-vous ?

```
explain (analyze,buffers) select * from colis where id_ville in (
    select id_ville from villes where localite ='PARIS'
);
```

```
                                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Hash Semi Join  (cost=1083.86..183312.59 rows=173060 width=16)
                 (actual time=48.975..4362.348 rows=170802 loops=1)
   Hash Cond: (colis.id_ville = villes.id_ville)
   Buffers: shared hit=7 read=54435
   I/O Timings: read=1219.212
   ->  Seq Scan on colis  (cost=0.00..154053.55 rows=9999955 width=16)
                          (actual time=6.178..2228.259 rows=9999911 loops=1)
         Buffers: shared hit=2 read=54052
         I/O Timings: read=1199.307
   ->  Hash  (cost=1071.94..1071.94 rows=954 width=
             (actual time=42.676..42.676 rows=940 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 37kB
         Buffers: shared hit=2 read=383
         I/O Timings: read=19.905
         ->  Seq Scan on villes  (cost=0.00..1071.94 rows=954 width=
               (actual time=35.900..41.957 rows=940 loops=1)
               Filter: (localite = 'PARIS'::text)
               Rows Removed by Filter: 54015
               Buffers: shared hit=2 read=383
               I/O Timings: read=19.905
 Total runtime: 4375.105 ms
(17 lignes)
```

Cette fois-ci le plan est bon, et les estimations aussi.

- Quelle solution pourrait-on adopter, si on doit réellement spécifier ces deux conditions ?

On pourrait indexer sur une fonction des deux. C'est maladroit, mais malheureusement la seule solution sûre :

```
CREATE FUNCTION test_ville (ville text,codepostal text) RETURNS text
IMMUTABLE LANGUAGE SQL as $$
SELECT ville || '-' || codepostal
$$ ;

CREATE INDEX idx_test_ville ON villes (test_ville(localite , codepostal));

ANALYZE villes;

EXPLAIN (analyze,buffers) SELECT * FROM colis WHERE id_ville IN (
    SELECT id_ville
    FROM villes
    WHERE test_ville(localite,codepostal) LIKE 'PARIS-75%'
);
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------
 Hash Semi Join  (cost=1360.59..183924.46 rows=203146 width=16)
                 (actual time=46.127..3530.348 rows=170802 loops=1)
   Hash Cond: (colis.id_ville = villes.id_ville)
   Buffers: shared hit=454 read=53989
   ->  Seq Scan on colis  (cost=0.00..154054.11 rows=9999911 width=16)
                          (actual time=0.025..1297.520 rows=9999911 loops=1)
         Buffers: shared hit=66 read=53989
   ->  Hash  (cost=1346.71..1346.71 rows=1110 width=8)
             (actual time=46.024..46.024 rows=940 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 37kB
         Buffers: shared hit=385
         ->  Seq Scan on villes  (cost=0.00..1346.71 rows=1110 width=8)
                                 (actual time=37.257..45.610 rows=940 loops=1)
               Filter: (((localite || '-'::text) || codepostal) ~~
                        'PARIS-75%'::text)
               Rows Removed by Filter: 54015
               Buffers: shared hit=385
 Total runtime: 3543.838 ms
```

On constate qu'avec cette méthode il n'y a plus d'erreur d'estimation. Elle est bien sûr très pénible à utiliser, et ne doit donc être réservée qu'aux quelques rares requêtes ayant été identifiées comme ayant un comportement pathologique.

On peut aussi créer une colonne supplémentaire maintenue par un trigger, plutôt qu'un index : cela sera moins coûteux à maintenir, et permettra d'avoir la même statistique.

3.13.3 CONCLUSION

Que peut-on conclure de cet exercice ?

- que la ré-écriture est souvent la meilleure des solutions : interrogez-vous toujours sur la façon dont vous écrivez vos requêtes, plutôt que de mettre en doute PostgreSQL **a priori** ;
- que la ré-écriture de requête est souvent complexe - néanmoins, surveillez un certain nombre de choses :
  - casts implicites suspects ;
  - jointures externes inutiles ;
  - sous-requêtes imbriquées ;
  - jointures inutiles (données constantes)

4 ANALYSES ET DIAGNOSTICS

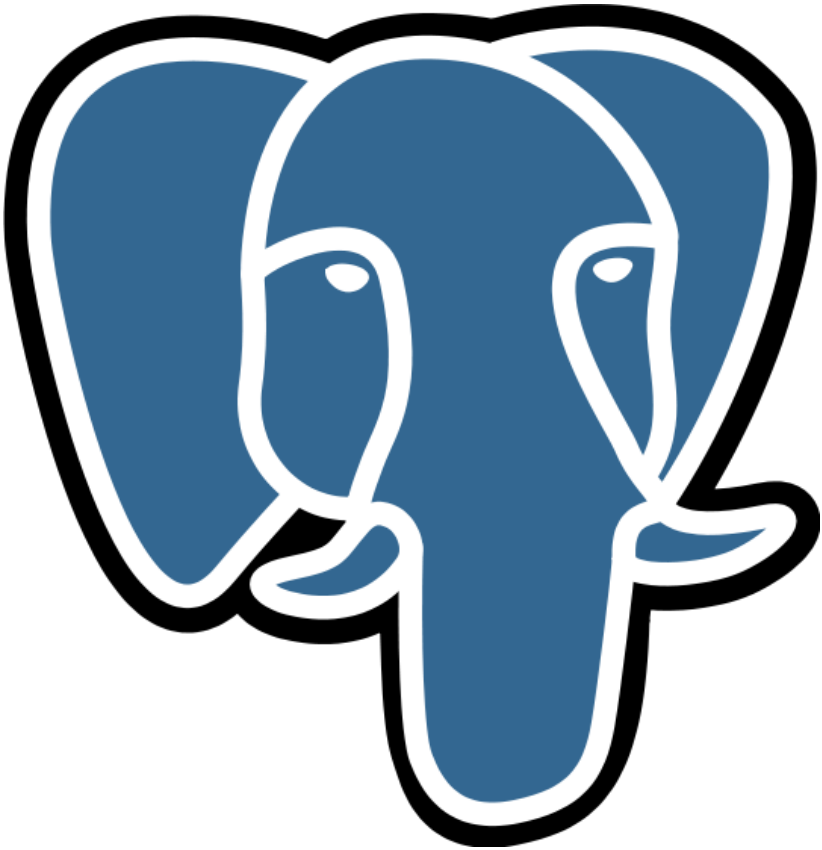


FIGURE 5: POSTGRESQL

4.1 INTRODUCTION

- Deux types de supervision
  - occasionnelle
  - automatique
- Superviser le matériel et le système
- Superviser PostgreSQL et ses statistiques

- Utiliser les bons outils

Superviser un serveur de bases de données consiste à superviser le moteur lui-même, mais aussi le système d'exploitation et le matériel. Ces deux derniers sont importants pour connaître la charge système, l'utilisation des disques ou du réseau, qui pourraient expliquer des lenteurs au niveau du moteur. PostgreSQL propose lui aussi des informations qu'il est important de surveiller pour détecter des problèmes au niveau de l'utilisation du SGBD ou de sa configuration.

Ce module a pour but de montrer comment effectuer une supervision occasionnelle (au cas où un problème surviendrait, savoir comment interpréter les informations fournies par le système et par PostgreSQL).

4.1.1 MENU

- Supervision occasionnelle système
  - Linux
  - Windows
- Supervision occasionnelle PostgreSQL
- Outils

4.2 SUPERVISION OCCASIONNELLE SOUS UNIX

- Nombreux outils
- Les tester pour les sélectionner

Il existe de nombreux outils sous Unix permettant de superviser de temps en temps le système. Cela passe par des outils comme `ps` ou `top` pour surveiller les processus à `iotop` ou `vmstat` pour les disques. Il est nécessaire de les tester, de comprendre les indicateurs et de se familiariser avec tout ou partie de ces outils afin d'être capable d'identifier rapidement un problème matériel ou logiciel.

4.2.1 UNIX - PS

- `ps` est l'outil de base pour les processus
- Exemples

- `ps aux`
- `ps -ef | grep postgres`

`ps` est l'outil le plus connu sous Unix. Il permet de récupérer la liste des processus en cours d'exécution. Les différentes options de `ps` peuvent avoir des définitions différentes en fonction du système d'exploitation (GNU/Linux, UNIX ou BSD)

Par exemple, l'option `f` active la présentation sous forme d'arborescence des processus. Cela nous donne ceci :

```
$ ps -e f | grep postgres
10149 pts/5    S      0:00  \_ postmaster
10165 ?        Ss     0:00  |   \_ postgres: checkpointer process
10166 ?        Ss     0:00  |   \_ postgres: writer process
10168 ?        Ss     0:00  |   \_ postgres: wal writer process
10169 ?        Ss     0:00  |   \_ postgres: autovacuum launcher process
10170 ?        Ss     0:00  |   \_ postgres: stats collector process
10171 ?        Ss     0:00  |   \_ postgres: bgworker: logical replication launcher
```

Les options `aux` permettent d'avoir une idée de la consommation processeur (colonne %CPU de l'exemple suivant) et mémoire (colonne %MEM) de chaque processus :

```
$ ps aux
USER  PID %CPU %MEM    VSZ   RSS STAT COMMAND
500 10149  0.0  0.0 294624 18776 S    postmaster
500 10165  0.0  0.0 294624  5120 Ss   postgres: checkpointer process
500 10166  0.0  0.0 294624  5120 Ss   postgres: writer process
500 10168  0.0  0.0 294624  8680 Ss   postgres: wal writer process
500 10169  0.0  0.0 295056  5976 Ss   postgres: autovacuum launcher process
500 10170  0.0  0.0 149796  3816 Ss   postgres: stats collector process
500 10171  0.0  0.0 294916  4004 Ss   postgres: bgworker: logical replication launcher
[...]
```

Attention aux colonnes `VSZ` et `RSS`. Elles indiquent la quantité de mémoire utilisée par chaque processus, en prenant aussi en compte la mémoire partagée lue par le processus. Il peut donc arriver que, en additionnant les valeurs de cette colonne, on arrive à une valeur bien plus importante que la mémoire physique. Ce n'est pas le cas.

Dernier exemple :

```
$ ps uf -C postgres
USER  PID %CPU %MEM    VSZ   RSS STAT COMMAND
500  9131  0.0  0.0 194156  7964 S    postmaster
500  9136  0.0  0.0 194156  1104 Ss    \_ postgres: checkpointer process
```

```
500  9137  0.0  0.0 194156  1372 Ss    \_ postgres: writer process
500  9138  0.0  0.0 194156  1104 Ss    \_ postgres: wal writer process
500  9139  0.0  0.0 194992  2360 Ss    \_ postgres: autovacuum launcher process
500  9140  0.0  0.0 153844  1140 Ss    \_ postgres: stats collector process
500  9141  0.0  0.0 194156  1372 Ss    \_ postgres: bgworker: logical replication laun
```

Il est à noter que la commande `ps` affiche un grand nombre d'informations sur le processus seulement si le paramètre `update_process_title` est activé. Un processus d'une session affiche ainsi la base, l'utilisateur et, le cas échéant, l'adresse IP de la connexion. Il affiche aussi la commande en cours d'exécution et si cette commande est bloquée en attente d'un verrou ou non.

```
$ ps -e f | grep postgres
 4563 pts/0    S      0:00  \_ postmaster
 4569 ?        Ss     0:00  |   \_ postgres: checkpointer process
 4570 ?        Ss     0:00  |   \_ postgres: writer process
 4571 ?        Ds     0:00  |   \_ postgres: wal writer process
 4572 ?        Ss     0:00  |   \_ postgres: autovacuum launcher process
 4573 ?        Ss     0:00  |   \_ postgres: stats collector process
 4574 ?        Ss     0:00  |   \_ postgres: bgworker: logical replication launcher
 4610 ?        Ss     0:00  |   \_ postgres: u1 b2 [local] idle in transaction
 4614 ?        Ss     0:00  |   \_ postgres: u2 b2 [local] DROP TABLE waiting
 4617 ?        Ss     0:00  |   \_ postgres: u3 b1 [local] INSERT
 4792 ?        Ss     0:00  |   \_ postgres: u1 b2 [local] idle
```

Dans cet exemple, quatre sessions sont ouvertes. La session 4610 n'exécute aucune requête mais est dans une transaction ouverte (c'est potentiellement un problème, à cause des verrous tenus pendant l'entièreté de la transaction et de la moindre efficacité des VACUUM). La session 4614 affiche le mot-clé `waiting` : elle est en attente d'un verrou, certainement détenu par une session en cours d'exécution d'une requête ou d'une transaction. Le `DROP TABLE` a son exécution mise en pause à cause de ce verrou non acquis. La session 4617 est en train d'exécuter un `INSERT` (la requête complète peut être obtenue avec la vue `pg_stat_activity` qui sera abordée plus loin dans ce chapitre). Enfin, la session 4792 n'exécute pas de requête et ne se trouve pas dans une transaction ouverte. `u1`, `u2` et `u3` sont les utilisateurs pris en compte pour la connexion, alors que `b1` et `b2` sont les noms des bases de données de connexion. De ce fait, la session 4614 est connectée à la base de données `b2` avec l'utilisateur `u2`.

Les processus des sessions ne sont pas les seuls à fournir quantité d'informations. Les processus de réplication et le processus d'archivage indiquent le statut et la progression de leur activité.

4.2.2 UNIX - TOP

- Principal intérêt : %CPU et %MEM
- Intérêts secondaires
  - charge CPU
  - consommation mémoire
- Autres outils
  - atop, htop, topas

top est un outil utilisant ncurses pour afficher un bandeau d'informations sur le système, la charge système, l'utilisation de la mémoire et enfin la liste des processus. Les informations affichées ressemblent beaucoup à ce que fournit la commande ps avec les options « aux ». Cependant, top rafraichit son affichage toutes les trois secondes (par défaut), ce qui permet de vérifier si le comportement détecté reste présent. top est intéressant pour connaître rapidement le processus qui consomme le plus en termes de processeur (touche P) ou de mémoire (touche M). Ces touches permettent de changer l'ordre de tri des processus. Il existe beaucoup plus de tris possibles, la sélection complète étant disponible en appuyant sur la touche F.

Parmi les autres options intéressantes, la touche c permet de basculer l'affichage du processus entre son nom seulement ou la ligne de commande complète. La touche u permet de filtrer les processus par utilisateur. Enfin, la touche 1 permet de basculer entre un affichage de la charge moyenne sur tous les processeurs et un affichage détaillé de la charge par processeur.

Exemple :

```
top - 11:45:02 up  3:40,  5 users,  load average: 0.09, 0.07, 0.10
Tasks: 183 total,   2 running, 181 sleeping,   0 stopped,   0 zombie
Cpu0  :  6.7%us,  3.7%sy,  0.0%ni, 88.3%id,  1.0%wa,  0.3%hi,  0.0%si,  0.0%st
Cpu1  :  3.3%us,  2.0%sy,  0.0%ni, 94.0%id,  0.0%wa,  0.3%hi,  0.3%si,  0.0%st
Cpu2  :  5.6%us,  3.0%sy,  0.0%ni, 91.0%id,  0.0%wa,  0.3%hi,  0.0%si,  0.0%st
Cpu3  :  2.7%us,  0.7%sy,  0.0%ni, 96.3%id,  0.0%wa,  0.3%hi,  0.0%si,  0.0%st
Mem:   3908580k total,  3755244k used,   153336k free,    50412k buffers
Swap:  2102264k total,    88236k used,  2014028k free,  1436804k cached

 PID PR  NI  VIRT  RES  SHR S %CPU %MEM COMMAND
8642 20   0  178m  29m  27m D 53.3  0.8 postgres: gui formation [local] INSERT
7894 20   0  147m 1928  508 S  0.4  0.0 postgres: stats collector process
```

```
7885 20   0  176m 7660 7064 S  0.0  0.2 /opt/postgresql-10/bin/postgres
7892 20   0  176m 1928 1320 S  0.8  0.0 postgres: wal writer process
7893 20   0  178m 3356 1220 S  0.0  0.1 postgres: autovacuum launcher process
```

Attention aux valeurs des colonnes `used` et `free`. La mémoire réellement utilisée correspond plutôt à la soustraction de used et de buffers (ce dernier étant le cache disque mémoire du noyau).

`top` n'existe pas directement sur Solaris. L'outil par défaut sur ce système est `prstat`.

4.2.3 UNIX - IOTOP

- Principal intérêt : `%IO`
- À partir du noyau 2.6.20

Utilisable à partir du noyau 2.6.20, iotop est l'équivalent de top pour la partie disque. Il affiche le nombre d'octets lus et écrits par processus, avec la commande complète. Cela permet de trouver rapidement le processus à l'origine de l'activité disque :

```
Total DISK READ:       19.79 K/s | Total DISK WRITE:   5.06 M/s
  TID  PRIO  USER DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
 1007 be/3 root    0.00 B/s  810.43 B/s  0.00 %  2.41 % [jbd2/sda3-8]
 7892 be/4 guill  14.25 K/s  229.52 K/s  0.00 %  1.93 % postgres:
                                                       wal writer process
  445 be/3 root    0.00 B/s    3.17 K/s  0.00 %  1.91 % [jbd2/sda2-8]
 8642 be/4 guill   0.00 B/s    7.08 M/s  0.00 %  0.76 % postgres:
                                                       gui formation [local] INSERT
 7891 be/4 guill   0.00 B/s  588.83 K/s  0.00 %  0.00 % postgres:
                                                       writer process
 7894 be/4 guill   0.00 B/s  151.96 K/s  0.00 %  0.00 % postgres:
                                                       stats collector process
    1 be/4 root    0.00 B/s    0.00 B/s  0.00 %  0.00 % init
```

Comme top, il s'agit d'un programme ncurses dont l'affichage est rafraichi fréquemment (toutes les secondes par défaut).

4.2.4 UNIX - VMSTAT

- Outil le plus fréquemment utilisé

- Principal intérêt
  - lecture et écriture disque
  - `iowait`
- Intérêts secondaires
  - nombre de processus en attente

`vmstat` est certainement l'outil système de supervision le plus fréquemment utilisé parmi les administrateurs de bases de données PostgreSQL. Il donne un condensé d'informations système qui permet de cibler très rapidement le problème.

Cette commande accepte plusieurs options en ligne de commande, mais il faut fournir au minimum un argument indiquant la fréquence de rafraichissement. Contrairement à top ou iotop, il envoie l'information directement sur la sortie standard, sans utiliser une interface particulière. En fait, la commande s'exécute en permanence jusqu'à son arrêt avec un Ctrl-C.

```
$ vmstat 1
procs-----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
2  0 145004 123464  51684 1272840    0    2    24    57   17  351  7  2 90  1  0
0  0 145004 119640  51684 1276368    0    0   256   384 1603 2843  3  3 86  9  0
0  0 145004 118696  51692 1276452    0    0     0    44 2214 3644 11  2 87  1  0
0  0 145004 118796  51692 1276460    0    0     0     0 1674 2904  3  2 95  0  0
1  0 145004 116596  51692 1277784    0    0     4   384 2096 3470  4  2 92  2  0
0  0 145004 109364  51708 1285608    0    0     0    84 1890 3306  5  2 90  3  0
0  0 145004 109068  51708 1285608    0    0     0     0 1658 3028  3  2 95  0  0
0  0 145004 117784  51716 1277132    0    0     0   400 1862 3138  3  2 91  4  0
1  0 145004 121016  51716 1273292    0    0     0     0 1657 2886  3  2 95  0  0
0  0 145004 121080  51716 1273292    0    0     0     0 1598 2824  3  1 96  0  0
0  0 145004 121320  51732 1273144    0    0     0   444 1779 3050  3  2 90  5  0
0  1 145004 114168  51732 1280840    0    0     0 25928 2255 3358 17  3 79  2  0
0  1 146612 106568  51296 1286520    0 1608    24 25512 2527 3767 16  5 75  5  0
0  1 146904 119364  50196 1277060    0  292    40 26748 2441 3350 16  4 78  2  0
1  0 146904 109744  50196 1286556    0    0     0 20744 3464 5883 23  4 71  3  0
1  0 146904 110836  50204 1286416    0    0     0 23448 2143 2811 16  3 78  3  0
1  0 148364 126236  46432 1273168    0 1460     0 17088 1626 3303  9  3 86  2  0
0  0 148364 126344  46432 1273164    0    0     0     0 1384 2609  3  2 95  0  0
1  0 148364 125556  46432 1273320    0    0    56  1040 1259 2465  3  2 95  0  0
0  0 148364 124676  46440 1273244    0    0     4 114720 1774 2982  4  2 84  9  0
0  0 148364 125004  46440 1273232    0    0     0     0 1715 2817  3  2 95  0  0
0  0 148364 124888  46464 1273256    0    0     4   552 2306 4014  3  2 79 16  0
```

17.12

```
0  0 148364 125060  46464 1273232    0    0     0     0 1888 3508  3  2 95  0  0
0  0 148364 124936  46464 1273220    0    0     0     4 2205 4014  4  2 94  0  0
0  0 148364 125168  46464 1273332    0    0    12   384 2151 3639  4  2 94  0  0
1  0 148364 123192  46464 1274316    0    0     0     0 2019 3662  4  2 94  0  0
^C
```

Parmi les colonnes intéressantes :

- procs r, nombre de processus en attente de temps d'exécution
- procs b, nombre de processus bloqués, ie dans un sommeil non interruptible
- free, mémoire immédiatement libre
- si, nombre de blocs lus dans le swap
- so, nombre de blocs écrits dans le swap
- buff et cache, mémoire cache du noyau Linux
- bi, nombre de blocs lus sur les disques
- bo, nombre de blocs écrits sur les disques
- us, pourcentage de la charge processeur sur une activité utilisateur
- sy, pourcentage de la charge processeur sur une activité système
- id, pourcentage d'inactivité processeur
- wa, attente d'entrées/sorties
- st, pourcentage de la charge processeur volé par un superviseur dans le cas d'une machine virtuelle

Les informations à propos des blocs manipulés (si/so et bi/bo) sont indiquées du point de vue de la mémoire. Ainsi, un bloc écrit vers le swap sort de la mémoire, d'où le `so`, comme « swap out ».

4.2.5 UNIX - IOSTAT

- Une ligne par partition
- Intéressant pour connaître la partition la plus concernée par
  - les lectures
  - ou les écritures

iostat fournit des informations plus détaillées que `vmstat`. Il est généralement utilisé quand il est intéressant de connaître le disque sur lequel sont fait les lectures et/ou écritures. Cet outil affiche des statistiques sur l'utilisation CPU et les I/O.

- L'option -d permet de n'afficher que les informations disque, l'option -c permettant de n'avoir que celles concernant le CPU.

- L'option -k affiche des valeurs en Ko/s au lieu de blocs/s. De même, -m pour des Mo/s.
- L'option -x permet d'afficher le mode étendu. Ce mode est le plus intéressant.
- Le nombre en fin de commande est l'intervalle de rafraichissement en secondes. On peut spécifier un second nombre après ce premier, qui sera le nombre de mesures à effectuer.

Comme la majorité de ces types d'outils, la première mesure retournée est une moyenne depuis le démarrage du système. Il ne faut pas la prendre en compte.

Exemple d'affichage de la commande en temps étendu :

```
$ iostat -d -x 1
Device: rrqm/s wrqm/s  r/s  w/s rkB/s wkB/s avgrq-sz avgqu-sz await svctm  %util
sda       0,00   2,67 1,33 4,67  5,33 29,33    11,56     0,02  4,00  4,00   2,40

Device: rrqm/s wrqm/s  r/s  w/s rkB/s wkB/s avgrq-sz avgqu-sz await svctm  %util
sda       0,00   0,00 0,00 0,00  0,00  0,00     0,00     0,00  0,00  0,00   0,00

Device: rrqm/s wrqm/s  r/s  w/s rkB/s wkB/s avgrq-sz avgqu-sz await svctm  %util
sda       1,33   5,00 1,33 5,33 16,00 41,33    17,20     0,04  5,20  2,40   1,60
```

Les colonnes ont les significations suivantes :

- `Device` : le périphérique
- `rrqm/s/wrms` : `read request merged per second` et `write request merged per second`, c'est-à-dire fusions d'entrées/sorties en lecture et en écriture. Cela se produit dans la file d'attente des entrées/sorties, quand des opérations sur des blocs consécutifs sont demandées... par exemple un programme qui demande l'écriture de 1 Mo de données, par bloc de 4 Ko. Le système fusionnera ces demandes d'écritures en opérations plus grosses pour le disque, afin d'être plus efficace. Un chiffre faible dans ces colonnes (comparativement à w/s et r/s) indique que le système ne peut fusionner les entrées/sorties, ce qui est signe de beaucoup d'entrées/sorties non contiguës (aléatoires). La récupération de données depuis un parcours d'index est un bon exemple...
- `r/s` et `w/s` : nombre de lectures et d'écritures par seconde. Il ne s'agit pas d'une taille en blocs, mais bien d'un nombre d'entrées/sorties par seconde. Ce nombre est le plus proche d'une limite physique, sur un disque (plus que son débit en fait) : le nombre d'entrées/sorties par seconde faisable est directement lié à la vitesse de rotation et à la performance des actuateurs des bras. Il est plus facile d'effectuer des entrées/sorties sur des cylindres proches que sur des cylindres éloignés, donc même cette valeur n'est pas parfaitement fiable. La somme de `r/s` et `w/s` devrait

être assez proche des capacités du disque. De l'ordre de 150 entrées/sorties par seconde pour un disque 7200 RPMS (SATA), 200 pour un 10000 RPMS, 300 pour un 15000 RPMS.

- `rkB/s` et `wkB/s` : les débits en lecture et écriture. Ils peuvent être faibles, avec un disque pourtant à 100%.
- `avgrq-sz` : taille moyenne d'une requête. Plus elle est proche de 1, plus les opérations sont aléatoires. Sur un SGBD, c'est un mauvais signe : dans l'idéal, soit les opérations sont séquentielles, soit elles se font en cache.
- `avgqu-sz` : taille moyenne de la file d'attente des entrées/sorties. Si ce chiffre est élevé, cela signifie que les entrées/sorties s'accumulent. Ce n'est pas forcément anormal, mais cela entrainera des latences, surtout avec des schedulers comme deadline. Si une grosse écriture est en cours, ce n'est pas choquant (voir le second exemple).
- `await` : temps moyen attendu par une entrée/sortie avant d'être totalement traitée. C'est le temps moyen écoulé, vu d'un programme, entre la soumission d'une entrée/sortie et la récupération des données. C'est un bon indicateur du ressenti des utilisateurs : c'est le temps moyen qu'ils ressentiront pour qu'une entrée/sortie se fasse (donc vraisemblablement une lecture, vu que les écritures sont asynchrones, vues par un utilisateur de PostgreSQL).
- `svctm` : temps moyen du traitement d'une entrée/sortie par le disque. Contrairement à await, on ne prend pas en compte le temps passé en file d'attente. C'est donc un indicateur de l'efficacité de traitement des entrées/sorties par le disque (il sera d'autant plus efficace qu'elles seront proches sur le disque).
- `%util` : le pourcentage d'utilisation. Il est calculé suivant cette formule :

```
(r/s+w/s) × (svctm/1000) × 100
```

(nombre d'entrées/sorties par seconde, multiplié par le temps de traitement d'une entrée/sortie en seconde, et multiplié par 100). Attention, à cause des erreurs d'arrondis, il est approximatif et dépasse quelquefois 100.

Exemple d'affichage de la commande lors d'une copie de 700 Mo :

```
$ iostat -d -x 1
Device: rrqm/s wrqm/s r/s   w/s  rkB/s  wkB/s avgrq-sz avgqu-sz await svctm %util
sda     60,7  1341,3 156,7  24,0 17534,7 2100,0 217,4 34,4      124,5  5,5   99,9

Device: rrqm/s wrqm/s r/s  w/s  rkB/s  wkB/s  avgrq-sz avgqu-sz await svctm %util
sda     20,7  3095,3 38,7 117,3 4357,3 12590,7  217,3  126,8    762,4  6,4  100,0

Device: rrqm/s wrqm/s r/s   w/s  rkB/s  wkB/s avgrq-sz avgqu-sz await svctm %util
```

```
sda      30,7    803,3 63,3   73,3 8028,0 6082,7    206,5   104,9     624,1   7,3   100,0

Device: rrqm/s wrqm/s r/s    w/s   rkB/s   wkB/s avgrq-sz avgqu-sz await svctm %util
sda      55,3   4203,0 106,0 29,7 12857,3 6477,3   285,0    59,1     504,0   7,4   100,0

Device: rrqm/s wrqm/s r/s    w/s   rkB/s   wkB/s avgrq-sz avgqu-sz await svctm %util
sda      28,3   2692,3 56,0  32,7 7046,7 14286,7   481,2    54,6     761,7 11,3   100,0
```

4.2.6 UNIX - SYSSTAT

- Outil le plus ancien
- Récupère des statistiques de façon périodique
- Permet de lire les statistiques datant de plusieurs heures, jours, etc.

`sysstat` est un paquet logiciel comprenant de nombreux outils permettant de récupérer un grand nombre d'informations système, notamment pour le système disque. Il est capable d'enregistrer ces informations dans des fichiers binaires, qu'il est possible de décoder par la suite.

Sur Debian/Ubuntu, une fois `sysstat` installé, il faut configurer son exécution automatique pour récupérer des statistiques périodiques. Le plus simple est d'utiliser dpkg-reconfigure :

```
# dpkg-reconfigure sysstat
```

Ce paquet dispose notamment de l'outil pidstat. Ce dernier récupère les informations système spécifiques à un processus (et en option à ses fils). Pour cela, il faut disposer d'un noyau 2.6.20 ou supérieur et de la version 7.1.5 de `sysstat`. Le noyau doit avoir la comptabilité des informations par processus, à savoir les options suivantes :

- `CONFIG_TASKSTATS=y`
- `CONFIG_TASK_DELAY_ACCT=y`
- `CONFIG_TASK_XACCT=y`
- `CONFIG_TASK_IO_ACCOUNTING=y`

Le tutoriel[30] est bien écrit, sa lecture est conseillée.

Pour plus di'information, consultez le site officiel[31] .

---

[30] http://pagesperso-orange.fr/sebastien.godard/tutorial.html
[31] http://pagesperso-orange.fr/sebastien.godard/index.html

4.2.7 UNIX - FREE

- Principal intérêt : connaître la répartition de la mémoire

Cette commande indique la mémoire totale, la mémoire disponible, celle utilisée pour le cache, etc.

```
# free -m
             total       used       free     shared    buffers     cached
Mem:         64567      64251        315          0        384      61819
-/+ buffers/cache:       2047      62519
Swap:         3812          0       3811
```

Ce serveur dispose de 64 Go de mémoire d'après la colonne totale. Le système et les applications utilisent un peu moins de 64 Go de mémoire. En fait, seuls 315 Mo ne sont pas utilisés. Le système utilise 384 Mo de cette mémoire pour ses informations internes (colonne buffers) et un peu moins de 62 Go pour son cache disque (colonne cache). Autrement dit, les applications n'utilisent que 2 Go de mémoire.

Si on veut aller plus loin, la ligne `-/+ buffers/cache` fournit des informations très intéressantes également. Elle nous montre que seuls 2 Go de mémoire sont réellement utilisés (colonne used). La colonne free nous montre que 62 Go de mémoire sont disponibles pour de prochaines allocations de mémoire. Cette dernière information est simplement la somme des colonnes `free`, `buffers` et `cached` de la ligne `Mem`.

`vmstat` fournit à peu près les mêmes informations avec la commande suivante :

```
# vmstat -s -S M | grep mem
        64567 M total memory
        64318 M used memory
        16630 M active memory
        46327 M inactive memory
          249 M free memory
          386 M buffer memory
```

Vous trouverez plus d'informations sur le site officiel[32] .

4.2.8 UNIX - IPCS, IPCRM

- Gestion des sémaphores
- ipcs pour avoir la liste

[32] http://momjian.us/main/blogs/pgblog/2012.html#May_2_2012

- ipcrm pour en supprimer

La mémoire partagée permet aux processus d'accéder à des structures et à des données communes. Les informations sont placées dans des segments de mémoire partagée. C'est la méthode la plus rapide disponible pour la communication interprocessus, car elle ne nécessite aucun appel aux fonctions du noyau pour passer les données. Les données ne sont même pas copiées entre processus. PostgreSQL utilise la mémoire partagée pour différents types de données, le plus intéressant (pour les performances) étant son cache disque. Pour voir le paramétrage de la mémoire partagée, exécutez la commande suivante :

```
$ ipcs -lm

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

Pour connaître l'utilisation de la mémoire partagée, utilisez la commande ipcs sans options :

```
$ ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x0052e2c1 196608     postgres  600        6656180224 75

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x0052e2c1 3735552    postgres  600        17
0x0052e2c2 3768321    postgres  600        17
0x0052e2c3 3801090    postgres  600        17
0x0052e2c4 3833859    postgres  600        17
0x0052e2c5 3866628    postgres  600        17
0x0052e2c6 3899397    postgres  600        17
0x0052e2c7 3932166    postgres  600        17
0x0052e2c8 3964935    postgres  600        17
0x0052e2c9 3997704    postgres  600        17
0x0052e2ca 4030473    postgres  600        17
0x0052e2cb 4063242    postgres  600        17
```

17.12

```
0x0052e2cc 4096011    postgres  600        17
0x0052e2cd 4128780    postgres  600        17
0x0052e2ce 4161549    postgres  600        17
0x0052e2cf 4194318    postgres  600        17
0x0052e2d0 4227087    postgres  600        17
0x0052e2d1 4259856    postgres  600        17
0x0052e2d2 4292625    postgres  600        17
0x0052e2d3 4325394    postgres  600        17

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

En cas de doute, il est possible d'identifier le segment de mémoire partagé utilisée par PostgreSQL en consultant la dernière valeur de la dernière ligne du fichier `$PGDATA/postmaster.pid` :

```
$ cat $PGDATA/postmaster.pid
3582
/var/lib/postgresql/10/main
1358945877
5432
/tmp
localhost
  5492001    196608
ready
```

Dans les anciennes versions, après un crash du serveur de bases de données, il était parfois nécessaire de supprimer manuellement le segment de mémoire partagée. Cela se fait directement avec la commande ipcrm ou plus simplement en redémarrant le serveur. Les versions récentes surveillent, au redémarrage, l'existence du segment de mémoire et le suppriment le cas échéant.

4.3 SUPERVISION OCCASIONNELLE SOUS WINDOWS

- Là aussi, nombreux outils
- Les tester pour les sélectionner

Bien qu'il y ait moins d'outils en ligne de commande, il existe plus d'outils graphiques, directement utilisables. Un outil très intéressant est même livré avec le système : les outils performances.

4.3.1 WINDOWS - TASKLIST

- `ps` et `grep` en une commande

tasklist est le seul outil en ligne de commande discutée ici.

Il permet de récupérer la liste des processus en cours d'exécution. Les colonnes affichées sont modifiables par des options en ligne de commande et les processus sont filtrables (option /fi).

Le format de sortie est sélectionnable avec l'option /fo.

La commande suivante permet de ne récupérer que les processus postgres.exe :

```
tasklist /v /fi "imagename eq postgres.exe"
```

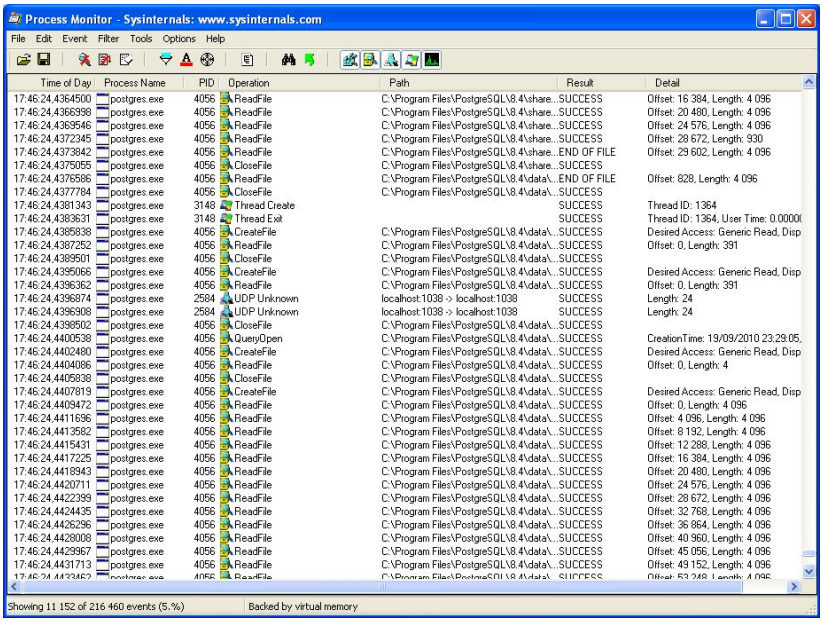
Voir (le site officiel)[https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/tasklist] pour plus de détails.

4.3.2 WINDOWS - PROCESS MONITOR

- Surveillance des processus
- Filtres
- Récupération de la ligne de commande, identificateur de session et utilisateur
- Site officiel[33]

Process Monitor permet de lister les appels système des processus, comme le montre la copie d'écran ci-dessous :

[33] https://docs.microsoft.com/en-us/sysinternals/downloads/procmon



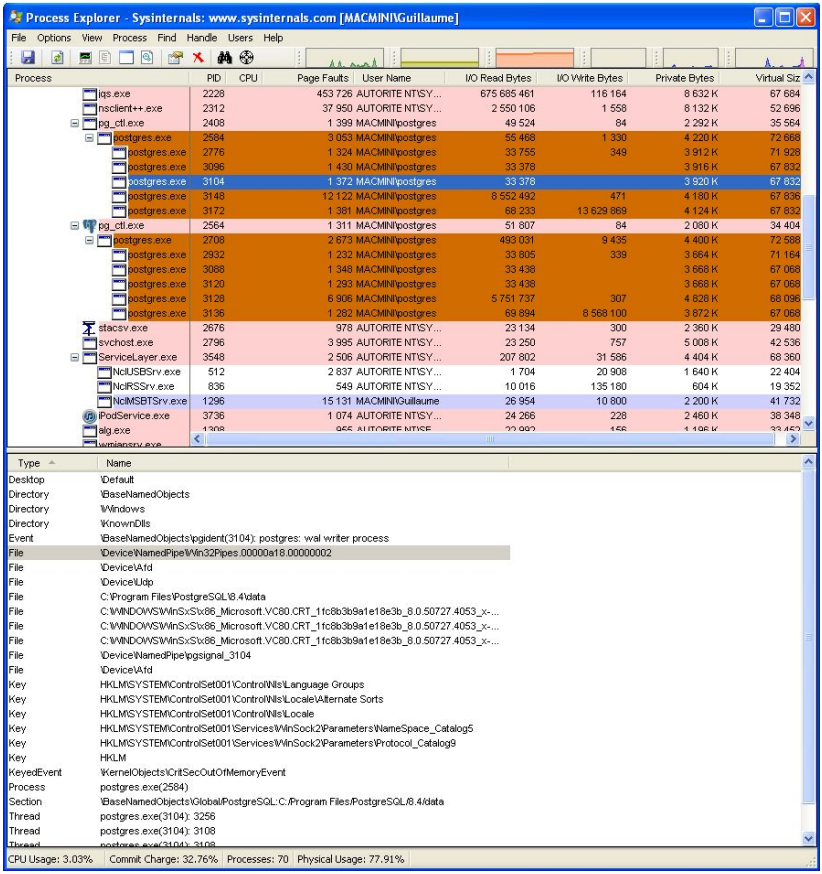
Il affiche en temps réel l'utilisation du système de fichiers, de la base de registre et de l'activité des processus. Il combine les fonctionnalités de deux anciens outils, FileMon et Regmon, tout en ajoutant un grand nombre de fonctionnalités (filtrage, propriétés des événements et des processus, etc) . Process Monitor permet d'afficher les accès aux fichiers (DLL et autres) par processus.

4.3.3 WINDOWS - PROCESS EXPLORER

- Semblable à top
- Site officiel[34]

Ce logiciel est un outil de supervision avancée sur l'activité du système et plus précisément des processus. Il permet de filtrer les processus affichés, de les trier, le tout avec une interface graphique facile à utiliser.

[34] https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer



La copie d'écran ci-dessus montre un système Windows avec deux instances PostgreSQL démarrées. L'utilisation des disques et de la mémoire est visible directement. Quand on demande les propriétés d'un processus, on dispose d'un dialogue avec plusieurs onglets, dont trois essentiels :

- le premier, « Image », donne des informations de base sur le processus :

- le deuxième, « Performances » fournit des informations textuelles sur les performances :

- le troisième affiche quelques graphes :

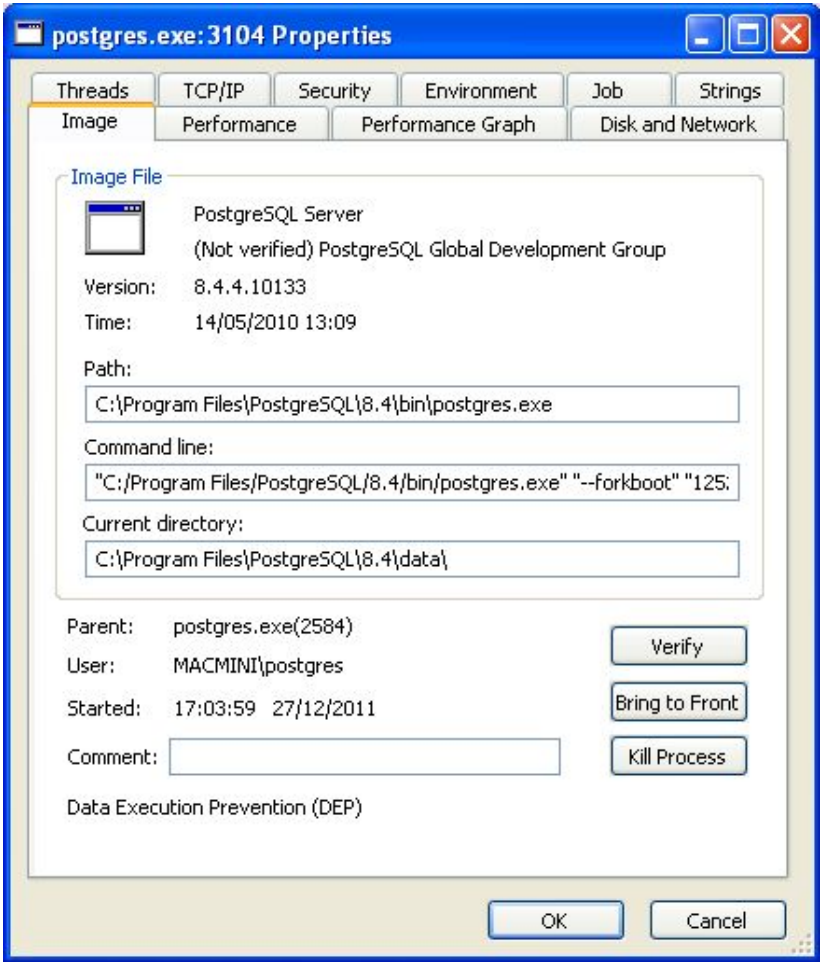


FIGURE 6: PROCESS EXPLORER - IMAGE

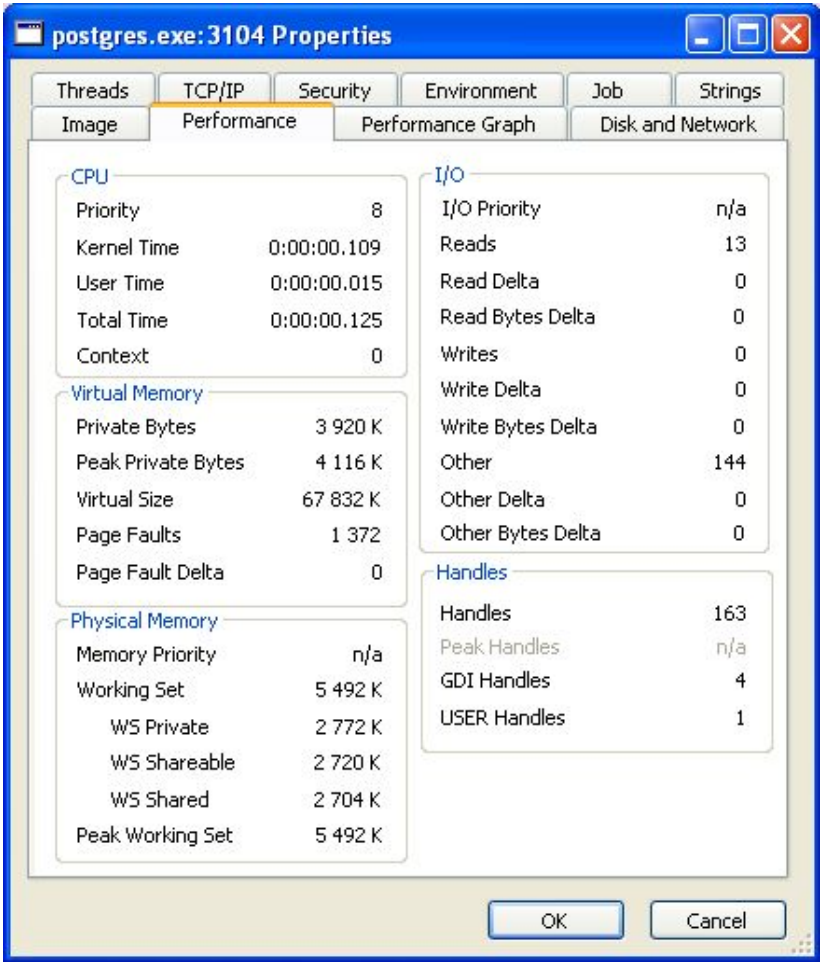


FIGURE 7: PROCESS EXPLORER- PERFORMANCES

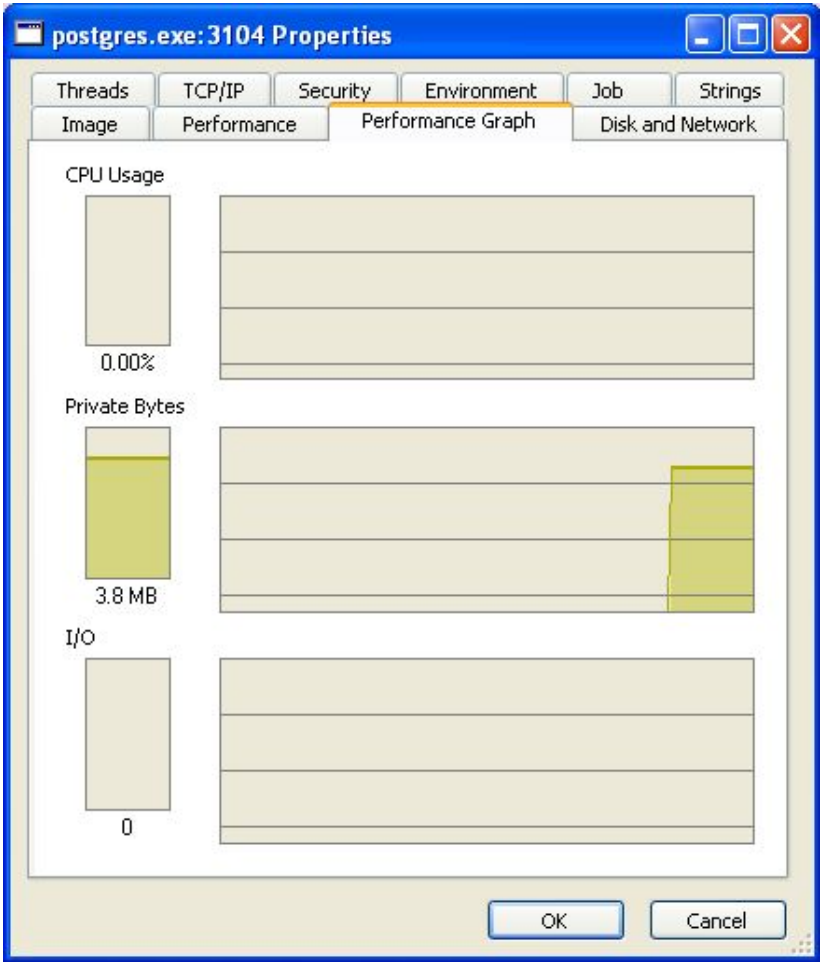


FIGURE 8: PROCESS EXPLORER - PERFORMANCE GRAPH

Il existe aussi sur cet outil un bouton System Information. Ce dernier affiche une fenêtre à quatre onglets, avec des graphes basiques mais intéressants sur les performances du système.

4.3.4 WINDOWS - OUTILS PERFORMANCES

- Semblable à `sysstat`
- Mais avec plus d'informations
- Et des graphes immédiats

Cet outil permet d'aller plus loin en termes de graphes. Il crée des graphes sur toutes les données disponibles, fournies par le système. Cela rend la recherche des performances plus simples dans un premier temps sur un système Windows.

4.4 SUPERVISION OCCASIONNELLE DE POSTGRESQL

- Plusieurs aspects à surveiller :
  - activité de la base
  - activité sur les tables
  - requêtes SQL
  - écritures

Dans le cadre de la supervision occasionnelle de PostgreSQL orientée performance, le SGBD permet d'accéder à différentes informations par différents canaux. L'objectif de cette partie est de décrire les informations les plus intéressantes à récupérer.

4.4.1 SURVEILLER L'ACTIVITÉ DE LA BASE

- qui est connecté ?
- qui fait quoi ?
- qui est bloqué ?
- qui bloque les autres ?
- comment arrêter une requête ?

4.4.2 VUE PG_STAT_ACTIVITY

- Liste des processus
  - sessions (backends)
  - processus en tâche de fond (10+)
- Change de définition en 9.2, puis en 9.6
- Requête en cours/dernière exécutée (9.2+)
- `IDLE IN TRANSACTION`
- Sessions en attente de verrou

Cette vue donne la liste des processus du serveur PostgreSQL (une ligne par session et processus en tâche de fond). Avant la version 10, il n'y avait que les sessions.

Son format a légèrement changé avec PostgreSQL 9.2 :

- `procpid` devient `pid` (pour être homogène avec d'autres vues comme `pg_locks` par exemple)
- `current_query` est séparé en `state` et `query`
  - `current_query` représentait soit la requête en cours, soit `IDLE` soit `<IDLE> in transaction`. On n'avait donc aucun moyen de connaître la dernière requête d'une session `IDLE In Transaction`, et donc aucun moyen de trouver simplement le code responsable
  - `query` contient maintenant la requête en cours si `state` est `active`, et la dernière requête effectuée si `state` vaut `idle`, `idle in transaction` ou `idle in transaction (aborted)` (transaction annulée à cause d'une erreur SQL, mais n'ayant pas reçu de rollback explicite).
- la colonne `state_change` est ajoutée et indique la date et l'heure du dernier changement d'état (utile surtout dans le cas d'une session en state `idle`, pour connaître l'heure de la dernière requête exécutée, et l'heure de fin de cette dernière requête)

Deux nouvelles colonnes ont été ajoutées pour la version 9.4 :

- `backend_xid` : identifiant de transaction courant pour cette session
- `backend_xmin` : identifiant de transaction représentant la vision de la base pour cette session

En version 9.6, la colonne `waiting` est remplacée par les colonnes `wait_event_type` et `wait_event` pour fournir plus de détails sur la nature du blocage (ou de l'attente) :

- `wait_event_type` : type d'événement en attente
- `wait_event` : nom de l'événement en attente

Voir le tableau des événements d'attente[35] pour plus de détails.

[35] http://docs.postgresql.fr/10/monitoring-stats.html#wait-event-table

La version 10 ajoute une colonne supplémentaire, `backend_type`, indiquant le type de processus. Il en existe plusieurs, par exemple `background worker`, `background writer`, `autovacuum launcher`, `client backend`, `walsender`, `checkpointer`, `walwriter`.

Les autres champs contiennent :

- `datid` : l'`OID` de la base à laquelle la session est connectée.
- `datname` : le nom de la base associée à cet OID.
- `pid` : le numéro du processus du *backend*, c'est à dire du processus PostgreSQL chargé de discuter avec le client.
- `usesysid` : l'`OID` de l'utilisateur connecté.
- `usename` : le nom de l'utilisateur associé à cet OID.
- `application_name` : un nom facultatif renseigné par l'application cliente (par `SET application_name TO 'mon_nom'`). Apparu en 9.0.
- `client_addr` : l'adresse IP du client connecté (ou `NULL` si connexion sur socket Unix). Apparu en 9.0.
- `client_hostname` : le nom associé à cette IP. Il n'est renseigné que si `log_hostname` est à `on`. Attention, ce paramètre peut fortement ralentir la connexion (résolution DNS). Apparu en 9.1.
- `client_port` : le numéro de port sur lequel le client est connecté, toujours s'il s'agit d'une connexion IP. Apparu en 9.0.
- `backend_start` : le timestamp de l'établissement de la session.
- `xact_start` : le timestamp de début de la transaction.
- `query_start` : le timestamp de début de la requête en cours/dernière requête suivant la version de la vue.

Certains champs de cette vue ne sont renseignés que si `track_activities` est à `on` (valeur par défaut).

4.4.3 ARRÊTER UNE REQUÊTE OU UNE SESSION

- Annuler une requête
  - `pg_cancel_backend (pid int)`
- Fermer une connexion
  - `pg_terminate_backend(pid int)`

Ces deux fonctions sont plus souvent utilisées.

La première permet d'annuler une requête en cours d'exécution. Elle requiert un argument, à savoir le numéro du PID du processus postgres exécutant cette requête. Généralement, l'annulation est immédiate. Voici un exemple de son utilisation.

17.12

L'utilisateur, connecté au processus de PID 10901 comme l'indique la fonction pg_backend_pid, exécute une très grosse insertion :

```
b1=# SELECT pg_backend_pid();
 pg_backend_pid
----------------
          10901
(1 row)

b1=# INSERT INTO t4 SELECT i, 'Ligne '||i
FROM generate_series(2000001, 3000000) AS i;
```

Supposons qu'on veuille annuler l'exécution de cette requête. Voici comment faire à partir d'une autre connexion :

```
b1=# SELECT pg_cancel_backend(10901);
 pg_cancel_backend
-------------------
 t
(1 row)
```

L'utilisateur qui a lancé la requête d'insertion verra ce message apparaître :

```
ERROR:  canceling statement due to user request
```

Si la requête du INSERT faisait partie d'une transaction, la transaction elle-même sera en ROLLBACK étant donné que cette requête est en erreur. À noter cependant qu'il n'est pas possible d'annuler une transaction qui n'exécute rien à ce moment.

Il est possible d'aller plus loin en supprimant la connexion d'un utilisateur. Cela se fait avec la fonction pg_terminate_backend :

```
b1=# SELECT pid, datname, usename, application_name,state FROM pg_stat_activity WHERE backend_type
 procpid | datname |  usename  | application_name | state
---------+---------+-----------+------------------+--------
   13267 | b1      | u1        | psql             | idle
   10901 | b1      | guillaume | psql             | active
(2 rows)

b1=# SELECT pg_terminate_backend(13267);
 pg_terminate_backend
----------------------
 t
(1 row)

b1=# SELECT pid, datname, usename, application_name, state FROM pg_stat_activity WHERE backend_typ
```

```
 procpid | datname |  usename  | application_name | state
---------+---------+-----------+------------------+--------
   10901 | b1      | guillaume | psql             | active
(1 row)
```

Elle utilise aussi le PID du processus à déconnecter.

Ces deux fonctions sont utilisables par les superutilisateurs et par les utilisateurs cherchant à annuler une requête ou supprimer une session qu'ils auraient ouvertes.

4.4.4 VUE PG_LOCKS

- Visualisation des verrous en place
- Tous types de verrous sur objets
- Complexe à interpréter
  - verrous sur enregistrements pas directement visibles
  - voir l'article détaillé[36] sur la base de connaissance Dalibo.

C'est une vue globale à l'instance. Voici la signification de ses colonnes :

- `locktype` : type de verrou, les plus fréquents étant `relation` (table ou index), `transactionid` (transaction), `virtualxid` (transaction virtuelle, utilisée tant qu'une transaction n'a pas eu à modifier de données, donc à stocker des identifiants de transaction dans des enregistrements).
- `database` : la base dans laquelle ce verrou est pris.
- `relation` : si locktype vaut `relation` (ou `page` ou `tuple`), l'`OID` de la relation cible.
- `page` : le numéro de la page dans une relation (quand verrou de type `page` ou `tuple`) cible.
- `tuple` : le numéro de l'enregistrement, (quand verrou de type `tuple`) cible.
- `virtualxid` : le numéro de la transaction virtuelle (quand verrou de type `virtualxid`) cible.
- `transactionid` : le numéro de la transaction cible.
- `classid` : le numéro d'`OID` de la classe de l'objet verrouillé (autre que relation) dans `pg_class`. Indique le catalogue système, donc le type d'objet, concerné. Aussi utilisé pour les advisory locks.
- `objid` : l'`OID` de l'objet dans le catalogue système pointé par classid.
- `objsubid` : l'ID de la colonne de l'objet objid concerné par le verrou.
- `virtualtransaction` : le numéro de transaction virtuelle possédant le verrou (ou tentant de l'acquérir si granted est à `f`).

[36] https://kb.dalibo.com/verrouillage%3E

- `pid` : le pid de la session possédant le verrou.
- `mode` : le niveau de verrouillage demandé.
- `granted` : acquis ou non (donc en attente).
- `fastpath` : information utilisée pour le débuggage surtout. Fastpath est le mécanisme d'acquisition des verrous les plus faibles.

La plupart des verrous sont de type relation, transactionid ou virtualxid. Une transaction qui démarre prend un verrou virtualxid sur son propre virtualxid. Elle acquiert des verrous faibles (`ACCESS SHARE`) sur tous les objets sur lesquels elle fait des `SELECT`, afin de garantir que leur structure n'est pas modifiée sur la durée de la transaction. Dès qu'une modification doit être faite, la transaction acquiert un verrou exclusif sur le numéro de transaction qui vient de lui être affecté. Tout objet modifié (table) sera verrouillé avec `ROW EXCLUSIVE`, afin d'éviter les `CREATE INDEX` non concurrents, et empêcher aussi les verrouillage manuels de la table en entier (`SHARE ROW EXCLUSIVE`).

4.4.5 TRACE DES ATTENTES DE VERROUS

- Message dans les traces
  - uniquement pour les attentes de plus d'une seconde
  - paramètre `log_lock_waits` à `on`
  - rapport pgBadger disponible

Le paramètre `log_lock_waits` permet d'activer la trace des attentes de verrous. Toutes les attentes ne sont pas tracées, seules les attentes qui dépassent le seuil indiqué par le paramètre `deadlock_timeout`. Ce paramètre indique à partir de quand PostgreSQL doit résoudre les deadlocks potentiels entre plusieurs transactions.

Comme il s'agit d'une opération assez lourde, elle n'est pas déclenchée lorsqu'une session est mise en attente, mais lorsque l'attente dure plus d'une seconde, si l'on reste sur la valeur par défaut du paramètre. En complément de cela, PostgreSQL peut tracer les verrous qui nécessite une attente et qui ont déclenché le lancement du gestionnaire de deadlock. Une nouvelle trace est émise lorsque la session a obtenu son verrou.

À chaque fois qu'une requête est mise en attente parce qu'une autre transaction détient un verrou, un message tel que le suivant apparaît dans les logs de PostgreSQL :

```
LOG:  process 2103 still waiting for ShareLock on transaction 29481
      after 1039.503 ms
DETAIL:  Process holding the lock: 2127. Wait queue: 2103.
CONTEXT:  while locking tuple (1,3) in relation "clients"
STATEMENT:  SELECT * FROM clients WHERE client_id = 100 FOR UPDATE;
```

Lorsque le client obtient le verrou qu'il attendait, le message suivant apparaît dans les logs :

```
LOG:  process 2103 acquired ShareLock on transaction 29481 after 8899.556 ms
CONTEXT:  while locking tuple (1,3) in relation "clients"
STATEMENT:  SELECT * FROM clients WHERE client_id = 100 FOR UPDATE;
```

L'inconvénient de cette méthode est qu'il n'y a aucune trace de la session qui a mis une ou plusieurs autres sessions en attente. Si l'on veut obtenir le détail de ce que réalise cette session, il est nécessaire d'activer la trace des requêtes SQL.

4.4.6 VUE PG_STAT_DATABASE

- Des informations globales à chaque base
- Nombre de sessions
- Nombre de transactions validées/annulées
- Nombre d'accès blocs
- Nombre d'accès enregistrements
- Taille et nombre de fichiers temporaires
- Temps d'entrées/sorties

Voici la signification des différentes colonnes :

- `datid`/`datname` : l'`OID` et le nom de la base de données.
- `numbackends` : le nombre de sessions en cours.
- `xact_commit` : le nombre de transactions ayant terminé avec commit sur cette base.
- `xact_rollback` : le nombre de transactions ayant terminé avec rollback sur cette base.
- `blks_read` : le nombre de blocs demandés au système d'exploitation.
- `blks_hit` : le nombre de blocs trouvés dans la cache de PostgreSQL.
- `tup_returned` : le nombre d'enregistrements réellement retournés par les accès aux tables.
- `tup_fetched` : le nombre d'enregistrements interrogés par les accès aux tables (ces deux compteurs seront explicités dans la vue sur les index).
- `tup_inserted` : le nombre d'enregistrements insérés en base.
- `tup_updated` : le nombre d'enregistrements mis à jour en base.
- `tup_deleted` : le nombre d'enregistrements supprimés en base.
- `conflicts` : le nombre de conflits de réplication (sur un esclave). Ajouté en 9.1.
- `temp_files` : le nombre de fichiers temporaires (utilisés pour le tri) créés par cette base depuis son démarrage. Ajouté en 9.2.

- `temp_bytes` : le nombre d'octets correspondant à ces fichiers temporaires. Cela permet de trouver les bases effectuant beaucoup de tris sur disque. Ajouté en 9.2.
- `deadlocks` : le nombre de deadlocks (interblocages). Ajouté en 9.2.
- `blk_read_time` et `blk_write_time` : le temps passé à faire des lectures et des écritures vers le disque. Il faut que `track_io_timing` soit à `on`, ce qui n'est pas la valeur par défaut. Ajoutés en 9.2.
- `stats_reset` : la date de dernière remise à zéro des compteurs de cette vue. Ajouté en 9.1.

4.4.7 TRACE DES CONNEXIONS

- Message dans les traces
  - à chaque connexion/déconnexion
  - paramètre `log_connections` et `log_disconnections`
  - rapport pgBadger disponible

Les paramètres `log_connections` et `log_disconnections` permettent d'activer les traces de toutes les connexions réalisées sur l'instance.

La connexion d'un client, lorsque sa connexion est acceptée, entraîne la trace suivante :

```
LOG:  connection received: host=::1 port=45837
LOG:  connection authorized: user=workshop database=workshop
```

Si la connexion est rejetée, l'événement est également tracé :

```
LOG:  connection received: host=[local]
FATAL:  pg_hba.conf rejects connection for host "[local]", user "postgres",
        database "postgres", SSL off
```

Une déconnexion entraîne la production d'une trace de la forme suivante :

```
LOG:  disconnection: session time: 0:00:00.003 user=workshop database=workshop
      host=::1 port=45837
```

Ces traces peuvent être exploitées par des outils comme pgBadger. Toutefois, pgBadger n'ayant pas accès à l'instance observée, il ne sera pas possible de déterminer quels sont les utilisateurs qui sont connectés de manière permanente à la base de données. Cela permet néanmoins de déterminer le volume de connexions réalisées sur la base de données, par exemple pour évaluer si un pooler de connexion serait intéressant.

4.4.8 SURVEILLER L'ACTIVITÉ SUR LES TABLES

- Quelle taille font mes objets ?
- Quel est leur taux de fragmentation ?
- Comment sont-ils accédés ?

4.4.9 OBTENIR LA TAILLE DES OBJETS (1/2)

- Table
  - `pg_relation_size(relation name)`
  - `pg_total_relation_size(relation name)`
  - `pg_table_size(table name)`
  - `pg_indexes_size(table name)`

Pour les relations, il existe quatre fonctions dont le résultat diffère :

- `pg_relation_size` donne la taille de la relation (donc uniquement la partie `HEAP` ou la partie `TOAST` d'une table, mais pas les deux... les index et les fichiers d'extension `_fsm` et `_vm` ne sont pas pris en compte).
- `pg_total_relation_size` donne la taille complète de la table (partie `HEAP`, partie `TOAST`, index, fichiers supplémentaires)
- `pg_table_size` ne donne que la taille de la table (partie `HEAP`, partie `TOAST`, fichiers supplémentaires mais pas les index)
- `pg_indexes_size` ne donne que la taille des index d'une table (partie `HEAP`, partie `TOAST` mais pas la table)

Les deux dernières sont disponibles depuis la version 9.0.

Voici un court exemple montrant les quatres fonctions :

```
b1=# CREATE TABLE t4(c1 serial PRIMARY KEY, c2 text);
NOTICE:  CREATE TABLE will create implicit sequence "t4_c1_seq" for serial
         column "t4.c1"
NOTICE:  ALTER TABLE / ADD PRIMARY KEY will create implicit index "t4_pkey"
         for table "t4"
CREATE TABLE
b1=# INSERT INTO t4 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) AS i;
INSERT 0 1000000
b1=# CREATE INDEX t4c2 ON t4(c2);
CREATE INDEX
b1=# \x
b1=# SELECT pg_relation_size('t4'), pg_total_relation_size('t4'),
pg_table_size('t4'), pg_indexes_size('t4');
```

```
 pg_relation_size | pg_total_relation_size | pg_table_size | pg_indexes_size
------------------+------------------------+---------------+-----------------
         51396608 |              105488384 |      51437568 |        54050816
(1 row)
```

Là aussi, l'argument requis est soit le nom de la table (attention au schéma) soit son `OID`. La fonction renvoie un entier correspondant au nombre d'octets de l'objet.

4.4.10 OBTENIR LA TAILLE DES OBJETS (2/2)

- Pour un affichage plus lisible
  - `pg_size_pretty(size bigint)`

Les fonctions vues précédemment sont difficile à lire car exprimées en octets. PostgreSQL peut afficher cette information avec une unité facilement compréhensible par un humain. Cette fonction s'appelle `pg_size_pretty` et renvoie du texte :

```
b1=# SELECT pg_size_pretty(pg_table_size('t4'));
 pg_size_pretty
----------------
 49 MB
(1 row)

b1=# SELECT datname, pg_size_pretty(pg_database_size(oid)) FROM pg_database;
  datname  | pg_size_pretty
-----------+----------------
 template1 | 6049 kB
 template0 | 6049 kB
 postgres  | 564 MB
 b1        | 107 MB
(4 rows)
```

4.4.11 MESURER LA FRAGMENTATION DES OBJETS

- Fragmentation induite par MVCC
  - touche tables et index
- Requêtes pour estimer la fragmentation :
  - Dépôt github[37]
  - supervision avec `check_pgactivity`

[37] https://github.com/ioguix/pgsql-bloat-estimation

- Mesure précise de la fragmentation :
  - extension `pgstattuple`

La fragmentation des tables et index est inhérente à l'implémentation de MVCC de PostgreSQL. Elle est contenue grâce à `VACUUM` et surtout à autovacuum. Cependant, certaines utilisations de la base de données peuvent entraîner une fragmentation plus importante que prévue (transaction ouverte pendant plusieurs jours, purge massive, etc.) et peuvent entraîner des ralentissements de la base de données. Il est donc nécessaire de pouvoir détecter les cas où la base présente une fragmentation trop élevée.

Les requêtes données dans le dépôt de *ioguix* permettent d'évaluer indépendamment la fragmentation des tables et des index. Elles sont utilisées dans la sonde check_pgactivity, qui permet d'être alerté automatiquement dès lors qu'une ou plusieurs tables/index présentent une fragmentation trop forte.

Les requêtes proposées donnent seulement une estimation de la fragmentation d'une table. Dans certains cas, elle n'est pas très précise. Pour mesurer très précisément la fragmentation d'une table, il est possible d'utiliser l'extension `pgstattuple`.

On génère d'abord de la fragmentation :

```
pgstattuple=# CREATE EXTENSION pgstattuple;
CREATE EXTENSION
pgstattuple=# CREATE TABLE demo_bloat (i integer);
CREATE TABLE
postgres=# ALTER TABLE demo_bloat SET (autovacuum_enabled=false);
ALTER TABLE
pgstattuple=# INSERT INTO demo_bloat SELECT i FROM generate_series(1, 10000) i;
INSERT 0 10000
pgstattuple=# DELETE FROM demo_bloat WHERE i < 9000;
DELETE 8999
```

L'extension pgstattuple permet de mesurer précisément l'espace libre d'une table, à condition d'avoir déclenché un `VACUUM`. Les colonnes `free_space` et `free_percent` donnent la taille et le pourcentage d'espace libre.

```
pgstattuple=# SELECT * FROM pgstattuple('demo_bloat');
-[ RECORD 1 ]------+-------
table_len          | 368640
tuple_count        | 1001
tuple_len          | 28028
tuple_percent      | 7.6
dead_tuple_count   | 8999
dead_tuple_len     | 251972
dead_tuple_percent | 68.35
```

```
free_space          | 7380
free_percent        | 2

pgstattuple=# VACUUM demo_bloat ;
VACUUM
pgstattuple=# SELECT * FROM pgstattuple('demo_bloat');
-[ RECORD 1 ]------+-------
table_len          | 368640
tuple_count        | 1001
tuple_len          | 28028
tuple_percent      | 7.6
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 295348
free_percent       | 80.12
```

L'estimation retournée par les requêtes proposées plus haut ne sont pas loin de la réalité :

```
pgstattuple=# \i table_bloat.sql
(...)
-[ RECORD 41 ]---+------------------------
current_database | pgstattuple
schemaname       | public
tblname          | demo_bloat
real_size        | 368640
extra_size       | 327680
extra_ratio      | 88.8888888888889
fillfactor       | 100
bloat_size       | 327680
bloat_ratio      | 88.8888888888889
is_na            | f
```

4.4.12 VUE PG_STAT_USER_TABLES

- Statistiques niveau «ligne»
- Nombre de lignes insérées/mises à jour/supprimées
- Type et nombre d'accès
- Opérations de maintenance
- Détection des tables mal indexées ou très accédées

Contrairement aux vues précédentes, cette vue est locale à chaque base.

Voici la définition de ses colonnes :

- `relid`, `relname` : `OID` et nom de la table concernée.
- `schemaname` : le schéma contenant cette table.
- `seq_scan` : nombre de parcours séquentiels sur cette table.
- `seq_tup_read` : nombre d'enregistrements accédés par ces parcours séquentiels.
- `idx_scan` : nombre de parcours d'index sur cette table.
- `idx_tup_fetch` : nombre d'enregistrements accédés par ces parcours séquentiels.
- `n_tup_ins`, `n_tup_upd`, `n_tup_del` : nombre d'enregistrements insérés, mis à jour, supprimés.
- `n_tup_hot_upd` : nombre d'enregistrements mis à jour par mécanisme `HOT` (c'est-à-dire sur place).
- `n_live_tup` : nombre d'enregistrements «vivants».
- `n_dead_tup` : nombre d'enregistrements «morts» (supprimés mais non nettoyés) depuis le dernier `VACUUM`.
- `n_mod_since_analyze` : nombre d'enregistrements modifiés depuis le dernier `ANALYZE`.
- `last_vacuum` : timestamp de dernier `VACUUM`.
- `last_autovacuum` : timestamp de dernier `VACUUM` automatique.
- `last_analyze` : timestamp de dernier `ANALYZE`.
- `last_autoanalyze` : timestamp de dernier `ANALYZE` automatique.
- `vacuum_count` : nombre de `VACUUM` manuels.
- `autovacuum_count` : nombre de `VACUUM` automatiques.
- `analyze_count` : nombre d'`ANALYZE` manuels.
- `autoanalyze_count` : nombre d'`ANALYZE` automatiques.

Les colonnes `vacuum_count`, `autovacuum_count`, `analyze_count`, `autoanalyze_count` apparaissent en 9.1. La colonne `n_mod_since_analyze` apparaît en 9.4.

4.4.13 VUE PG_STAT_USER_INDEXES

- Vue par index
- Nombre d'accès et efficacité

Voici la liste des colonnes de cette vue :

- `relid`, `relname` : `OID` et nom de la table qui possède l'index
- `indexrelid`, `indexrelname` : `OID` et nom de l'index en question
- `schemaname` : schéma contenant l'index

- `idx_scan` : nombre de parcours de cet index
- `idx_tup_read` : nombre d'enregistrements retournés par cet index
- `idx_tup_fetch` : nombre d'enregistrements accédés sur la table associée à cet index

`idx_tup_read` et `idx_tup_fetch` retournent des valeurs différentes pour plusieurs raisons :

- Un parcours d'index peut très bien accéder à des enregistrements morts. Dans ce cas, la valeur de `idx_tup_read` sera supérieure à celle de `idx_tup_fetch`.
- Un parcours d'index peut très bien ne pas entraîner d'accès direct à la table :
  - si c'est un Index Only Scan, on n'accède moins fortement (voire pas du tout) à la table puisque toutes les colonnes accédées sont dans l'index
  - si c'est un Bitmap Index Scan, on va éventuellement accéder à plusieurs index, faire une fusion (Or ou And) et ensuite seulement accéder aux enregistrements (moins nombreux si c'est un And).

Dans tous les cas, ce qu'on surveille le plus souvent dans cette vue, c'est tout d'abord les index ayant `idx_scan` à 0. Ils sont le signe d'un index qui ne sert probablement à rien. La seule exception éventuelle étant un index associé à une contrainte d'unicité (et donc aussi les clés primaires), les parcours de l'index réalisés pour vérifier l'unicité n'étant pas comptabilisés dans cette vue.

Les autres indicateurs intéressants sont un nombre de `tup_read` très grand par rapport aux parcours d'index, qui peuvent suggérer un index trop peu sélectif, et une grosse différence entre les colonnes `idx_tup_read` et `idx_tup_fetch`. Ces indicateurs ne permettent cependant pas de conclure quoi que ce soit par eux-même, ils peuvent seulement donner des pistes d'amélioration.

4.4.14 VUE PG_STATIO_USER_{TABLES,INDEXES}

- Opérations au niveau bloc
- Demandés au système ou trouvés dans le cache de PostgreSQL
- Calculer des hit ratios

Voici la description des différentes colonnes de `pg_statio_user_tables` :

- `relid`,`relname` : `OID` et nom de la table.
- `schemaname` : nom du schéma contenant la table.
- `heap_blks_read` : nombre de blocs accédés de la table demandés au système d'exploitation. `Heap` signifie *tas*, et ici *données non triées*, par opposition aux index.

- heap_blks_hit : nombre de blocs accédés de la table trouvés dans le cache de PostgreSQL.
- idx_blks_read : nombre de blocs accédés de l'index demandés au système d'exploitation.
- idx_blks_hit : nombre de blocs accédés de l'index trouvés dans le cache de PostgreSQL.
- toast_blks_read, toast_blks_hit, tidx_blks_read, tidx_blks_hit : idem que précédemment, mais pour la partie TOAST des tables et index.

Et voici la description des différentes colonnes de pg_statio_user_indexes :

- relid, relname : OID et nom de la table associée à l'index
- indexrelid, indexrelname : OID et nom de l'inde
- schemaname : nom du schéma dans lequel se trouve l'index
- idx_blks_read : nombre de blocs accédés de l'index demandés au système d'exploitation.
- idx_blks_hit : nombre de blocs accédés de l'index trouvés dans le cache de PostgreSQL.

Pour calculer un *hit ratio*, qui est un indicateur fréquemment utilisé, on utilise la formule suivante (cet exemple cible uniquement les index) :

```
SELECT schemaname,
       indexrelname,
       relname,
       idx_blks_hit::float/CASE idx_blks_read+idx_blks_hit
           WHEN 0 THEN 1 ELSE idx_blks_read+idx_blks_hit END
FROM pg_statio_user_indexes;
```

Notez que :

- idx_blks_hit::float convertit le numérateur en type float, ce qui entraîne que la division est à virgule flottante. Sinon on divise des entiers par des entiers, et on obtient donc un résultat entier, 0 la plupart du temps (division euclidienne entière).
- la clause suivante évite la division par zéro, en divisant par 1 quand les deux compteurs sont à 0 :

```
CASE idx_blks_read+idx_blks_hit
WHEN 0 THEN 1
ELSE idx_blks_read + idx_blks_hit
END
```

4.4.15 SURVEILLER L'ACTIVITÉ SQL

- Quelles sont les requêtes lentes ?
- Quelles sont les requêtes les plus fréquentes ?
- Quelles requêtes génèrent des fichiers temporaires ?
- Quelles sont les requêtes bloquées ?
  - et par qui ?
- Progression d'une requête

4.4.16 TRACE DES REQUÊTES EXÉCUTÉES

- `log_min_duration_statements` =
  - `0` permet de tracer toutes les requêtes
  - trace des paramètres
  - traces exploitables par des outils tiers
  - pas d'informations sur les accès, ni des plans d'exécution
- D'autres paramètres existent mais sont peu intéressants

Le paramètre `log_min_duration_statements` permet d'activer une trace sélective des requêtes lentes. Le paramètre accepte plusieurs valeurs :

- `-1` pour désactiver la trace,
- `0` pour tracer systématiquement toutes les requêtes exécutées,
- une durée en millisecondes pour tracer les requêtes que l'on estime être lentes.

Si le temps d'exécution d'une requête dépasse le seuil défini par le paramètre `log_min_duration_statements`, PostgreSQL va alors tracer le temps d'exécution de la requête, ainsi que ces paramètres éventuels. Par exemple :

```
LOG:  duration: 43.670 ms  statement:
    SELECT DISTINCT c.numero_commande,
    c.date_commande, lots.numero_lot, lots.numero_suivi FROM commandes c JOIN
    lignes_commandes l ON (c.numero_commande = l.numero_commande) JOIN lots
    ON (l.numero_lot_expedition = lots.numero_lot)
    WHERE c.numero_commande = 72199;
```

Ces traces peuvent ensuite être exploitées par l'outil pgBadger qui pourra établir un rapport des requêtes les plus fréquentes, des requêtes les plus lentes, etc.

4.4.17 TRACE DES FICHIERS TEMPORAIRES

- log_temp_files = <taille minimale>
  - 0 trace tous les fichiers temporaires
  - associe les requêtes SQL qui les génèrent
  - traces exploitable par des outils tiers

Le paramètre log_temp_files permet de tracer les fichiers temporaires générés par les requêtes SQL. Il est généralement positionné à 0 pour tracer l'ensemble des fichiers temporaires, et donc de s'assurer que l'instance n'en génère que rarement.

Par exemple, la trace suivante est produite lorsqu'une requête génère un fichier temporaire :

```
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp2181.0", size 276496384
STATEMENT:  select * from lignes_commandes order by produit_id;
```

Si une requête nécessite de générer plusieurs fichiers temporaires, chaque fichier temporaire sera tracé individuellement. pgBadger permet de réaliser une synthèse des fichiers temporaires générés et propose un rapport sur les requêtes générant le plus de fichiers temporaires et permet donc de cibler l'optimisation.

4.4.18 VUE PG_STAT_STATEMENTS

- Extension disponible depuis la 8.4
  - réellement intéressante à partir de la 9.2
- Ajoute une nouvelle vue statistique, appelée pg_stat_statements
- Les requêtes sont normalisées
- Indique les requêtes exécutées, avec durée d'exécution, utilisation du cache, etc.

pg_stat_statements est un module contrib apparaissant en 8.4. Contrairement à pgbadger, il ne nécessite pas de tracer les requêtes exécutées. Il est connecté directement à l'exécuteur de requêtes qui fait appel à lui à chaque fois qu'il a exécuté une requête. pg_stat_statements a ainsi accès à beaucoup d'informations. Certaines sont placées en mémoire partagée et accessible via une vue statistique appelée pg_stat_statements.

Voici un exemple de requête sur la vue pg_stat_statements :

```
postgres=# SELECT * FROM pg_stat_statements ORDER BY total_time DESC LIMIT 3;
-[ RECORD 1 ]---------------------------------------------------------
userid     | 10
dbid       | 63781
```

```
query      | UPDATE branches SET bbalance = bbalance + $1 WHERE bid = $2;
calls      | 3000
total_time | 20.716706
rows       | 3000
[...]
-[ RECORD 2 ]----------------------------------------------------------
userid     | 10
dbid       | 63781
query      | UPDATE tellers SET tbalance = tbalance + $1 WHERE tid = $2;
calls      | 3000
total_time | 17.1107649999999
rows       | 3000
[...]
```

`pg_stat_statements` possède des paramètres de configuration pour indiquer le nombre maximum d'instructions tracées, la sauvegarde des statistiques entre chaque démarrage du serveur, etc.

4.4.19 VUE PG_STAT_STATEMENTS - MÉTRIQUES 1/2

Métriques intéressantes :

- Durée d'exécution :
  - `total_time`
  - `min_time`/`max_time` (9.5+)
  - `stddev_time` (9.5+)
  - `mean_time` (9.5+)
- Nombre de lignes retournées : `rows`

`pg_stat_statements` apporte des statistiques sur les durées d'exécutions des requêtes normalisées. Ainsi, `total_time` indique le cumul d'exécution total. Cette métrique peut s'avérer insuffisante, de nouvelles métriques sont donc apparues avec la version 9.5 :

- `min_time` et `max_time` : Donne la durée d'exécution minimale et maximale d'une requête normalisée
- `mean_time` : Donne la durée moyenne d'exécution
- `stddev_time` : Donne l'écart-type de la durée d'exécution. Cette métrique peut être intéressante pour identifier une requête dont le temps d'exécution varie fortement.

La métrique `row` indique le nombre total de lignes retournées.

4.4.20 VUE PG_STAT_STATEMENTS - MÉTRIQUES 2/2

- Accès à la mémoire partagée
  - `shared_blks_hit/read/dirtied/written`
- Accès à la mémoire locale (objets dédiés à la session comme les tables temporaires)
  - `local_blks_hit/read/dirtied/written`
- Lecture/écriture de fichier temporaire
  - `temp_blks_read/written`
- Temps d'accès en entrée/sortie
  - `blk_read_time/blk_write_time`

`pg_stat_statements` fournit également des métriques sur les accès aux blocs :

Lors des accès à la mémoire partagée (*shared_buffers*) les compteurs suivants peuvent être incrémentés :

- `shared_blks_hit` : Lorsque les lectures se font directement dans le cache.
- `shared_blks_read` : Lorsque les lectures nécessitent une lecture sur le disque.
- `shared_blks_dirtied` : Lorsque la requête génère des blocs *sales* (`dirty`) qui seront nettoyés ultérieurement par le `Background Writer` ou le `Checkpointer`.
- shared_blks_written‘ : Lorsque les accès à des blocs nécessitent des écritures sur disque. Ce cas peut arriver lorsqu'il n'y a plus pages disponibles en mémoire partagée et que le processus backend doit nettoyer des pages "sales" (*dirty*) sur disque pour libérer des pages en mémoire partagée.

Il existe les même métriques mais pour les accès à la mémoire du backend utilisée pour les objets temporaires : `local_blks_*` Ces derniers ne nécessitent pas d'être partagés avec les autres sessions comme les tables temporaires, index sur tables temporaires...

Les métriques `temp_blks_read` et `temp_blks_written` correspondent au nombre de blocs lus et écris depuis le disque dans des fichiers temporaires. Par exemple lorsqu'un tri ne rentre pas dans la `work_mem`.

Enfin les métriques suivantes donnent le cumul des durées de lectures et écritures des accès sur disques si le paramètre `track_io_timing` est activé :

`blk_read_time / blk_write_time`

4.4.21 REQUÊTES BLOQUÉES

- Vue `pg_stat_activity`
  - colonnes `wait_event` et `wait_event_type`
- Vue `pg_locks`
  - colonne `granted`
- Fonction `pg_blocking_pids`

Lors de l'exécution d'une requête, le processus chargé de cette exécution va tout d'abord récupérer les verrous dont il a besoin. En cas de conflit, la requête est mise en attente. Cette attente est visible à deux niveaux :

- au niveau des sessions, via les colonnes `wait_event` et `wait_event_type` de la vue `pg_stat_activity`
- au niveau des verrous, via la colonne `granted` de la vue `pg_locks`

Il est ensuite assez simple de trouver qui bloque qui. Prenons par exemple deux sessions, une dans une transaction qui a lu une table :

```
postgres=# begin;
BEGIN
postgres=# select * from t2 limit 1;
 id
----
(0 rows)
```

La deuxième session cherche à supprimer cette table :

```
postgres=# drop table t2;
```

Elle se trouve bloquée. La première session ayant lu cette table, elle a posé pendant la lecture un verrou d'accès partagé (`AccessShareLock`) pour éviter une suppression ou une redéfinition de la table pendant la lecture. Les verrous étant conservés pendant toute la durée d'une transaction, la transaction restant ouverte, le verrou reste. La deuxième session veut supprimer la table. Pour réaliser cette opération, elle doit obtenir un verrou exclusif sur cette table, verrou qu'elle ne peut pas obtenir vu qu'il y a déjà un autre verrou sur cette table. L'opération de suppression est donc bloquée, en attente de la fin de la transaction de la première session. Comment peut-on le voir ? tout simplement en interrogeant les tables `pg_stat_activity` et `pg_locks`.

Avec `pg_stat_activity`, nous pouvons savoir quelle session est bloquée :

```
postgres=# select pid, query from pg_stat_activity where wait_event is not null AND ba
  pid  |     query
-------+----------------
```

```
 17396 | drop table t2;
(1 row)
```

Pour savoir de quel verrou a besoin le processus 17396, il faut interroger la vue pg_locks :

```
postgres=# SELECT locktype, relation, pid, mode, granted FROM pg_locks
WHERE pid=17396 AND NOT granted;
 locktype | relation |  pid  |        mode         | granted
----------+----------+-------+---------------------+---------
 relation |    24581 | 17396 | AccessExclusiveLock | f
(1 row)
```

Le processus 17396 attend un verrou sur la relation 24581. Reste à savoir qui dispose d'un verrou sur cet objet :

```
postgres=# SELECT locktype, relation, pid, mode, granted FROM pg_locks
WHERE relation=24581 AND granted;
 locktype | relation |  pid  |      mode       | granted
----------+----------+-------+-----------------+---------
 relation |    24581 | 17276 | AccessShareLock | t
(1 row)
```

Il s'agit du processus 17276. Et que fait ce processus ?

```
postgres=# SELECT usename, datname, state, query
FROM pg_stat_activity WHERE pid=17276;
 usename  | datname  |        state        |           query
----------+----------+---------------------+---------------------------
 postgres | postgres | idle in transaction | select * from t2 limit 1;
(1 row)
```

Nous retrouvons bien notre session en transaction.

Notons que la version 9.6 nous permet d'aller plus vite. Elle fournit une fonction nommée pg_blocking_pids() renvoyant les PID des sessions bloquant une session particulière.

```
postgres=# SELECT pid, pg_blocking_pids(pid)
FROM pg_stat_activity WHERE wait_event IS NOT NULL;
  pid  | pg_blocking_pids
-------+------------------
 17396 | {17276}
(1 row)
```

Le processus 17276 bloque bien le processus 17396.

4.4.22 PROGRESSION D'UNE REQUÊTE

- API de progression de requêtes
- Utilisé par la commande `VACUUM`
- À partir de la 9.6

La version 9.6 implémente une API pour surveiller la progression de l'exécution d'une requête. Cette API est utilisée actuellement uniquement par la commande `VACUUM`.

Il est donc possible de suivre l'exécution d'un `VACUUM` par l'intermédiaire de la vue `pg_stat_progress_vacuum`. Elle contient une ligne par `VACUUM` en cours d'exécution. Voici un exemple de son contenu :

```
pid                 | 4299
datid               | 13356
datname             | postgres
relid               | 16384
phase               | scanning heap
heap_blks_total     | 127293
heap_blks_scanned   | 86665
heap_blks_vacuumed  | 86664
index_vacuum_count  | 0
max_dead_tuples     | 291
num_dead_tuples     | 53
```

Dans cet exemple, le `VACUUM` exécuté par le PID 4299 a parcouru 86665 blocs (soit 68% de la table), et en a traité 86664.

4.4.23 SURVEILLER LES ÉCRITURES

- Quelle quantité de données sont écrites ?
- Quel canal d'écriture est utilisé ?

4.4.24 TRACE DES CHECKPOINTS

- `log_checkpoints = on`
- Affiche des informations à chaque checkpoint :
  - mode de déclenchement
  - volume de données écrits

  - durée du `checkpoint`
- Trace exploitable par des outils tiers

Le paramètre `log_checkpoints`, lorsqu'il est actif, permet de tracer les informations liées à chaque checkpoint déclenché.

PostgreSQL va produire une trace de ce type pour un checkpoint déclenché par `checkpoint_timeout` :

```
LOG:  checkpoint starting: time
LOG:  checkpoint complete: wrote 56 buffers (0.3%); 0 transaction log file(s)
      added, 0 removed, 0 recycled; write=5.553 s, sync=0.013 s, total=5.573 s;
      sync files=9, longest=0.004 s, average=0.001 s; distance=464 kB,
      estimate=2153 kB
```

Un outil comme `pgBadger` peut exploiter ces informations.

4.4.25 VUE PG_STAT_BGWRITER

- Activité des écritures dans les fichiers de données
- Visualisation du volume d'allocations et d'écritures

Cette vue ne comporte qu'une seule ligne. Voici la signification de ses colonnes :

- `checkpoints_timed` : nombre de checkpoints déclenchés par `checkpoint_timeout`.
- `checkpoints_req` : nombre de checkpoints déclenchés par `checkpoint_segments` (ou `max_wal_size` à partir de la version 9.5).
- `checkpoint_write_time` : temps passé par `checkpointer` à écrire des données.
- `checkpoint_sync_time` : temps passé à s'assurer que les écritures ont été synchronisées sur disque lors des checkpoints.
- `buffers_checkpoint` : nombre de blocs écrits par `checkpointer`.
- `buffers_clean` : nombre de blocs écrits par `writer`.
- `maxwritten_clean` : nombre de fois où `writer` s'est arrêté pour avoir atteint la limite configurée par `bgwriter_lru_maxpages`.
- `buffers_backend` : nombre de blocs écrits par les backends avant de pouvoir allouer de la mémoire (car pas de bloc disponible).
- `buffers_backend_fsync` : nombre de blocs synchronisés par les backends parce que la liste des blocs à synchroniser est pleine.
- `buffers_alloc` : nombre de blocs alloués dans le `shared_buffers`.
- `stats_reset` : date de remise à zéro de cette vue statistique

17.12

Les deux premières colonnes permettent de vérifier que la configuration de `checkpoint_segments` (ou `max_wal_size` à partir de la version 9.5) n'est pas trop basse par rapport au volume d'écriture que subit la base. Les colonnes `buffers_clean` (à comparer à `buffers_checkpoint` et `buffers_backend`) et `maxwritten_clean` permettent de vérifier que la configuration du bgwriter est adéquate : si `maxwritten_clean` augmente fortement en fonctionnement normal, c'est que le paramètre `bgwriter_lru_maxpages` l'empêche de libérer autant de buffers qu'il l'estime nécessaire (ce paramètre sert de garde fou). Dans ce cas, `buffers_backend` va augmenter.

Il faut toutefois prendre ce compteur avec prudence : une session qui modifie énormément de blocs n'aura pas le droit de modifier tout le contenu du cache disque, elle sera cantonnée à une toute petite partie. Elle sera donc obligée de vider elle-même ses buffers. C'est le cas par exemple d'une session chargeant un volume conséquent de données avec `COPY`.

Cette vue statistique peut être mise à zéro par l'appel à la fonction :

```
pg_stat_reset_shared('bgwriter')
```

4.5 OUTILS D'ANALYSE

- Différents outils existent autour de PostgreSQL
- Outils d'analyse occasionnel :
  - `pg_activity`
- Outils d'analyse des traces :
  - `pgbadger`
- Outils d'analyse des statistiques :
  - `pgCluu`, `OPM`
  - `pg_stat_statements`, `PoWA`

Différents outils d'analyse sont apparus pour superviser les performances d'un serveur PostgreSQL. Ce sont généralement des outils développés par la communauté, mais qui ne sont pas intégrés au moteur. Par contre, ils utilisent les fonctionnalités du moteur.

4.5.1 PG_ACTIVITY

- Script Python
- `top` pour PostgreSQL

- Affiche :
  - les requêtes en cours
  - les sessions bloquées
  - les sessions bloquantes
- Dépôt github[38]

`pg_activity` est un projet libre qui apporte une fonctionnalité équivalent à top, mais appliqué à PostgreSQL. Elle dispose de trois écrans qui affichent chacuns les requêtes en cours, les sessions bloquées et les sessions bloquantes.

4.5.2 PGBADGER

- Script Perl
- Traite les journaux applicatifs
- Recherche des informations sur les requêtes
- Génération d'un rapport HTML très détaillé
- Dépôt github[39]

pgBadger est un projet sous licence BSD très actif. Le site officiel se trouve sur http://projects.dalibo.org/pgbadger.

Voici une liste des options les plus utiles :

- `--top` : nombre de requêtes à afficher, par défaut 20
- `--extension` : format de sortie (html, text, bin, json ou tsung)
- `--dbname` : choix de la base à analyser
- `--prefix` : permet d'indiquer le format utilisé dans les logs.

4.5.3 PGCLUU

- Outils de collectes de métriques de performances
  - Dépôt github[40]
  - génère un rapport HTML complet
- Différents aspects mesurés :
  - informations sur le système
  - consommation des ressources CPU, RAM, I/O

[38] https://github.com/julmon/pg_activity/
[39] http://dalibo.github.io/pgbadger/
[40] https://github.com/darold/pgcluu

  - utilisation de la base de données

4.5.4 OPEN POSTGRESQL MONITORING

- Plate-forme de supervision dédiée à PostgreSQL
  - extensible
  - sondes adaptées à PostgreSQL
  - graphes
  - alertes
- Site officiel[41]
- Site de démonstration[42]
  - `opm/demo`

OPM est un projet qui a pour vocation à fournir à la communauté PostgreSQL un outil aussi puissant que leurs équivalents propriétaires tels qu'Oracle Grid Control. Il repose actuellement sur une architecture assez complexe, basée sur Nagios. Les prochaines évolutions amèneront néanmoins la possibilité de déployer un agent de supervision offrant plus de possibilités.

Le coeur de l'architecture OPM est une base de données qui centralise les métriques de performance en provenance des serveurs supervisés par Nagios :

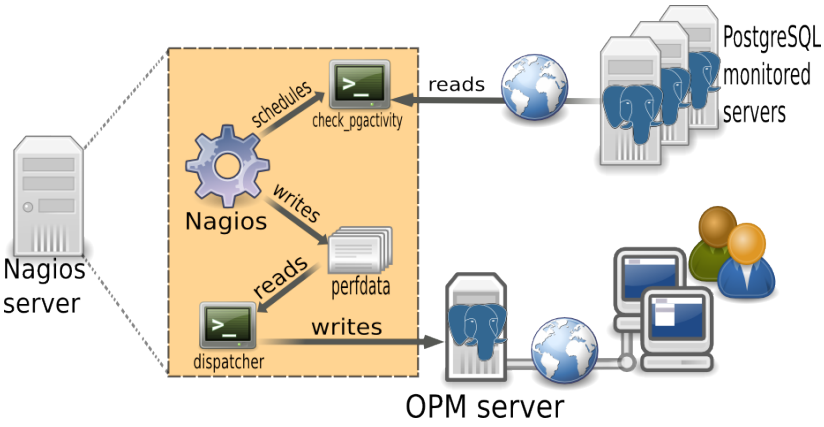


FIGURE 9: ARCHITECTURE OPM

[41] http://opm.io/
[42] http://demo.opm.io/

4.5.5 POSTGRESQL WORKLOAD ANALYZER

- Objectif : identifier les requêtes coûteuses
  - sans devoir accéder aux logs
  - quasi en temps-réel
- Background worker
  - dépendant de `pg_stat_statements`
  - nécessite PostgreSQL 9.4
- Site officiel[43]

Aucune historisation n'est en effet réalisée par `pg_stat_statements`. PoWA a été développé pour combler ce manque et ainsi fournir un outil équivalent à AWR d'Oracle, permettant de connaître l'activité du serveur sur une période donnée.

Sur l'instance de production de Dalibo, la base de données PoWA occupe moins de 300 Mo sur disque, avec les caractéristiques suivantes :

- 10 jours de rétention
- fréquence de capture : 1 min
- 17 bases de données
- 45263 requêtes normalisées
- dont ~28000 `COPY`, ~11000 `LOCK`
- dont 5048 requêtes applicatives

4.6 CONCLUSION

- Un système est pérenne s'il est bien supervisé
- Les système de supervision automatique ont souvent besoin d'être complété
- PostgreSQL fourni énormément d'indicateurs utiles à la supervision
- Les outils de supervision ponctuels sont utiles pour rapidement diagnostiquer l'état d'un serveur

Une bonne politique de supervision est la clef de voûte d'un système pérenne. Pour cela, il faut tout d'abord s'assurer que les traces et les statistiques soient bien configurées. Ensuite, s'intéresser à la métrologie et compléter ou installer un système de supervision avec des indicateurs compréhensibles.

[43] http://dalibo.github.io/powa/